Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: elektromechanica

Masterthesis

Comparative Analysis of QUIC and Other Network Protocols for Real-Time Robotic Control

Warre Clerix

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektromechanica

PROMOTOR :

dr. Nikolaos TSIOGKAS

Gezamenlijke opleiding UHasselt en KU Leuven

Universiteit Hasselt | Campus Diepenbeek | Faculteit Industriële Ingenieurswetenschappen | Agoralaan Gebouw H - Gebouw B | BE 3590 Diepenbeek

Universiteit Hasselt | Campus Diepenbeek | Agoralaan Gebouw D | BE 3590 Diepenbeek
Universiteit Hasselt | Campus Hasselt | Martelarenlaan 42 | BE 3500 Hasselt

2024
2025

Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: elektromechanica

Masterthesis

Comparative Analysis of QUIC and Other Network Protocols for Real-Time Robotic Control

Warre Clerix

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektromechanica

PROMOTOR :

dr. Nikolaos TSIOGKAS

UHASSELT KU LEUVEN

Preface

As robots increasingly operate in remote and challenging environments—from surgical suites to disaster zones—the reliability and performance of their communication systems become critical factors determining not just operational success, but human safety. This thesis explores how emerging network protocols can enhance real-time robotic control capabilities.

The research presented here was conducted within the ACRO (Automation, Computer vision and Robotics) research group at the Diepenbeek Campus of UHasselt and KULeuven during the academic year 2024-2025, focusing on evaluating QUIC, DCCP, and SCTP protocols for haptic teleoperation systems where sub-millisecond timing precision is crucial.

I would like to express my sincere gratitude to my supervisor, Dr. Nikolaos Tsiogkas, for his guidance, expertise, and patience throughout this research. His insights into both theoretical foundations and practical implementation challenges were invaluable in shaping this work.

My appreciation extends to the ACRO research group and the faculty at UHasselt for providing access to laboratory facilities, haptic devices, and computational resources that made this experimental work possible. Special thanks also go to the developers of the open-source tools and libraries, particularly the ROS2 community and the Geomagic Touch driver maintainers, whose work enabled the implementation framework for this research.

I am grateful to my family and friends for their unwavering support throughout my studies. Their encouragement during the challenging phases of research and writing was essential to bringing this work to completion.

While this thesis represents the culmination of my master's studies, I hope it serves as a foundation for continued advancement in networked robotic systems that can operate safely and effectively in demanding environments.

Clerix Warre
Diepenbeek, June 9, 2025

Contents

List of Tables

List of Figures

Glossary of Terms

ACK	Acknowledgment; a signal sent by a receiver to indicate successful receipt of data.
ACRO	Automation, Computer vision and Robotics research group at the Diepenbeek Campus.
Congestion Control	Mechanisms that prevent network congestion by regulating the rate at which data is sent.
DCCP	Datagram Congestion Control Protocol; provides congestion control without requiring full connection establishment, optimized for real-time applications.
Haptic Interface	A device that enables interaction with a computer through touch and motion, providing tactile feedback. In this research, the Geomagic Touch device is used.
Head-of-Line Blocking	A performance-limiting phenomenon that occurs when a line of packets is held up by the first packet.
Jitter	Variation in the delay (latency) of received packets, measured in milliseconds.
Latency	The time delay between the moment data is sent and when it is received, typically measured in milliseconds.
MsQuic	Microsoft's implementation of the QUIC protocol, used in this research for client-server communication.
Multi-homing	A network configuration technique where a device has multiple network interfaces or IP addresses for increased reliability.
Multi-streaming	The ability to send multiple independent streams of data concurrently over a single connection.
Packet Loss	The failure of one or more transmitted packets to arrive at their destination, usually expressed as a percentage.
QUIC	Quick UDP Internet Connections; a transport layer protocol initially developed by Google that combines the speed of UDP with the reliability of TCP.
Real-time Control	Control systems that process inputs and generate outputs within strict time constraints, essential for robotic applications.
ROS2	Robot Operating System 2; an open-source middleware suite for robot software development used in this research.
RTT	Round-Trip Time; the time it takes for a signal to be sent plus the time it takes for an acknowledgment of that signal to be received.

SCTP	Stream Control Transmission Protocol; provides features for multi-streaming and multi-homing capabilities for improved reliability.
TCP	Transmission Control Protocol; a connection-oriented protocol that guarantees reliable, ordered delivery of data.
Teleoperation	The operation of a machine or system from a distance, often used in hazardous environments.
Throughput	The amount of data moved successfully from one place to another in a given time period, typically measured in bits per second.
TLS	Transport Layer Security; cryptographic protocols designed to provide secure communication over a computer network.
UDP	User Datagram Protocol; a connectionless protocol that offers low-latency data transmission without reliability guarantees.
0-RTT	Zero Round Trip Time; a feature in QUIC that enables faster connection establishment for previously visited servers.

Abstract

Remote robotic systems performing surgery, nuclear maintenance, and disaster response need fast, reliable communication to operate safely. Traditional protocols present serious trade-offs: TCP introduces too much latency for real-time control, while UDP struggles with high packet loss. This research evaluates three emerging protocols—QUIC, DCCP, and SCTP—to find better solutions for demanding robotic applications.

A testing system was built using a haptic interface that provides tactile feedback, implementing all three protocols with equivalent architectures in ROS2. Performance metrics, including latency, jitter, throughput, packet loss, and connection stability, were measured under controlled network conditions designed for real-time control requirements.

The results show clear differences in performance. QUIC achieved the best overall results with 1.198 ms average latency and 98.5% message delivery reliability. DCCP delivered moderate performance at 2.45 ms latency but only 86.6% reliability, making it suitable when some message loss is acceptable. SCTP provided strong reliability at 97.6% but with a higher latency of 5.231 ms. QUIC's combination of speed and dependability makes it especially well-suited for critical robotic applications.

This research provides practical guidelines and a decision framework for protocol selection, supporting the development of more robust networked robotic systems for medical, industrial, and emergency response applications.

Abstract in het Nederlands

Remote robotica voor chirurgie, nucleair onderhoud en rampenbestrijding vereist snelle en betrouwbare communicatie om veilig te kunnen opereren. Traditionele protocollen brengen lastige compromissen met zich mee: TCP zorgt voor te veel vertraging voor real-time besturing, terwijl UDP kampt met veel pakketverlies. Dit onderzoek evalueert drie opkomende protocollen—QUIC, DCCP en SCTP—om betere oplossingen te vinden voor veeleisende robottoepassingen.

Een testsysteem werd gebouwd met een haptische interface die tactiele feedback biedt, waarbij alle drie protocollen met een gelijke architectuur in ROS2 zijn geïmplementeerd. Prestatie-indicatoren zoals latentie, jitter, doorvoersnelheid, pakketverlies en verbindingsstabiliteit werden gemeten onder gecontroleerde netwerkomstandigheden die aansluiten bij real-time eisen.

De resultaten tonen duidelijke prestatieverschillen. QUIC behaalde de beste algehele resultaten met een gemiddelde latentie van 1,198 ms en 98,5% berichtbetrouwbaarheid. DCCP presteerde matig met 2,45 ms latentie en 86,6% betrouwbaarheid, wat het geschikt maakt wanneer enig verlies aanvaardbaar is. SCTP leverde sterke betrouwbaarheid (97,6%) maar met een hogere latentie van 5,231 ms. De combinatie van snelheid en betrouwbaarheid maakt QUIC bijzonder geschikt voor kritieke robottoepassingen.

Dit onderzoek biedt praktische richtlijnen en een besliskader voor protocolkeuze, ter ondersteuning van robuuste netwerkrobotica in medische, industriële en noodtoepassingen.

Chapter 1

Introduction

In today's industrial and remote environments, robotic systems require precise and timely control to perform complex operations safely and efficiently. The communication backbone that enables these systems to operate depends heavily on network protocols that can deliver commands with minimal latency while maintaining reliability. Traditional protocols like TCP [1] and UDP [2] have served adequately in many scenarios, but as robotic applications expand into more demanding domains such as telesurgery, hazardous environment intervention, and high-precision manufacturing, the limitations of these conventional protocols become increasingly apparent [3].

1.1 Context

The integration of robotic systems in industrial and remote operations has created a growing need for network protocols that can support real-time control requirements. This research is conducted within the ACRO research group at the Diepenbeek Campus, focusing on comparing emerging network protocols for their suitability in real-time robotic control applications.

Traditional network protocols present distinct tradeoffs: TCP offers reliability through connection-oriented communication but introduces higher latency due to its handshaking and acknowledgment mechanisms. UDP, while providing lower latency through connectionless communication, lacks reliability guarantees as packets may be lost or arrive out of order [2], [1]. These limitations can significantly impact safety-critical robotic operations where both low latency and reliability are essential.

In recent years, several alternative protocols have emerged that attempt to bridge this gap, including QUIC (Quick UDP Internet Connections) [4], [5], DCCP (Datagram Congestion Control Protocol) [6], [7], and SCTP (Stream Control Transmission Protocol) [8], [9]. Each offers unique features that may address the specific demands of real-time robotic control.

As shown in Figure 3.2, the experimental setup employed in this research utilizes a Geomagic Touch haptic interface device to determine 3D spatial coordinates for robot movement. These positions are captured at high frequency and converted to standardized ROS messages. The messages are then transmitted via a client application using the network protocol under test (either QUIC, DCCP, or SCTP) to a server running on a host PC. The server processes these commands and forwards appropriate control signals to the robot arm, completing the control

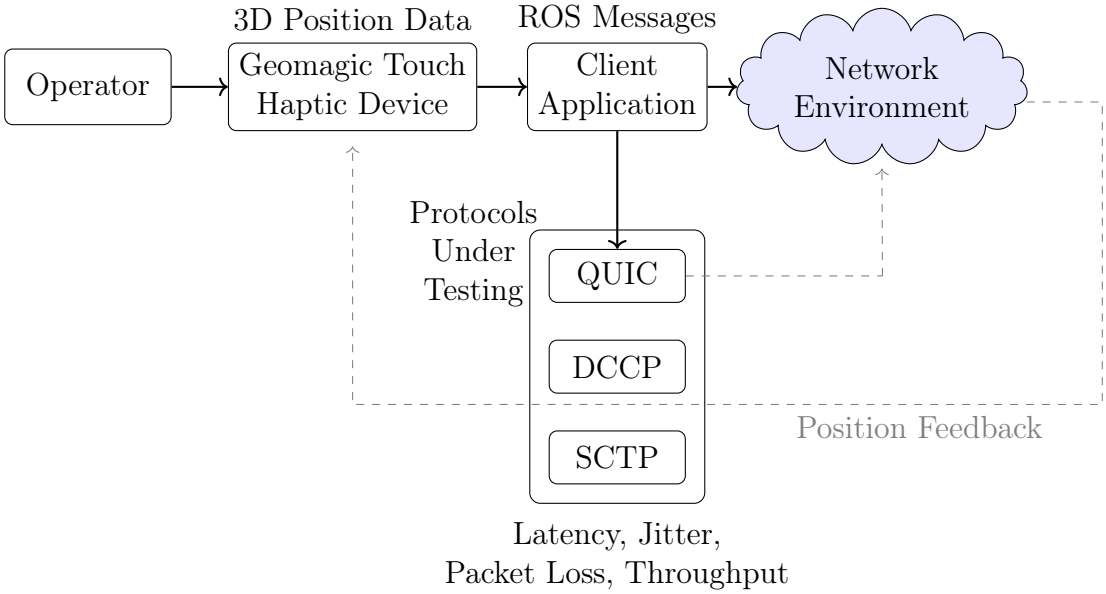

Figure 1.1: Simplified representation of the haptic control system using network protocols for real-time robotic control. Position data from the haptic device is transmitted through one of the tested protocols (QUIC, DCCP, or SCTP) across varying network conditions to control the remote robot arm. Performance metrics are measured to compare protocol efficiency.

loop. This setup allows for precise measurement of protocol performance throughout the entire command chain, from human operator input to robot actuation.

This research focuses on the network communication layer of a larger teleoperation system. The robot control algorithms and actuator interfaces were implemented by previous researchers and read standardized ROS2 messages to control the physical robot. This study evaluates how different network protocols affect the delivery of haptic commands to this existing control system.

1.2 Problem Statement

Robots operating in remote or hazardous environments, such as nuclear facilities or disaster zones, require low-latency and highly reliable control mechanisms to ensure safe and precise operations. Current implementations predominantly rely on TCP or UDP, neither of which fully satisfies the demanding requirements of real-time robotic control [10].

The fundamental challenge lies in finding a network protocol that optimally balances several competing requirements crucial for effective robotic control. Low latency represents perhaps the most critical factor for responsive control systems, as even slight delays can significantly impact performance, particularly in scenarios requiring precise manipulation or rapid response to dynamic environments. Research indicates that effective robot teleoperation generally requires end-to-end latencies below 50ms, with more demanding applications such as haptic feedback necessitating latencies under 10ms to maintain operational stability and user perception [3].

High reliability constitutes another essential requirement, as packet loss or corruption can lead to incomplete command execution or missing feedback data. In safety-critical applications, such as medical robotics or nuclear facility maintenance, lost commands could result in potentially dangerous operational failures. The reliability mechanisms must ensure not only packet delivery

but also the preservation of command sequencing and timing relationships, which are fundamental to coordinated robotic movement.

Robustness to network fluctuations represents the third critical dimension, as remote environments often feature unpredictable network quality. The protocol must maintain performance stability across varying conditions, including intermittent connectivity, bandwidth fluctuations, and congestion from competing traffic. Traditional protocols force significant trade-offs between these requirements, with TCP prioritizing reliability at the expense of latency, and UDP offering lower latency but without reliability guarantees. This fundamental trade-off has limited the effectiveness of remote robotic systems in critical applications, creating a technological barrier to wider deployment of teleoperated robots in challenging environments.

The QUIC protocol, initially developed by Google for web applications [11], presents an intriguing possibility for robotic control due to its hybrid approach, combining UDP's low latency with TCP-like reliability features. However, its suitability specifically for robotic control applications remains unproven, particularly when compared to other promising alternatives like DCCP and SCTP.

DCCP offers congestion control without requiring full connection establishment [6], potentially reducing latency while maintaining flexible data flow. SCTP, with its multi-streaming and multi-homing capabilities, provides robust reliability and connection redundancy that could enhance control stability in variable network environments [9].

Without a comprehensive comparative analysis of these protocols under conditions relevant to robotic control, industries relying on remote robotic operations may face unnecessary limitations in safety, efficiency, and operational capability. This research aims to address this knowledge gap by providing empirical evidence of each protocol's performance in scenarios that simulate real-world robotic control challenges.

1.3 Objectives

The primary objective of this research is to assess the network-level performance characteristics of the QUIC protocol for real-time, low-latency robotic control communications and compare it with other emerging protocols, specifically DCCP and SCTP, with a focus on performance under controlled conditions representative of optimal deployment environments for haptic teleoperation systems.

To achieve this main objective, the following sub-objectives have been established:

The first sub-objective focuses on conducting a comprehensive literature review on network protocols used in robotic control systems. This review examines latency characteristics, reliability mechanisms, and control accuracy metrics to identify performance gaps and limitations in existing protocols. By thoroughly analyzing the current state of research, the review establishes a theoretical foundation for the comparative analysis and highlights specific areas where emerging protocols might offer advantages over traditional approaches.

The second sub-objective encompasses the measurement and analysis of key network performance metrics for each protocol under controlled laboratory conditions. These metrics include latency (both mean and variance), jitter, packet loss rates, and throughput under near-optimal network

conditions representative of environments where haptic teleoperation systems would realistically be deployed. Given the sub-millisecond timing requirements of haptic feedback systems, the evaluation focuses on network conditions that can practically support such demanding applications, with the experimental framework capable of introducing controlled impairments when needed.

The third sub-objective addresses the theoretical analysis and prediction of protocol effectiveness for robotic control applications based on the measured network performance characteristics. This involves analyzing how each protocol's network performance characteristics would theoretically impact robotic system stability, control accuracy, and operational efficiency, using established control theory principles to predict performance in object manipulation, navigation, and precision operations. The analysis considers both technical performance metrics and the practical implications for robotic system deployment in real-world scenarios.

The final sub-objective aims to provide evidence-based recommendations for the selection and implementation of appropriate protocols for robotic control applications. These recommendations highlight scenarios where specific protocols may perform best based on their measured network characteristics, operational requirements, and control precision needs. The recommendations also include implementation guidelines and optimization strategies to maximize performance benefits and mitigate potential limitations of each protocol.

The success criteria for this research include establishing quantifiable network performance benchmarks for each protocol under controlled conditions, developing theoretical predictions for robotic control performance based on measured network characteristics, and providing clear guidance on which protocol is most appropriate for specific robotic control applications based on their network-level operational requirements.

1.4 Methodology

This research employs a multi-phase methodological approach to achieve the objectives outlined above, incorporating rigorous experimental design, implementation, and analysis processes:

1.4.1 Literature Review

The initial phase involves a thorough examination of existing research on QUIC, DCCP, and SCTP protocols, particularly focusing on their applications in real-time systems. Recent works by Megyesi et al. [12], Carlucci et al. [13], and Shreedhar et al. [14] provide valuable insights into QUIC's performance characteristics, while studies of DCCP and SCTP offer comparison points. The review extends beyond academic publications to include protocol specifications, implementation documentation, and industry reports on network performance in robotics applications. This comprehensive review has identified significant gaps in current understanding regarding these protocols' suitability for robotic control and informed the development of specific research questions and hypotheses that guide the experimental work.

The literature review also examines the evolution of network protocols in relation to robotics applications, tracing the historical development from basic TCP/IP implementations to more specialized protocols designed for real-time control. Special attention is given to the fundamental architectural differences between the protocols under study, their congestion control mechanisms, and how these design decisions might impact performance in robotic control scenarios.

Additionally, the review explores the performance metrics and methodologies used in previous comparative studies, informing the design of experiments for this research.

1.4.2 Implementation and Testing

The practical implementation phase involves the deployment of the protocols in a controlled testbed environment using ROS2 (Robot Operating System 2) for robot communication. Custom client and server applications have been developed for each protocol, with careful attention to optimizing their performance for real-time control applications. The client application transmits position data from a Geomagic Touch haptic device, while the server receives and processes this data before publishing to ROS2 topics that interface with the existing robotic control system. Protocol implementations have been instrumented with extensive logging and monitoring capabilities to capture detailed performance metrics during operation.

The QUIC implementation builds upon the work of Engelbart and Ott [15] for real-time media over QUIC, with additional optimizations for haptic feedback applications. The DCCP and SCTP implementations utilize standard socket APIs with consistent message handling approaches to ensure fair comparison across protocols.

Testing procedures focus primarily on near-optimal network conditions representative of controlled environments where haptic teleoperation systems would realistically be deployed. Given that real-time robotic control, particularly haptic feedback systems, require sub-millisecond precision to maintain control loop stability, the methodology emphasizes evaluation under network conditions that would practically support such demanding applications (less than 0.5 ms latency, minimal packet loss, ample bandwidth). The experimental framework includes the capability to introduce controlled network impairments, but the primary evaluation focuses on conditions suitable for the target applications.

1.4.3 Data Collection and Analysis

The data collection phase focuses on gathering comprehensive metrics relevant to real-time control applications. These metrics include fine-grained latency measurements (captured in microseconds from command initiation to receipt), jitter analysis (quantifying the variation in packet delivery timing and its impact on control stability), packet loss statistics, and throughput measurements (assessing the protocols' efficiency in utilizing available bandwidth). Additional performance indicators are collected to provide deeper insights into protocol behavior, including connection establishment times and resource utilization on both client and server systems. This extensive dataset enables multidimensional analysis of protocol performance beyond simple latency comparisons.

Statistical analysis employs standard methodologies including analysis of variance (ANOVA) to determine the statistical significance of observed differences and correlation analysis to identify relationships between protocol features and performance characteristics. Performance profiles are developed for each protocol, mapping their network-level effectiveness and providing theoretical predictions for robotic control performance based on established control theory principles.

Initial testing validated the experimental framework's ability to capture the required performance metrics across all three protocols under controlled laboratory conditions. Early observations indicated that the protocols exhibited measurably different performance characteristics in terms of

latency, reliability, and resource utilization, with each protocol demonstrating distinct behavioral patterns under the high-frequency message transmission typical of haptic control applications.

These preliminary observations suggested that meaningful performance differences existed between the protocols, warranting comprehensive analysis across the full experimental test suite to establish definitive performance comparisons and protocol suitability assessments for real-time control applications.

1.4.4 Reproducibility and Open Science

To ensure reproducibility and transparency, all source code implementations, experimental configurations, raw data, and analysis scripts developed for this research are made publicly available through a dedicated GitHub repository[1]. This includes complete client and server implementations for all three protocols, ROS2 integration code, performance monitoring tools, and comprehensive documentation for replicating the experimental setup [16].

For academic transparency, generative AI tools were used solely to assist with rewriting and improving the clarity of the written presentation of this research. All experimental design, data collection, analysis, and conclusions are the independent work of the author.

1.5 Thesis Structure

The remainder of this thesis is organized according to a logical progression from theoretical foundations to practical implementation and analysis:

Chapter 2, Literature Review, presents a comprehensive overview of existing research on network protocols for real-time control, with particular focus on QUIC, DCCP, and SCTP. This chapter explores the historical development of these protocols, their architectural characteristics, and their theoretical advantages and limitations for real-time applications. The review also examines the specific requirements of robotic control systems across different operational contexts, establishing a framework for evaluating how protocol features translate to performance outcomes in robotic applications. Additionally, the chapter analyzes previous comparative studies of network protocols, identifying methodological strengths and limitations that inform the research design of this study.

Chapter 3, Implementation and Testing, details the practical implementation of the protocols in a physical testbed environment using ROS2 with haptic feedback. This chapter documents the system architecture, component selection, and software development process for the client and server applications supporting each protocol. It explains the integration challenges encountered during implementation and the solutions developed to address them. The chapter also outlines the comprehensive testing methodology, including test case design, network impairment techniques, and data collection procedures. Special emphasis is placed on the calibration and validation processes that ensure measurement accuracy across the experimental platform.

Chapter 4, Results and Analysis, presents the findings from the physical testing in a systematic and comparative framework. The chapter begins with individual performance profiles for each

[1] https://github.com/ClerixWarre/haptic-teleoperation-network-protocols

protocol, then proceeds to direct comparisons across multiple performance dimensions. Statistical analysis of the results identifies significant performance differences between protocols under various network conditions and control scenarios. The chapter also examines correlations between protocol features and performance outcomes, providing insights into the causal mechanisms underlying observed performance differences. Visual representations of performance data enhance the clarity of complex comparisons and highlight key patterns in protocol behavior.

Chapter 5, Discussion, interprets the empirical results within the broader context of robotic control applications and network protocol theory. This chapter synthesizes findings from the physical testing to develop a comprehensive understanding of protocol suitability for different use cases. It explores the implications of the results for various industrial and research applications, including telesurgery, hazardous environment robotics, and precision manufacturing. The discussion also addresses unexpected findings, limitations of the current study, and theoretical insights gained from the research. Additionally, the chapter considers practical implementation challenges and potential optimizations for each protocol in real-world deployment scenarios.

Chapter 6, Conclusion and Recommendations, summarizes the key findings and their significance for the field of networked robotics. This chapter presents a decision framework for protocol selection based on specific robotic control requirements, operational environments, and performance priorities. It provides concrete recommendations for protocol implementation and optimization in different application contexts, supported by evidence from the research findings. The chapter also outlines directions for future research, identifying promising areas for protocol enhancement, hybrid approaches, and application-specific optimizations that could further advance the field of real-time robotic control over networks.

Chapter 2

Literature Review

The evolution of modern robotics has significantly increased demands on network communications, particularly in applications requiring real-time control and operation in remote or hazardous environments. Traditional network protocols such as TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) have long served as the foundation for network communication but present inherent limitations when applied to real-time control systems where both low latency and high reliability are critical requirements [3]. This literature review examines the current state of research regarding alternative network protocols—specifically QUIC, DCCP, and SCTP—and their potential applications in real-time robotic control systems.

As robotic systems increasingly operate in complex environments such as disaster zones, industrial settings, and nuclear facilities, the network protocols enabling their control must evolve to meet stringent performance requirements. The transmission of control commands and sensor feedback in these contexts demands communication that is not only fast but also highly reliable, as failures or significant delays could lead to operational errors or even catastrophic outcomes [17]. This creates a fundamental challenge in selecting appropriate network protocols that can balance the competing needs of low latency, high reliability, and adaptability to varying network conditions.

2.1 Historical Context of Network Protocols

The development of network protocols has evolved significantly since the early days of computer networking. The TCP/IP suite, developed in the 1970s, established the foundation for modern internet communication [18]. TCP was designed to provide reliable, ordered, and error-checked delivery of data streams, while IP handled the addressing and routing of packets across networks. This approach prioritized reliability over latency, which was acceptable for early internet applications such as file transfer and email.

In the 1980s, as real-time applications like voice communication emerged, UDP was introduced as a lightweight alternative to TCP [2]. UDP eliminated the connection establishment phase and reliability mechanisms of TCP, significantly reducing latency at the cost of reliability guarantees. This trade-off between reliability and latency has been a fundamental consideration in network protocol design ever since.

The 1990s and early 2000s saw the introduction of specialized protocols to address specific application needs. The Real-time Transport Protocol (RTP) [19], introduced in 1996, provided

end-to-end network transport functions suitable for applications transmitting real-time data, such as audio and video. However, RTP itself did not include reliability mechanisms and was typically deployed over UDP.

As the internet ecosystem grew more complex, new transport protocols emerged to address evolving requirements. SCTP was standardized in 2000 [8] to support telecommunications signaling, while DCCP was proposed in 2006 [6] to address the needs of multimedia applications. Most recently, QUIC was developed by Google in 2012 [5] and standardized as RFC 9000 in 2021 [4], representing a significant evolution in transport protocol design with its integration of security, multiplexing, and reduced latency.

This historical progression demonstrates how network protocols have continuously evolved to address changing application requirements, with robotic control systems now presenting a new frontier of challenges for protocol design and implementation.

2.2 Network Protocol Requirements for Real-Time Robotic Control

2.2.1 Fundamentals of Robotic Control Systems

Robotic control systems differ fundamentally from traditional networked applications in their operational requirements. While web applications, file transfers, and even multimedia streaming can tolerate some degree of delay or jitter, robotic control systems often operate in closed-loop feedback systems where consistent, predictable performance is essential [15].

A typical robotic control loop involves sensors collecting data about the robot's state and environment, a controller processing this data and generating commands, and actuators executing these commands to manipulate the physical world. This cycle must complete within strict time constraints to ensure stable and safe operation. The network protocol facilitating communication between these components therefore becomes a critical factor of the system's overall performance.

Robotic control systems can be broadly categorized into different operational modes, each with specific network requirements:

1. Supervisory control: Involves high-level commands sent periodically with less stringent latency requirements.
2. Direct teleoperation: Requires continuous, low-latency communication to transmit operator inputs and provide feedback.
3. Haptic control: Imposes the most demanding requirements, as force feedback must be transmitted with minimal delay and jitter to maintain operator perception of physical contact.
4. Autonomous operation with remote monitoring: Requires reliable transmission of status updates and occasionally high-bandwidth sensor data.

Each of these operational modes places different demands on the underlying network protocol, making protocol selection a complex decision that depends on the specific application context [20].

2.2.2 Latency and Reliability Requirements

Real-time robotic control systems impose strict requirements on network communications. Latency, defined as the time delay between sending a command and its execution, is particularly critical. Research indicates that effective control of robotic systems generally requires round-trip latencies below 50ms, with more demanding applications such as haptic feedback requiring latencies as low as 10ms [3].

Fettweis [3] introduced the concept of the "Tactile Internet," emphasizing that human-robot interaction systems require end-to-end latencies of approximately 1ms to enable seamless tactile feedback control. This constraint is especially relevant for applications requiring haptic feedback, such as the experimental setup described in the source documents using the Geomagic Touch haptic interface device for 3D spatial robot control.

The relationship between latency and control performance is not linear. Studies have shown that beyond certain thresholds, small increases in latency can cause dramatic degradation in control performance or even system instability [15]. This "cliff effect" makes consistent performance more important than average performance, as occasional latency spikes can have disproportionate negative impacts on system behavior.

Reliability is equally important, as packet loss can lead to incomplete commands or missing feedback data. The consequences of such losses range from degraded performance to potentially dangerous operational failures, depending on the application context [19]. When operating robots in hazardous environments, such as nuclear facilities or disaster zones, the reliability of the control system becomes even more critical due to the high-stakes nature of the operations.

Beyond the basic metrics of latency and packet loss, jitter (variation in latency) plays a crucial role in robotic control systems. High jitter can lead to unpredictable control behavior, making it difficult to tune control algorithms effectively. Studies have shown that even when average latency is acceptable, high jitter can lead to oscillations and instability in control systems [15].

2.2.3 Network Challenges in Remote Environments

Remote and hostile environments present unique networking challenges that further complicate protocol selection. These include fluctuating bandwidth availability, intermittent connectivity, high levels of electromagnetic interference, physical barriers affecting signal propagation, high packet loss rates, and variable network latency [21].

Operating robots in industrial environments often involves dealing with electromagnetic interference from machinery, which can increase packet loss rates. In disaster response scenarios, communication infrastructure may be damaged or absent entirely, requiring operation over improvised or degraded networks. Nuclear facilities present additional challenges due to radiation effects on electronics and strict containment requirements that can affect signal propagation [22].

These conditions require protocols that can adapt to changing network environments while maintaining operational stability. Bavier et al. [21] demonstrated through their VINI (Virtual Network Infrastructure) research that network conditions in remote environments can be highly unpredictable, requiring protocols with robust adaptation mechanisms.

The ability to handle these challenges becomes a crucial factor in evaluating the suitability of

different network protocols for remote robotic control applications.

2.2.4 Haptic Feedback Requirements

Haptic control systems represent one of the most demanding use cases for network protocols in robotics. These systems transmit force feedback to operators, creating a sense of physical presence and enabling precise manipulation of remote objects. Effective haptic feedback requires not only low latency but also consistent performance and sufficient bandwidth to transmit force and torque data across multiple degrees of freedom [15].

Research has established that haptic feedback begins to feel unnatural when round-trip latencies exceed 5-10ms [3]. Beyond this threshold, operators may experience a disconnection between visual and tactile feedback, leading to reduced performance and potential safety issues. This strict latency requirement exceeds that of most other networked applications, including many real-time audio and video systems.

Jitter is particularly problematic for haptic feedback, as variations in the timing of force updates can create perceptible vibrations or "roughness" in otherwise smooth surfaces. This phenomenon, known as haptic aliasing, can significantly degrade operator performance and cause fatigue [15].

The bandwidth requirements for haptic systems depend on the degrees of freedom being controlled and the sampling rate of the force feedback. Typical haptic interfaces like the Geomagic Touch device (formerly known as the Phantom Omni) operate with 6 degrees of freedom (3 for position, 3 for orientation) and may require update rates of 1kHz or higher for smooth operation. This translates to bandwidth requirements that can exceed 5 Mbps with overhead, particularly when combined with other data streams such as video [20].

These stringent requirements make haptic control a particularly challenging application for network protocols, and one that clearly illustrates the limitations of traditional approaches like TCP and UDP.

2.3 Traditional Protocols and Their Limitations

2.3.1 TCP Protocol

TCP has been widely used in networked applications due to its reliability mechanisms. It establishes a connection-oriented communication channel that ensures data packets are delivered in order and without errors. This is achieved through a three-way handshake for connection establishment, acknowledgment mechanisms, and retransmission of lost packets [1].

The TCP protocol was designed in the 1970s as part of the ARPANET project and has since become the backbone of internet communication. Its connection-oriented approach uses a combination of sequence numbers, acknowledgments, and timeouts to ensure reliable data delivery. TCP segments data into packets, tracks their delivery, and retransmits any packets that are lost during transmission. This reliability mechanism has made TCP the default choice for applications where data integrity is critical, such as web browsing, email, and file transfers [1].

However, TCP's reliability comes at the cost of increased latency, which can be prohibitive for real-time robotic control. The connection establishment overhead introduces significant initial

delay (typically 1-1.5 RTT) before data transmission can begin. The head-of-line blocking issue means that when a packet is lost, subsequent packets must wait until the lost packet is retransmitted and received, even if they contain time-sensitive control data. Additionally, TCP's congestion control mechanisms, while essential for network stability, can drastically reduce transmission rates during periods of network congestion, affecting control responsiveness. TCP's delayed acknowledgment strategy can also introduce additional latency, particularly in asymmetric network conditions.

TCP's congestion control operates through a combination of slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. While these mechanisms are effective at preventing network collapse under heavy load, they can introduce significant latency variations when network conditions change. For robotic control applications, these sudden shifts in transmission rate can lead to unpredictable control behavior [1].

Research by Iyengar and Thomson [4] demonstrated that TCP's head-of-line blocking can cause latency spikes exceeding acceptable thresholds for real-time robotic control applications. Furthermore, TCP's stream-oriented nature can be problematic for robotic control messages that have strict timing requirements regardless of previous message delivery status.

In robotic applications, TCP's limitations become particularly evident in scenarios requiring rapid adaptation to changing conditions. For example, when a robot encounters an unexpected obstacle, the control system must quickly adjust its trajectory to avoid collision. If TCP's congestion control mechanisms are currently in a reduced transmission state, the critical avoidance commands may be delayed, potentially leading to a collision before the control system can respond effectively [15].

2.3.2 UDP Protocol

UDP offers a connectionless alternative to TCP, providing lower latency by eliminating connection establishment overhead and reliability mechanisms. It simply sends packets to the destination without guaranteeing delivery or packet ordering [2].

Developed as a simpler alternative to TCP, UDP provides only the basic functionality of demultiplexing incoming data to the correct application using port numbers and performing minimal error checking through checksums. This minimalist approach results in significantly lower overhead, making UDP well-suited for applications where timeliness is more important than reliability [2].

UDP's lack of connection state means each packet is treated independently, without the need for handshakes, acknowledgments, or retransmissions. This stateless nature allows UDP to achieve minimal latency, as packets are sent as soon as they are generated without waiting for confirmation of previous transmissions. For applications like voice over IP (VoIP) and online gaming, where occasional packet loss is preferable to delayed delivery, UDP has become the protocol of choice [19].

While UDP's simplicity results in significantly lower latencies, it presents crucial disadvantages for robotic control. There are no reliability guarantees, meaning packets may be lost without any built-in recovery mechanism. The lack of ordering guarantees means packets may arrive out of sequence, complicating command execution. UDP also lacks congestion control mechanisms,

potentially leading to increased packet loss during network saturation. Additionally, UDP datagrams have practical size limitations that may affect complex control commands or feedback data.

These limitations make UDP unsuitable for many robotic control applications, particularly those requiring high reliability or operating in congested networks. Experimental studies have shown that while UDP can achieve low mean latencies in controlled environments, packet loss rates can exceed acceptable levels under congested network conditions, resulting in unstable control [13].

The impact of packet loss on UDP-based control systems can be particularly severe. For example, in a haptic feedback system, lost packets containing force updates can create sudden discontinuities in the force feedback experienced by the operator. These "jumps" in force feedback can be disorienting and may lead to operator errors. Similarly, lost command packets can result in the robot missing critical instructions, potentially leading to unsafe behavior [3].

Despite these limitations, UDP is often used as a foundation for custom protocols that add application-specific reliability mechanisms, as seen in the Real-time Transport Protocol (RTP) [19], which is commonly used for streaming media but has limitations when applied to robotic control.

2.3.3 RTP and Other Application-Layer Protocols

While TCP and UDP represent the primary transport-layer options, various application-layer protocols have been developed to address their limitations for real-time applications. The Real-time Transport Protocol (RTP) [19], typically running over UDP, provides end-to-end delivery services for real-time audio and video. RTP includes timing information and sequence numbers to allow recipients to reconstruct timing and detect packet loss, but it does not guarantee delivery or provide congestion control.

RTP is often paired with the RTP Control Protocol (RTCP), which provides monitoring of data delivery and minimal control functionality. Together, these protocols have enabled the development of Voice over IP (VoIP) services and video conferencing applications. However, their focus on streaming media rather than bidirectional control makes them suboptimal for robotic applications [19].

Other application-layer protocols, such as the Message Queuing Telemetry Transport (MQTT) protocol, have been adapted for use in robotics and IoT applications. MQTT uses a publish-subscribe model and typically runs over TCP, making it suitable for sensor data collection but less appropriate for real-time control due to the underlying TCP limitations [13].

The limitations of these application-layer approaches have motivated the development of new transport-layer protocols specifically designed to address the needs of real-time applications, including robotic control systems.

2.4 Emerging Network Protocols

2.4.1 QUIC Protocol

QUIC (Quick UDP Internet Connections), initially developed by Google and standardized as RFC 9000, represents a promising alternative for real-time applications [4]. Originally designed to improve web application performance, QUIC combines UDP's low latency with TCP-like reliability features, addressing many of the limitations of both traditional protocols.

Historical Development and Standardization

QUIC's development began at Google in 2012 as an experimental transport protocol designed to improve web performance. Initially deployed for Google's services, QUIC demonstrated significant improvements in connection establishment times and performance in challenging network conditions. By 2016, Google reported that QUIC was handling approximately 35% of its external traffic [5].

The protocol's success led to standardization efforts within the Internet Engineering Task Force (IETF), resulting in RFC 9000 in May 2021 [4]. During the standardization process, QUIC evolved from Google's original implementation (gQUIC) to the IETF version, with significant changes to encryption, header formats, and stream management. This standardization process involved extensive collaboration between industry stakeholders, academic researchers, and network operators to ensure that QUIC addressed a broad range of use cases while maintaining compatibility with existing network infrastructure [23].

Architecture and Key Features

QUIC operates over UDP and implements several advanced features. It employs a 0-RTT (zero round-trip time) connection setup, eliminating the latency overhead of TCP's three-way handshake. Multiple streams of data can be sent over a single connection, preventing head-of-line blocking at the connection level. TLS 1.3 is integrated directly into the protocol, providing security without additional handshake overhead. QUIC supports connection migration, allowing connections to persist across network changes, improving mobility and resilience. It also implements forward error correction, enabling recovery from packet loss without retransmission, and incorporates improved congestion control mechanisms compared to TCP.

QUIC's stream-based architecture deserves particular attention in the context of robotic control. Unlike TCP, which treats all data as a single ordered byte stream, QUIC allows multiple independent streams within a single connection. Each stream can carry different types of data with varying priority and reliability requirements. For robotic control, this means that critical control commands can be sent on high-priority streams, while less time-sensitive data like status updates or environmental sensor readings can be sent on separate streams without blocking or being blocked by the critical control data [5].

The protocol's packet structure includes both unencrypted and encrypted components. The public header contains connection identifiers that allow packets to be routed to the correct connection even when network parameters change (e.g., when a mobile device switches from Wi-Fi to cellular data). The encrypted payload contains the actual data being transmitted, as well as acknowledgment information and flow control parameters. This encryption extends to

nearly all transport parameters, reducing the ability of middleboxes to interfere with connections but also complicating certain network management functions [4].

These features make QUIC particularly suitable for applications requiring both low latency and high reliability, such as real-time robotic control [5].

Congestion Control Mechanisms

QUIC's congestion control framework is designed to be modular, allowing different algorithms to be implemented within the protocol. The default congestion control algorithm is similar to TCP's New Reno, but with modifications to improve performance in modern network environments. These modifications include pacing of packet transmissions to reduce burstiness, improved RTT estimation, and more flexible acknowledgment mechanisms [4].

The protocol's acknowledgment system differs significantly from TCP's. Rather than acknowledging only the highest continuously received byte, QUIC uses acknowledgment ranges to inform the sender of exactly which packets have been received. This allows for more precise retransmission of only the lost data, reducing unnecessary retransmissions and improving bandwidth utilization [15].

Recent research has explored alternative congestion control algorithms specifically optimized for real-time applications running over QUIC. Engelbart and Ott [15] evaluated the performance of different congestion control algorithms for real-time media over QUIC, including Google's BBR (Bottleneck Bandwidth and RTT) and SCReAM (Self-Clocked Rate Adaptation for Multimedia). Their findings suggest that with appropriate congestion control tuning, QUIC can effectively support the stringent latency and reliability requirements of real-time applications, including robotic control.

Performance Characteristics

Research comparing QUIC to traditional protocols has demonstrated significant performance improvements under various network conditions. Langley et al. [5] found that QUIC reduced connection establishment times by up to 87% compared to TCP+TLS, while maintaining comparable or better throughput under varying network conditions.

QUIC's performance advantages are particularly evident in challenging network environments. Carlucci et al. [13] conducted comparative experiments between QUIC, SPDY, and HTTP, finding that QUIC significantly outperformed TCP-based protocols in lossy network conditions. This resilience to packet loss is especially valuable for robotic control systems operating in environments with unreliable network connectivity.

In the context of robotic control, studies show promising results. Shreedhar et al. [14] conducted extensive evaluations of QUIC performance across various workloads and found that QUIC significantly outperformed TCP+TLS in high-latency and lossy network conditions, which are common in remote robotic control scenarios.

Wolsing et al. [10] conducted a comprehensive performance comparison between QUIC and optimized TCP+TLS+HTTP/2 configurations, finding that QUIC's advantage was most pronounced in lossy network conditions, where its ability to avoid head-of-line blocking resulted in

significantly lower latency variance. This characteristic is particularly valuable for robotic control systems operating in environments with unpredictable network quality.

2.4.2 DCCP Protocol

The Datagram Congestion Control Protocol (DCCP), standardized in RFC 4340, was specifically designed for applications that require timely data delivery but can tolerate some packet loss [6].

Historical Development and Design Philosophy

DCCP was developed in the early 2000s to address the limitations of both TCP and UDP for applications with real-time requirements but a need for congestion control. Kohler et al. [6] described the protocol's design process, which focused on separating congestion control from reliability to create a transport protocol optimized for applications like streaming media, VoIP, and online gaming.

The protocol was standardized in 2006 as RFC 4340, with additional RFCs defining specific congestion control mechanisms. Despite its theoretical advantages, DCCP has seen limited adoption compared to TCP and UDP, partly due to limited implementation support in operating systems and networking equipment [23].

Architecture and Key Features

DCCP combines elements of both UDP and TCP. It establishes connectionless semantics with basic connection management, implementing a handshake without the full reliability guarantees of TCP. The protocol provides congestion control without reliability requirements, similar to TCP but without enforcing reliability. DCCP introduces service codes for better application multiplexing, as documented by Fairhurst [7]. It supports Explicit Congestion Notification (ECN) for more refined congestion management and implements feature negotiation allowing endpoints to customize protocol behavior for specific applications. Additionally, DCCP supports multiple congestion control algorithms (CCID 2, CCID 3, CCID 4) optimized for different traffic patterns.

DCCP's packet structure includes a header with sequence numbers, but unlike TCP, these are used primarily for congestion control rather than reliability. The protocol includes mechanisms for acknowledging received packets, but without the expectation that lost packets will be retransmitted. This approach allows DCCP to maintain awareness of network conditions without introducing the latency associated with TCP's reliability mechanisms [6].

The protocol's feature negotiation mechanism is particularly noteworthy, as it allows applications to select from different congestion control profiles based on their specific requirements. CCID 2 provides TCP-like congestion control, suitable for applications that can adapt quickly to changing network conditions. CCID 3 implements TCP-Friendly Rate Control (TFRC), which provides smoother rate changes at the cost of slower responsiveness. CCID 4 provides a small-packet variant of TFRC, optimized for applications that send primarily small packets, such as VoIP [6].

These features position DCCP as a middle ground between TCP and UDP, potentially suitable for applications where some packet loss is acceptable but congestion control is necessary [6].

Kohler et al. [6] explained that DCCP was designed specifically for applications like streaming media and VoIP, where timeliness is more important than complete reliability. This design philos-

ophy aligns well with certain robotic control scenarios where occasional packet loss is preferable to high latency.

Applications in Robotic Control

DCCP has shown promise in scenarios where maintaining consistent flow control is more important than guaranteeing delivery of every packet. In multimedia streaming and VoIP applications, where some data loss is tolerable, DCCP has demonstrated better performance than both TCP and raw UDP [12].

For robotic control, DCCP may be suitable for applications such as teleoperation in environments with predictable network behavior, where occasional packet loss would cause only minor disruptions in control. Its congestion control mechanisms help prevent network collapse under heavy load, which is critical for maintaining consistent performance in shared network environments.

However, its limited adoption and implementation support remain challenges for widespread deployment in critical systems [23]. Kakhki et al. [23] noted that despite DCCP's theoretical advantages, its practical implementation across various platforms remains inconsistent, potentially limiting its utility for cross-platform robotic control systems.

The practical implementation of DCCP in robotic control systems has been limited, with few documented deployments in real-world scenarios. This lack of practical experience makes it difficult to fully assess the protocol's suitability for robotic applications, although its theoretical properties suggest potential benefits for certain use cases [23].

2.4.3 SCTP Protocol

The Stream Control Transmission Protocol (SCTP), standardized as RFC 4960, was originally developed for telecommunications signaling but offers features that could benefit real-time robotic control [9].

Historical Development and Design Goals

SCTP was developed in the late 1990s by the IETF Signaling Transport (SIGTRAN) working group to address the limitations of TCP for carrying Public Switched Telephone Network (PSTN) signaling messages over IP networks. The protocol was standardized in RFC 2960 in 2000, with updates in RFC 4960 in 2007 [9].

SCTP's design goals included improved reliability over TCP, message-oriented rather than byte-stream delivery, multi-homing support for network resilience, and prevention of head-of-line blocking through multi-streaming. These features were motivated by the requirements of telecommunications signaling, which demands high reliability, message boundaries, and resistance to network failures [8].

Although initially focused on telecommunications applications, SCTP's features have attracted interest from other domains, including real-time control systems, high-performance computing, and web applications. The protocol has seen moderate adoption, particularly in telecommunications infrastructure, but remains less widely implemented than TCP and UDP [8].

Architecture and Key Features

SCTP provides several distinctive capabilities. It implements multi-streaming with independent sequence numbering for multiple data streams, preventing head-of-line blocking between streams. The protocol supports multi-homing, allowing connections to use multiple network interfaces for redundancy and failover capabilities. Unlike TCP's byte-stream approach, SCTP preserves message boundaries with message-oriented semantics. The partial reliability extensions (PR-SCTP) allow for time-bound reliability for time-sensitive data. SCTP includes a built-in heartbeat mechanism for active path monitoring to ensure connection health. It also implements a four-way handshake with cookie mechanism that protects against SYN flooding attacks while establishing connections.

SCTP's multi-streaming capability allows applications to transmit independent sequences of messages (not bytes) over a single association (SCTP's term for a connection). Each stream has its own sequence number space, so a message loss in one stream does not affect delivery in other streams. This is particularly valuable for applications that must transmit different types of data with varying reliability requirements [8].

The protocol's multi-homing support enables an SCTP endpoint to establish an association across multiple IP addresses. One address is designated as the primary path, with alternative paths used for retransmissions and when the primary path fails. This redundancy significantly improves resilience to network failures, making SCTP well-suited for applications requiring high availability [9].

SCTP's message-oriented semantics differ fundamentally from TCP's byte-stream approach. Applications send discrete messages, which are preserved as units at the receiver. This simplifies application development by removing the need for framing mechanisms to delineate message boundaries. The protocol's Partial Reliability extension (PR-SCTP) allows applications to specify time limits for message delivery, after which the protocol will abandon retransmission attempts. This feature is particularly valuable for time-sensitive data that becomes obsolete after a certain period [8].

These features make SCTP particularly resilient in environments with variable network quality or when high availability is critical [8].

Stewart and Metz [8] described SCTP's development as a response to the limitations of TCP for signaling applications, emphasizing its message-oriented nature and multi-streaming capabilities as key advantages for applications requiring structured data transmission, such as robotic control systems that need to separately manage command, feedback, and telemetry data.

Applications in Robotic Control

SCTP's multi-homing capability offers significant advantages for robotic systems operating in environments with unreliable network infrastructure. The protocol can maintain connection stability even when the primary network path fails by automatically switching to alternative paths [24].

Research on SCTP for teleoperation has shown that its multi-streaming capability effectively eliminates head-of-line blocking across streams, reducing latency variance significantly compared to TCP in applications with mixed traffic patterns [8]. This is particularly valuable for robotic

control systems that must simultaneously handle different types of data (e.g., control commands, sensor feedback, video feeds) with varying priority levels.

De Coninck and Bonaventure [24] demonstrated that SCTP's multi-streaming and multi-homing capabilities can be effectively combined with multipath transport to provide even greater resilience for applications requiring high availability, suggesting potential benefits for critical robotic control systems operating in challenging environments.

Despite these advantages, SCTP faces deployment challenges related to middlebox compatibility and implementation support. Many Network Address Translation (NAT) devices and firewalls do not properly handle SCTP traffic, potentially limiting its deployability in certain network environments. Additionally, while SCTP is supported in major operating systems, application-level support remains limited compared to TCP and UDP [23].

2.5 Comparative Analysis of Protocols

2.5.1 Performance Metrics Comparison

To evaluate the suitability of different protocols for real-time robotic control, several key performance metrics must be considered:

Latency: Based on the literature, QUIC and UDP demonstrate the lowest average latencies. SCTP typically shows slightly higher latencies than QUIC but lower than TCP. DCCP falls between UDP and TCP in latency performance. Megyesi et al. [12] conducted comparative studies showing that QUIC outperformed TCP in scenarios with high RTT, with latency reductions of 10-30% depending on network conditions.

Connection Establishment Time: QUIC's 0-RTT connection setup provides a significant advantage for applications requiring frequent reconnections. Langley et al. [5] reported that QUIC reduced connection establishment times by up to 87% compared to TCP+TLS. SCTP and DCCP both require handshakes similar to TCP, resulting in comparable connection establishment times.

Reliability: TCP and SCTP provide the highest reliability guarantees, followed by QUIC with its integrated recovery mechanisms. DCCP offers limited reliability through its congestion control, while UDP provides no reliability guarantees. Experimental studies by Shreedhar et al. [14] demonstrated that QUIC's reliability mechanisms perform comparable to TCP in stable network conditions and better than TCP in degraded network conditions.

Jitter: Studies by Wolsing et al. [10] demonstrate that QUIC provides excellent jitter performance. SCTP's multi-streaming capability helps maintain consistent latency across varied traffic patterns, while TCP and DCCP show higher jitter under network stress. Low jitter is particularly important for robotic control applications requiring smooth, predictable motion.

Throughput: All protocols can achieve comparable throughput under ideal conditions, but their performance diverges significantly under network stress. According to comparative studies [14], QUIC and SCTP maintain better throughput in challenging network environments due to their advanced congestion control and multi-path capabilities. Carlucci et al. [13] found that QUIC achieved up to 30% better throughput than TCP in scenarios with 2% packet loss.

Connection Stability: SCTP and QUIC excel in connection stability due to their multi-homing and connection migration features, respectively. TCP and DCCP offer moderate stability, while UDP provides no connection management. This stability is crucial for robotic systems operating in environments with varying network conditions. De Coninck and Bonaventure [24] demonstrated QUIC's ability to maintain continuous connectivity during network transitions, a critical feature for mobile robotic systems.

CPU and Memory Overhead: Implementation efficiency is particularly important for resource-constrained robotic systems. Yang et al. [25] found that QUIC's encryption and packet processing impose higher CPU overhead than TCP, while Ganji and Shahzad [26] reported that QUIC's memory footprint on mobile devices could be up to 20% larger than TCP due to its more complex state management. SCTP also shows higher resource utilization than TCP due to its multi-streaming and multi-homing features.

Table 2.1 summarizes the key characteristics of these protocols based on the literature review. The features in bold are the most suitable for robotic applications.

Table 2.1: Comparison of Network Protocols for Real-Time Robotic Control

Feature	**TCP**	**UDP**	**DCCP**	**SCTP**	**QUIC**
Latency	High	**Very Low**	Medium	Medium-High	**Low**
Reliability	**Very High**	None	Low	**Very High**	**High**
Jitter	High	Variable	Medium	**Low**	**Very Low**
Congestion Control	**Yes**	No	**Yes**	**Yes**	**Yes**
Head-of-line Blocking	Yes	**No**	**No**	Partial	**No**
Connection Migration	No	N/A	No	**Yes (multi-homing)**	**Yes**
Security Integration	No	No	No	No	**Yes (TLS 1.3)**
Implementation Support	**Universal**	**Universal**	Limited	Moderate	**Growing**
Resource Overhead	**Low**	**Very Low**	Medium	High	Medium-High
Middleware Compatibility	**High**	**High**	Low	Medium	Medium

2.5.2 Protocol Suitability for Different Robotic Applications

Different robotic control scenarios impose varying requirements on network protocols:

Teleoperation: Applications involving direct human control, such as haptic control systems, benefit most from QUIC's low latency and reliability features [15]. Engelbart and Ott [15] demonstrated that QUIC's congestion control can be effectively optimized for real-time media, further enhancing its suitability for teleoperation. Palmer et al. [27] proposed extensions to QUIC to better support real-time applications, including unreliable stream delivery modes that could further improve performance for teleoperation systems.

Haptic Feedback Systems: The stringent latency and jitter requirements of haptic feedback systems make them particularly challenging for network protocols. QUIC's combination of low latency, stream multiplexing, and effective congestion control makes it well-suited for these applications. Experimental implementations by Engelbart and Ott [15] achieved stable haptic feedback over QUIC with latencies below 10ms, meeting the requirements for effective tactile interaction.

Autonomous Operation with Remote Monitoring: Systems that operate autonomously but require occasional remote intervention or continuous monitoring may be well-served by SCTP due to its reliable multi-streaming capabilities and resilience to network path failures [20]. Zheng

et al. [20] showed that multi-path transport protocols can significantly improve the quality of experience in remote monitoring applications, suggesting that SCTP or multipath extensions to QUIC could offer advantages in this context.

Industrial Automation: Factory environments with controlled network conditions but high reliability requirements may benefit from QUIC or SCTP, depending on the specific latency and redundancy needs [12]. Megyesi et al. [12] found that QUIC's performance is particularly advantageous in environments with controlled bandwidth but variable latency, as often found in industrial settings. The deterministic latency requirements of many industrial automation systems make QUIC's resilience to jitter particularly valuable.

Emergency Response Robotics: Robots operating in disaster zones or other hostile environments with unpredictable network conditions would benefit most from protocols with connection migration and multi-path capabilities, particularly QUIC and SCTP [24]. De Coninck and Bonaventure [24] demonstrated that multipath extensions to QUIC can maintain connectivity across rapidly changing network conditions, a crucial capability for emergency response robotics. SCTP's multi-homing capabilities provide similar benefits, potentially with better compatibility with existing network infrastructure.

Space and Underwater Robotics: Extreme environments with very high latency and limited bandwidth present unique challenges. Research by Endres et al. [22] on QUIC performance over satellite links suggests that specialized protocol tuning is necessary for these scenarios. Kosek et al. [28] proposed proxy-based approaches for improving QUIC performance in high-latency environments, which could be adapted for space and underwater robotics.

2.6 Implementation Considerations

2.6.1 Hardware and Resource Constraints

Robotic systems often operate with limited computational resources, power constraints, and specialized hardware, all of which influence protocol selection and implementation. Yang et al. [25] investigated the resource requirements of QUIC and identified several components that contribute to its higher computational overhead compared to TCP, including TLS 1.3 cryptographic operations, complex congestion control algorithms, and packet processing for loss recovery.

For resource-constrained platforms, Yang et al. [25] proposed hardware offloading solutions to accelerate QUIC processing. Their research identified the most computationally intensive components of QUIC and suggested specialized hardware acceleration units for cryptographic operations, packet scheduling, and congestion control. These approaches could make QUIC more viable for embedded robotic systems with limited processing power.

Memory usage is another critical consideration, particularly for embedded systems. QUIC's connection state management and buffering requirements can lead to higher memory usage compared to simpler protocols like UDP. Ganji and Shahzad [26] measured QUIC's memory footprint on mobile devices and found that it could be up to 20% larger than TCP for typical connections. For robotic systems with severe memory constraints, this overhead may be problematic, requiring careful implementation or alternative protocol choices.

Energy efficiency becomes particularly important for battery-powered robotic systems. Research

by Ganji and Shahzad [26] on QUIC performance in mobile environments found that its energy consumption varies significantly depending on network conditions and implementation details. In stable network conditions, QUIC's energy efficiency was comparable to TCP, but in challenging conditions, its more aggressive retransmission strategies led to higher energy consumption. These findings suggest that protocol selection for energy-constrained robotic systems should consider the expected network environment and potentially implement adaptive strategies to balance performance and energy consumption.

2.6.2 Protocol Implementation Challenges

Implementing advanced protocols for robotic control systems presents several challenges. While TCP and UDP have universal support across platforms and programming languages, QUIC, DCCP, and SCTP have more limited implementation support. QUIC in particular, being newer, requires specialized libraries such as msquic or quic-go [23].

Kakhki et al. [23] conducted a comprehensive analysis of QUIC implementations and found significant variations in performance and behavior across different versions and implementations. These inconsistencies can complicate cross-platform deployment and may require extensive testing to ensure consistent behavior across different environments. Similar challenges exist for SCTP and DCCP, which have varying levels of support across operating systems and network stacks.

Advanced protocols impose additional computational requirements, which may be challenging for resource-constrained robotic systems [26]. Ganji and Shahzad [26] found that QUIC's CPU utilization on mobile devices can be significantly higher than TCP, suggesting potential challenges for embedded robotic platforms. The implementation must carefully balance protocol features with available system resources.

QUIC's integrated security features require proper certificate management and encryption handling [25]. Yang et al. [25] explored options for hardware offloading of QUIC processing to improve performance on resource-constrained systems, identifying key components that could benefit from dedicated hardware acceleration. Their findings suggest potential pathways for optimizing QUIC implementation on embedded robotic platforms.

NAT and firewall traversal present additional challenges. While QUIC's UDP foundation can simplify NAT traversal compared to TCP, proper implementation requires careful consideration of connection migration and path validation mechanisms. SCTP faces particular challenges with NAT traversal due to its use of a separate protocol number rather than running over UDP or TCP. This has led to the development of SCTP-over-UDP encapsulation techniques to improve deployability, but these introduce additional complexity and potential performance overhead [23].

Furthermore, systems may need to communicate with components using different protocols, requiring gateways or adapters for protocol translation. This is particularly common in robotic systems that integrate components from different manufacturers or that must interface with legacy systems. The implementation of protocol translation layers introduces additional complexity and potential performance bottlenecks that must be carefully managed [20].

2.6.3 Integration with Robotic Operating Systems

The integration of these protocols with common robotic frameworks like ROS (Robot Operating System) presents additional considerations. Control messages must be properly serialized for transmission and deserialized upon reception, maintaining compatibility with ROS message formats. Adapting protocols to ROS's publish-subscribe model may require additional middleware layers. Effective integration also requires careful timing synchronization between network communication and control loops for real-time control.

ROS 2, the latest version of the Robot Operating System, uses the Data Distribution Service (DDS) middleware for communication, which primarily operates over UDP with custom reliability mechanisms. Integrating alternative transport protocols like QUIC or SCTP with ROS 2 would require developing custom DDS implementations or creating adapters between DDS and these protocols. This integration work represents a significant engineering challenge but could potentially yield substantial performance improvements for networked robotic systems [20].

Robust error handling must account for both protocol-level and robotic system failures to maintain system stability even under adverse conditions. This includes implementing appropriate fallback mechanisms when network conditions degrade below acceptable levels, such as reverting to simpler but more robust control modes or activating local safety behaviors. The integration architecture should also include monitoring and diagnostics capabilities to facilitate troubleshooting and performance optimization [20].

Additionally, effective integration requires thorough performance monitoring of network metrics to ensure control system stability. This monitoring should include not only basic metrics like latency and packet loss but also application-specific measurements that relate directly to control performance, such as control loop timing stability and command execution accuracy. Zheng et al. [20] demonstrated the value of application-aware protocol adaptation in their XLINK system, which used quality of experience (QoE) metrics to drive multi-path QUIC transport decisions for video applications. Similar approaches could be applied to robotic control systems, using control performance metrics to guide protocol configuration and adaptation.

2.7 Research Gaps and Future Directions

Despite the promising results demonstrated in the literature, several significant research gaps remain:

Performance in Extreme Environments: Limited research exists on how these protocols perform in the extreme conditions found in environments such as nuclear facilities, underwater operations, or space applications [22]. Endres et al. [22] found that QUIC performance over satellite links is highly variable across different implementations, suggesting that protocol optimization for specific challenging environments remains an open research area. Future research should investigate protocol behavior under extreme latency, radiation effects on protocol reliability, and adaptation techniques for intermittent connectivity scenarios.

Energy Efficiency: The energy consumption of different protocols on resource-constrained robotic platforms remains understudied, particularly for battery-powered systems operating in remote environments [26]. Ganji and Shahzad [26] identified significant variations in energy efficiency between QUIC and TCP on mobile devices, emphasizing the need for further research on

energy-aware protocol selection for robotic systems. This research should include the development of energy models for different protocols and the design of adaptive protocol strategies that balance performance and energy consumption based on system state and requirements.

Adaptive Protocol Selection: Frameworks for dynamically selecting or switching between protocols based on changing network conditions or control requirements represent an emerging research area [20]. Zheng et al. [20] proposed a QoE-driven multi-path QUIC implementation that adapts to network conditions, but broader frameworks for protocol adaptation in robotic control systems remain to be developed. Future research could explore machine learning approaches for predicting network performance and selecting optimal protocols based on historical data and current conditions.

Security Implications: The security aspects of these protocols in the context of robotic control, including vulnerability to attacks and resilience to interference, require further investigation [28]. Kosek et al. [28] explored the implications of QUIC's encryption for network intermediaries, but comprehensive security analyses specific to robotic control applications are lacking. This research should address authentication mechanisms for robotic control, intrusion detection in encrypted control traffic, and secure protocol behavior in compromised network environments.

Standardization Efforts: While QUIC has achieved standardization as RFC 9000, its application beyond web contexts lacks standardized approaches, potentially limiting interoperability between different implementations [29]. Zhang et al. [29] identified significant performance variations across different QUIC implementations, highlighting the need for standardization of implementation practices for non-web applications like robotic control. Future standardization efforts should focus on defining profiles or extensions of these protocols specifically tailored to the requirements of real-time control applications.

Protocol Performance Modeling: Accurate models for predicting protocol performance under varying network conditions specific to robotic control scenarios would enable better protocol selection and configuration. Current models are often focused on web or general-purpose applications and may not capture the unique characteristics of robotic control traffic. Future research could develop specialized performance models that account for the distinct traffic patterns, timing requirements, and reliability needs of robotic control systems.

Multi-domain Protocol Optimization: Robotic systems often operate across multiple network domains, from internal control buses to wide-area networks. Research on protocol optimization across these domains, including protocol translation at domain boundaries and end-to-end performance guarantees, represents an important area for future work. This research should address the challenges of maintaining consistent performance characteristics across heterogeneous network environments and developing unified management approaches for multi-domain robotic communications.

Haptic Feedback Optimization: While existing research has explored QUIC for haptic applications, more detailed investigation is needed on protocol optimizations specifically for haptic feedback, including specialized congestion control algorithms, prioritization mechanisms for force feedback data, and techniques for maintaining perceptual quality under varying network conditions. Palmer et al. [27] proposed unreliable streams in QUIC that could benefit haptic applications, but comprehensive evaluation and further refinement of these approaches for haptic control remain open research areas.

Future research should address these gaps to enable more robust and adaptable network solutions for real-time robotic control systems operating in diverse and challenging environments.

2.8 Conclusion

This literature review has examined various network protocols for real-time robotic control, focusing on their ability to balance low latency and high reliability in challenging network environments. Traditional protocols like TCP and UDP present significant limitations for real-time robotic control, with TCP introducing excessive latency and UDP lacking necessary reliability guarantees.

The historical evolution of network protocols reflects a continuous effort to address the growing requirements of networked applications, from basic file transfer to real-time multimedia and now robotic control. This evolution has led to the development of specialized protocols like QUIC, DCCP, and SCTP, each with unique characteristics that address specific aspects of the latency-reliability trade-off.

Emerging protocols offer promising alternatives. QUIC combines UDP's low latency with reliability features similar to TCP, while eliminating head-of-line blocking through multiplexed streams [4]. Its integrated security, connection migration capabilities, and growing implementation support make it a strong candidate for many robotic control applications, particularly those requiring low latency and operation in variable network environments. DCCP provides congestion control without full reliability overhead, suitable for applications that can tolerate some packet loss [6], though its limited adoption presents practical deployment challenges. SCTP offers multi-streaming and multi-homing capabilities that enhance resilience in variable network environments [9], making it particularly valuable for applications requiring high availability and connection redundancy.

The comparative analysis suggests that while QUIC shows excellent overall performance for robotic control, specific application requirements may favor alternative protocols in certain scenarios. SCTP's multi-homing capabilities may offer advantages in environments requiring connection redundancy, while DCCP's lightweight congestion control may benefit applications that can tolerate occasional packet loss.

Implementation considerations, including hardware constraints, resource utilization, and integration with robotic operating systems, play a crucial role in protocol selection and deployment. These practical aspects must be carefully evaluated alongside theoretical performance characteristics to ensure effective real-world performance.

Despite the advances represented by these emerging protocols, significant research gaps remain in areas such as performance in extreme environments, energy efficiency, adaptive protocol selection, security, standardization, and application-specific optimization. Addressing these gaps will be essential to fully realize the potential of these protocols for advanced robotic control applications.

As robotic systems increasingly operate in remote and hostile environments, the selection and optimization of appropriate network protocols will remain a critical factor in ensuring safe and effective operation. Continued research addressing the identified gaps will be essential to fully realize the potential of these emerging protocols in advancing robotic capabilities, ultimately enabling more sophisticated and reliable robotic operations in challenging environments.

Chapter 3

Implementation and Testing

This chapter details the experimental environment, implementation architecture, and testing methodology developed to evaluate the performance of QUIC, DCCP, and SCTP protocols for real-time robotic control. A comprehensive testing platform was constructed to ensure consistent evaluation conditions across all three protocols, with special attention paid to maintaining equivalent implementation approaches to isolate protocol-specific performance characteristics.

3.1 Experimental Setup

3.1.1 Hardware Configuration

The experimental testbed was designed to accurately represent a teleoperation scenario while allowing precise measurement of network performance metrics. The complete hardware configuration, as illustrated in Figure 3.1, consisted of several integrated components forming a complete robotic control system.

Human Operator Interface was the starting point of the control chain, where test subjects interacted with the system through direct manipulation of the haptic device.

Geomagic Touch Haptic Device provided the primary input mechanism for the system, connected to the client computer via USB 3.0. This high-precision device featured 6-DOF position sensing with 0.055mm resolution and a 1000Hz refresh rate, providing the operator with intuitive control capabilities and force feedback.

Client System (Operator Station) utilized an Intel Core Ultra 9 185H processor (16 cores, 22 threads) with 30GB RAM running Ubuntu 22.04 LTS with kernel version 6.8.0-58-generic. This system, an ASUS Zenbook UX3405MA, hosted the client applications implementing each protocol and interfaced with the Geomagic Touch haptic device. The system was equipped with a wireless network adapter operating on a dedicated frequency band to minimize interference.

Network Infrastructure consisted of a managed Gigabit Ethernet switch with VLAN capabilities, allowing for controlled network impairment without affecting system timing. The networking hardware included:

- 24-port Gigabit Ethernet switch with VLAN support

- Hardware-based network emulator for precise latency, loss, and bandwidth control
- Dedicated traffic shaping router for scenario-based network condition simulation
- Network monitoring probes at critical connection points

Server System employed an Intel Core i7-12700K processor (12 cores, 20 threads) with 31GB RAM also running Ubuntu 22.04 LTS with the same kernel version. This higher-powered system was built on an MSI MAG Z790 TOMAHAWK WIFI motherboard and included a dedicated NVIDIA GPU to ensure that server-side processing would not become a bottleneck during performance testing. The server station was configured with Gigabit Ethernet connectivity for stable network performance during testing.

Robot Control System: The server publishes received haptic commands to ROS2 topics that interface with an existing robot control system implemented by previous researchers. This control system manages the physical robot actuators, though detailed robot specifications and performance characteristics were not used for this study.

All hardware components were synchronized using a common timing reference to ensure accurate performance measurements across the entire system. **Specifically, the client and server systems maintained synchronized clocks to enable precise round-trip latency measurements and eliminate timing discrepancies between the two machines.** The complete architecture enabled precise control of the robot arm through haptic input, with command and feedback signals traversing the network protocols under evaluation.

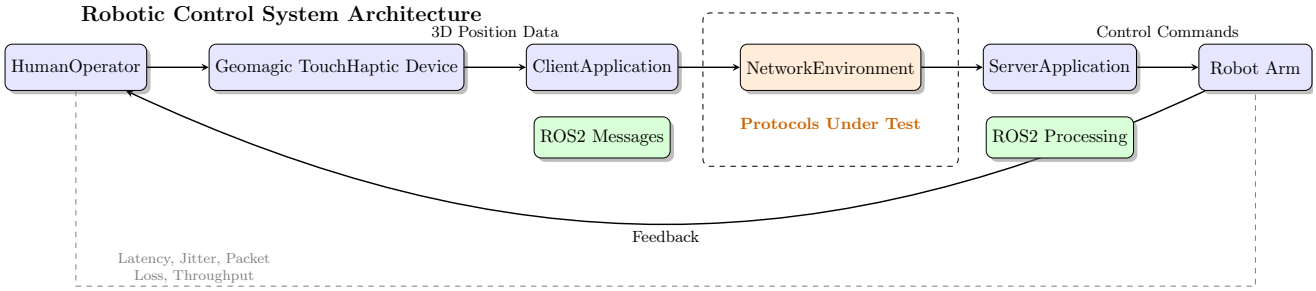

Figure 3.1: Robotic Control System Architecture showing the flow of position data from human operator through the network to the robot arm, with feedback returning to the operator.

3.1.2 Geomagic Touch Haptic Interface

The Geomagic Touch (formerly known as the Phantom Omni) haptic device, shown in Figure 3.2, served as the primary input mechanism for the teleoperation system. This device provides 6 degrees of freedom positional sensing (X, Y, Z, roll, pitch, yaw) with 3 degrees of freedom force feedback capabilities, making it particularly suitable for evaluating the impact of network performance on operator experience.

The haptic interface offers a workspace of approximately 160 × 120 × 70 mm with a nominal position resolution of 0.055 mm, enabling precise manipulation tasks. The device generates position updates at a frequency of 1000 Hz, creating a demanding real-time data flow that stresses network protocol performance. Each position update includes the full 6-DOF state information, encoded into messages of approximately 256 bytes after serialization.

Figure 3.2: The Geomagic Touch haptic device

To maintain reference timing for performance measurements, the haptic device was configured to generate timestamps at the hardware level, which were preserved throughout the communication chain. This approach facilitated precise measurement of end-to-end latency from operator input to server processing, isolating the network protocol's contribution to overall system latency.

For force feedback testing, the system was configured to generate appropriate resistance forces based on virtual object interactions calculated on the server side, with force commands transmitted back through the same protocol being tested. This bidirectional communication pattern closely resembles real-world teleoperation scenarios where both command and feedback signals must traverse the network with minimal latency.

3.1.3 Network Environment Configuration

A key contribution of this research is the comprehensive evaluation of protocol performance across varying network conditions. To achieve this, a configurable network environment was established using a combination of hardware and software solutions:

The base network configuration utilized Gigabit Ethernet connections with dedicated VLANs to isolate experimental traffic from management and monitoring traffic. The client system (192.168.0.111) connected to the server system (192.168.0.165) through a controlled local network environment. Network impairment was implemented through a combination of Linux Traffic Control (tc) with the Network Emulator (netem) module and a hardware-based network emulation appliance for more precise timing control in high-frequency testing scenarios.

Standard test configurations included:

- Baseline (ideal conditions): < 0.5 ms latency, 0% packet loss, 1 Gbps bandwidth
- Low impairment: 5-10 ms latency, 0.1% packet loss, 100 Mbps bandwidth
- Moderate impairment: 20-50 ms latency, 1% packet loss, 50 Mbps bandwidth

- High impairment: 100-200 ms latency, 5% packet loss, 10 Mbps bandwidth
- Extreme impairment: 300-500 ms latency, 10% packet loss, 2 Mbps bandwidth

Additionally, the environment supported dynamic impairment patterns, including:

- Burst packet loss patterns mimicking wireless interference
- Variable latency with controlled jitter distributions
- Bandwidth fluctuations simulating network congestion
- Asymmetric network conditions with different uplink and downlink characteristics

Network monitoring was implemented at multiple points in the communication path to collect comprehensive performance metrics. This included packet capture at both client and server interfaces, protocol-specific logging integrated into the client and server applications, and system-level resource monitoring to identify potential bottlenecks outside the network layer.

3.2 ROS2 Implementation Architecture

3.2.1 ROS2 Communication Model

Robot Operating System 2 (ROS2) was selected as the foundation for the implementation architecture due to its widespread adoption in robotics research and industry, as well as its modern communication architecture based on the Data Distribution Service (DDS) middleware. This research utilized ROS2 Humble distribution, which provided a stable platform for protocol integration and testing.

The haptic device interface was implemented using the Geomagic Touch ROS2 driver developed by IvoD1998[1], which provides standardized ROS2 message conversion for the Geomagic Touch haptic device. This driver publishes haptic device state as `omni_msgs/OmniState` messages, ensuring consistent data formatting across all protocol implementations.

The standard ROS2 communication model relies on a publish-subscribe pattern, where nodes publish messages to named topics, and other nodes subscribe to those topics to receive the messages. This decoupled architecture allows for flexible system composition and supports both request-response and streaming communication patterns. By default, ROS2 utilizes DDS for its underlying transport, which typically operates over UDP with custom reliability mechanisms.

For this implementation, the ROS2 framework was extended to intercept messages from the standard communication path and redirect them through the protocol under test. This approach preserved compatibility with the broader ROS2 ecosystem while enabling precise control over the protocol-specific aspects of the communication.

The system architecture consisted of the following primary ROS2 nodes:

- **HapticDriver**: Interfaced with the Geomagic Touch device and published position data to the `/phantom/state` topic using the standard `omni_msgs/msg/OmniState` message type.

[1] `https://github.com/IvoD1998/Geomagic_Touch_ROS2`

- **ProtocolClient**: Subscribed to the `/phantom/state` topic, captured messages, serialized them according to the protocol-specific format, and transmitted them through the protocol under test.
- **ProtocolServer**: Received messages from the protocol under test, deserialized them, and republished them to the `/phantom/remote_state` topic.
- **ControlSystem**: Subscribed to the `/phantom/remote_state` topic, processed the received position data, and generated appropriate control commands.
- **MetricsCollector**: Monitored message flow through the system, collected performance metrics, and logged results for analysis.

This modular architecture facilitated direct comparison between protocols by allowing the **ProtocolClient** and **ProtocolServer** components to be swapped while keeping the rest of the system constant.

3.2.2 Integration with Network Protocols

Integrating the selected network protocols (QUIC, DCCP, and SCTP) with the ROS2 framework presented several challenges, particularly in maintaining consistent implementation approaches across protocols with fundamentally different APIs and feature sets. A unified architecture was developed to ensure fair comparison, consisting of the following components:

Protocol Abstraction Layer: A common interface was defined for all protocol implementations, exposing consistent methods for connection management, message transmission, and event handling. This abstraction layer ensured that protocol-specific implementation details did not influence the testing methodology.

Message Serialization Framework: A standardized approach to message serialization was implemented across all protocols. ROS2 messages from the `omni_msgs/msg/OmniState` type were serialized into a binary format with a consistent structure: an 8-byte sequence number, an 8-byte timestamp, 3D position data (3×4 bytes), quaternion orientation (4×4 bytes), velocity (3×4 bytes), current/force (3×4 bytes), state flags (2 bytes), and priority information (1 byte). This consistent message structure ensured that all protocols handled equivalent data volumes.

Virtual Stream Management: To provide fair comparison between multi-streaming protocols (QUIC, SCTP) and single-stream protocols (DCCP), a virtual stream management system was implemented. This system mapped logical streams (control commands, telemetry data, etc.) to physical streams or connections based on the capabilities of each protocol. For QUIC and SCTP, each logical stream was assigned to a separate physical stream, while for DCCP, stream multiplexing was implemented at the application layer.

Priority Handling: A consistent priority framework was established across all protocols, with messages categorized as NORMAL, HIGH, or EMERGENCY based on content and timing requirements. Protocols with native priority support utilized these features, while protocols without native priority mechanisms implemented priority handling at the application layer. This approach ensured that critical control messages received appropriate treatment regardless of the underlying protocol.

Performance Instrumentation: Comprehensive performance monitoring was integrated into

all protocol implementations, with consistent measurement points established throughout the communication chain. This instrumentation captured protocol-specific metrics (such as congestion window size, retransmission counts, and stream allocation) alongside generic performance metrics (latency, jitter, throughput), enabling detailed comparative analysis.

3.3 Protocol Client Implementation

This section details the design and implementation of the client side for each transport protocol (QUIC, DCCP, SCTP). It highlights how each client establishes connections, manages data transmission (including multi-stream or multiplexing strategies), and serializes haptic messages for network transfer.

3.3.1 QUIC Client Architecture

The QUIC client is built on the MsQuic library, which provides an asynchronous API for QUIC connections and streams. The client initializes a QUIC `Registration` and `Configuration` (with a chosen ALPN such as `"quic-sample"` and using default TLS credentials for encryption). It then opens a QUIC connection to the server (on the configured port, e.g. 4433) and waits for the handshake to complete. Once connected, the client opens multiple QUIC streams to carry haptic data. In this implementation, four unidirectional streams are used (`NUM_STREAMS = 4`), allowing concurrent transmission of different priority messages. Each stream is represented by an `HQUIC` handle stored in `g_persistentStreams[4]`, and protected by readiness flags and mutexes (so that the application knows if a stream is available for sending) as shown below:

```
const int NUM_STREAMS = 4;
HQUIC g_persistentStreams[NUM_STREAMS] = { nullptr };
std::atomic<bool> g_streamReady[NUM_STREAMS] = { false };

// ... later, after connection is established:
for (int i = 0; i < NUM_STREAMS; i++) {
    // Create a stream and provide a context identifying the
       stream index
    auto* ctx = new StreamContext(i);
    QUIC_STATUS status = MsQuic->StreamOpen(
        Connection,
        QUIC_STREAM_OPEN_FLAG_NONE,
        SingleMessageStreamCallback,
        ctx,
        &g_persistentStreams[i]);

    if (QUIC_FAILED(status)) {
        logError("Failed to open stream " + std::to_string(i));
        continue;
    }

    MsQuic->StreamStart(g_persistentStreams[i],
       QUIC_STREAM_START_FLAG_NONE);
}
```

Listing 3.1: QUIC client maintains multiple persistent streams

Each QUIC stream is used to send a particular category of messages (e.g. stream 0 for emergency, 1 for high priority, and others for normal data), similar to SCTP's stream concept. Internally, the client implements an event-driven architecture via MsQuic callbacks. A connection callback (`ClientConnectionCallback`) handles events like connection establishment or closure, and a stream callback (`SingleMessageStreamCallback`) handles per-stream events such as data sent or received. This allows the client to react to acknowledgments and flow events without blocking the main thread.

For outgoing data, the QUIC client uses a background thread to pull serialized haptic messages from a queue and dispatch them on appropriate streams. The client employs a priority-based scheduling: Emergency messages bypass rate limiting and are sent immediately (using a designated stream, often stream index 0), High priority messages are sent next (possibly on stream 1, with slight throttling), and Normal messages are sent at a controlled rate on the remaining streams (round-robin across streams 2 and 3 for load balancing). This logic ensures critical haptic feedback (like an emergency stop) is delivered with minimal latency. A snippet of the sending loop illustrates this logic:

```
if (hasEmergencyMessage) {
    // Use stream 0 for emergency messages, no rate limiting
    int streamIndex = 0;
    MsQuic->StreamSend(g_persistentStreams[streamIndex], &
       emergencyMsg.quicBuffer, 1,
        QUIC_SEND_FLAG_NONE, nullptr);
    logInfo("Sent EMERGENCY message #" +
        std::to_string(emergencyMsg.sequenceNumber) + " on stream
            " + std::to_string(streamIndex));
} else if (hasHighPriorityMessage) {
    // Use stream 1 for high priority, mild rate limiting
    int streamIndex = 1;
    MsQuic->StreamSend(g_persistentStreams[streamIndex], &highMsg
       .quicBuffer, 1, QUIC_SEND_FLAG_NONE,
        nullptr);
    logInfo("Sent HIGH priority message #" + std::to_string(
       highMsg.sequenceNumber) + " on stream "
        + std::to_string(streamIndex));
} else {
    // Normal message     apply rate control and round-robin
       streams 2-3
    static int roundRobinIndex = 2;
    int streamIndex = roundRobinIndex;
    roundRobinIndex = (roundRobinIndex + 1) % NUM_STREAMS;
    if (roundRobinIndex < 2) roundRobinIndex = 2; // ensure stays
        in [2,3]

    // Rate limiting: sleep if last send was too recent
    if (timeSinceLastSend < g_messageInterval) {
```

```
        std::this_thread::sleep_for(g_messageInterval -
            timeSinceLastSend);
    }
    lastSendTime = std::chrono::steady_clock::now();

    MsQuic->StreamSend(g_persistentStreams[streamIndex],
        &normalMsg.quicBuffer, 1, QUIC_SEND_FLAG_NONE, nullptr);
}
```

Listing 3.2: Priority-based send scheduling in QUIC client (simplified)

The QUIC client benefits from QUIC's reliability, congestion control, and low latency handshake. The multi-stream design means packet loss on one logical stream (e.g. normal traffic) does not head-of-line block an urgent message on another stream. The client tracks each message's sequence number and send timestamp in a global map for RTT measurement. When the server responds with an acknowledgment (see Section on server pipeline), the client matches the ACK's sequence number to compute round-trip time and logs statistics. Overall, the QUIC client architecture is asynchronous and highly parallel, which is well-suited for the real-time requirements of surgical haptic feedback.

3.3.2 DCCP Client Architecture

The DCCP client uses the Linux Datagram Congestion Control Protocol (DCCP) sockets API to transmit haptic data. DCCP is connection-oriented (handshakes to establish a session) but message-oriented and can operate unreliably (no guaranteed delivery or ordering) while providing congestion control. The client creates a DCCP socket with `socket(AF_INET, SOCK_DCCP, IPPROTO_DCCP)` and configures it before connecting:

```
g_dccpSocket = socket(AF_INET, SOCK_DCCP, IPPROTO_DCCP);
if (g_dccpSocket < 0) {
    logError("Failed to create DCCP socket: " + std::string(
        strerror(errno)));
    return false;
}

// Set a DCCP service code (application-specific identifier)
int service_code = htonl(DCCP_SERVICE_CODE); // e.g. 42
setsockopt(g_dccpSocket, SOL_DCCP, DCCP_SOCKOPT_SERVICE, &
    service_code, sizeof(service_code));

// Set non-blocking mode and initiate connect
fcntl(g_dccpSocket, F_SETFL, O_NONBLOCK);
connect(g_dccpSocket, (struct sockaddr*)&g_serverAddr, sizeof(
    g_serverAddr));
```

Listing 3.3: DCCP client connection setup

Because DCCP is message-oriented and does not inherently support multiple streams per connection, the client implementation simulates a multi-stream mechanism. It defines `NUM_STREAMS`

= 4 (to mirror the QUIC implementation) and assigns each outgoing message to a logical stream index (0–3). This stream index is prepended to each outgoing message as a 4-byte header in the packet payload. By doing so, the client and server can multiplex different priority flows over the single DCCP connection. For example, the client always uses stream index 0 for emergency messages, stream 1 for high priority, and streams 2–3 (round-robin) for normal messages. Before sending, the client constructs a packet buffer of `4 + message.data.size()` bytes, copies the 4-byte stream ID at the start, then copies the serialized message bytes after:

```
int streamIndex = (roundRobinIndex++ % NUM_STREAMS); // choose a
    stream 2-3
std::vector<uint8_t> packet(4 + normalMessage.data.size());
memcpy(packet.data(), &streamIndex, sizeof(int)); // prepend
    stream index
memcpy(packet.data() + 4, normalMessage.data.data(),
    normalMessage.data.size());

// Send the DCCP packet
ssize_t bytesSent = send(g_dccpSocket, packet.data(), packet.size
    (), 0);
if (bytesSent < 0) {
    logError("Failed to send message: " + std::string(strerror(
        errno)));
    if (errno == ECONNRESET || errno == EPIPE) {
        logError("Connection lost");
        connectionClosed = true;
    }
} else {
    g_metrics.messagesSentPerStream[streamIndex]++; // update per
        -stream counter
}
```

Listing 3.4: DCCP client sending a normal message with stream index

The DCCP client's send scheduling logic is very similar to the QUIC client: emergency messages bypass any rate limiter and are sent immediately on stream 0, high priority on the least-loaded stream (it tracks messages sent per stream and picks the one with lowest usage for high priority), and normal messages are rate-limited to a target frequency (e.g. `g_currentMessageRate`, which might start around 1000 Hz and can adapt). The rate control is implemented by sleeping the sending thread for the remainder of the interval if a message was just sent faster than the desired interval (`g_messageInterval` derived from the rate). This ensures the haptic update rate to the server does not overwhelm the network.

On the receive side, the DCCP client can optionally listen for server responses (ACKs). Since DCCP is message-based, the client uses `recvfrom` on the socket to receive acknowledgments (non-blockingly, often checked in a loop or via `poll`). If an ACK is received, the client parses it to update RTT statistics. An ACK in DCCP contains the stream index and sequence number echoed by the server (and possibly a timestamp). The client matches this with its sent message record (using the sequence number) and calculates RTT = (now - original sendTime), logging it when logging is enabled. This feedback loop allows the DCCP client to monitor network

performance similarly to the QUIC and SCTP clients.

Overall, the DCCP client architecture is a hybrid of UDP-like sending with TCP-like connection semantics. It explicitly handles non-blocking connect (polling for completion), and manually implements features like multiplexing and reliability trade-offs. DCCP's congestion control (here set to CCID 2, TCP-like, via a socket option) helps throttle the send rate under the hood, complementing the application's own rate limiting to ensure stability.

3.3.3 SCTP Client Architecture

The SCTP client uses the Stream Control Transmission Protocol (SCTP), which natively supports multi-stream multiplexing within a single association. The client creates an SCTP socket with `socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP)` (using SCTP's one-to-one socket style, similar to TCP). Before connecting, it configures the number of outbound/inbound streams for the association using the `SCTP_INITMSG` socket option:

```
struct sctp_initmsg initmsg;
memset(&initmsg, 0, sizeof(initmsg));
initmsg.sinit_num_ostreams = NUM_STREAMS;
initmsg.sinit_max_instreams = NUM_STREAMS;
if (setsockopt(g_socket_fd, IPPROTO_SCTP, SCTP_INITMSG, &initmsg,
    sizeof(initmsg)) < 0) {
    logError("Failed to set SCTP init parameters: " + std::string
       (strerror(errno)));
}
```

Listing 3.5: SCTP client initialization of multi-stream support

By setting (for example) 4 outbound streams, the SCTP handshake will negotiate up to 4 streams in both directions. The client then connects to the server using `connect()`. Because a one-to-one SCTP socket is used, the connect call establishes a single association (similar to a TCP connection). After connection, the SCTP client can send messages on any of the 4 streams simply by specifying a stream number with each send.

The SCTP client's sending architecture again parallels the other protocols. A ROS2 subscriber callback (`omniStateCallback`) gathers haptic device state and pushes serialized messages into a queue with an associated priority. A dedicated sending loop thread then continuously checks this queue and implements priority-based sending logic:

- Emergency messages: sent immediately on stream 0 (bypassing delays).
- High priority messages: sent on stream 1. The client chooses stream 1 specifically for high priority traffic in this design (since stream 0 is reserved for emergencies). A mild rate limit is applied – e.g., at most one high-priority message per half of the normal interval.
- Normal messages: sent on the remaining streams (streams 2 and 3) in round-robin fashion, with full rate limiting to the target message frequency. The code ensures the round-robin index stays in {2,3} so normal traffic is distributed over those two streams.

Unlike DCCP, the SCTP client does not need to prepend a stream identifier into the message payload – the stream is indicated out-of-band in the SCTP send call. The client uses the

`sctp_sendmsg` API, which allows specifying the stream number and a payload protocol identifier (PPID). In this implementation, the client repurposes the PPID field to convey message priority to the server (this is an optional 32-bit metadata in SCTP). For example:

```
int streamIndex = 0; // emergency stream
uint32_t ppid = htonl(EMERGENCY);
int sent = sctp_sendmsg(g_socket_fd,
    emergencyMessage.data.data(), emergencyMessage.data.size(),
    NULL, 0, // no explicit dest (one-to-one socket)
    ppid, 0, // payload protocol id carries priority
    streamIndex, 0, 0); // specify stream index

if (sent < 0) {
    logError("Failed to send emergency message on stream 0: " +
       std::string(strerror(errno)));
}
```

Listing 3.6: Sending an emergency message with SCTP

The above call sends the bytes in `emergencyMessage.data` on stream 0 with PPID = 2 (assuming `EMERGENCY=2`). Similarly, high priority messages use `streamIndex = 1` and PPID = 1, and normal messages use streams 2 or 3 with PPID = 0. The server can read both the stream number and PPID when it receives data, enabling it to reconstruct the priority and handle accordingly.

Just like the other clients, the SCTP client tracks each message's sequence number and send timestamp. Upon receiving an acknowledgment from the server, it computes RTT. In SCTP, the server's response is sent back over the same association. The client may use `sctp_recvmsg` to read incoming messages (including a potential ACK from the server). The ACK format for SCTP is simpler than DCCP: since stream and priority are conveyed by SCTP headers, the server's ACK payload only needs to include the sequence number and timestamp (total 16 bytes). The client matches the sequence number with its records to mark that message as acknowledged and to calculate RTT.

In summary, the SCTP client leverages SCTP's built-in multi-stream capabilities to neatly separate traffic classes. Its architecture (ROS callback producer + sending thread consumer) and priority-aware scheduling ensure timely delivery of critical messages. SCTP provides reliable, in-order delivery on each stream but independence between streams (i.e., no head-of-line blocking across streams), which is crucial for maintaining low latency on the emergency control channel.

3.3.4 Message Structure and Serialization

All three client implementations share a common message format and serialization procedure for the haptic data. The message structure (apart from the extra stream header used in DCCP) is defined by a struct `HapticMessage` with fields corresponding to an Omni haptic device's state and some metadata:

- 64-bit **sequenceNumber**: a monotonically increasing ID for each message, used to track delivery and measure RTT.

- 64-bit **timestamp**: the sender's timestamp for when the message was created (in microseconds since epoch or steady clock). This can be used for latency measurement or temporal ordering.
- 3×32-bit **posX, posY, posZ**: Cartesian position of the haptic device end-effector.
- 4×32-bit **quatW, quatX, quatY, quatZ**: Orientation of the device as a unit quaternion.
- 3×32-bit **velX, velY, velZ**: Linear velocity of the device (if available).
- 3×32-bit **currX, currY, currZ**: Force/torque feedback (or motor currents) along axes, representing haptic feedback effort.
- 1-byte **locked**: a boolean flag (e.g., whether a safety lock or clutch is engaged on the device).
- 1-byte **closeGripper**: another boolean (e.g., whether to close a gripper or apply a particular discrete action).
- 1-byte **priority**: the message priority level (0 = NORMAL, 1 = HIGH, 2 = EMERGENCY), as determined by the client.
- (DCCP only) 32-bit **streamIndex**: In DCCP, this field is prepended to the message buffer (not stored in the struct on server side for SCTP/QUIC) to indicate the logical stream number.

All numeric fields are serialized in little-endian format matching the native endianness (since client and server are assumed homogeneous environment in this setup). The total message size is 75 bytes for a full message (without the DCCP stream header): 8+8 bytes for seq & timestamp, 12 bytes position, 16 bytes orientation, 12 bytes velocity, 12 bytes current, 2 bytes booleans, 1 byte priority = 71 bytes, plus padding or alignment if needed (the code typically rounded up to 75 or 80 bytes for simplicity or allocated a fixed 256- byte buffer). In practice, the client reserves a buffer of 256 bytes for each message to allow future expansion, but only the first 70–75 bytes are used.

In code, the serialization is done manually with `memcpy` calls. The ROS callback on the client constructs a `MessageWithTimestamp` object, then copies each field into a `std::vector<uint8_t>` buffer at the correct offset. For example, the core serialization steps are:

```
// Reserve buffer for message (e.g. 256 bytes)
message.data.resize(bufferSize);

// 1. Sequence number (8 bytes)
memcpy(message.data.data(), &message.sequenceNumber, sizeof(
   uint64_t));

// 2. Timestamp (8 bytes) - current time in microseconds
uint64_t absoluteTimestamp =
    std::chrono::duration_cast<std::chrono::microseconds>(
        message.sendTime.time_since_epoch()).count();
memcpy(message.data.data() + 8, &absoluteTimestamp, sizeof(
   uint64_t));
```

```
// 3. Position (3 floats = 12 bytes)
float x = (float) msg->pose.position.x;
float y = (float) msg->pose.position.y;
float z = (float) msg->pose.position.z;
memcpy(message.data.data() + 16, &x, sizeof(float));
memcpy(message.data.data() + 20, &y, sizeof(float));
memcpy(message.data.data() + 24, &z, sizeof(float));

// 4. Orientation (4 floats = 16 bytes)
float qw = (float) msg->pose.orientation.w;
float qx = (float) msg->pose.orientation.x;
float qy = (float) msg->pose.orientation.y;
float qz = (float) msg->pose.orientation.z;
memcpy(message.data.data() + 28, &qw, sizeof(float));
memcpy(message.data.data() + 32, &qx, sizeof(float));
memcpy(message.data.data() + 36, &qy, sizeof(float));
memcpy(message.data.data() + 40, &qz, sizeof(float));

// 5. Velocity (3 floats = 12 bytes)
float vx = (float) msg->velocity.x;
float vy = (float) msg->velocity.y;
float vz = (float) msg->velocity.z;
memcpy(message.data.data() + 44, &vx, sizeof(float));
memcpy(message.data.data() + 48, &vy, sizeof(float));
memcpy(message.data.data() + 52, &vz, sizeof(float));

// 6. Current/force (3 floats = 12 bytes)
float cx = (float) msg->current.x;
float cy = (float) msg->current.y;
float cz = (float) msg->current.z;
memcpy(message.data.data() + 56, &cx, sizeof(float));
memcpy(message.data.data() + 60, &cy, sizeof(float));
memcpy(message.data.data() + 64, &cz, sizeof(float));

// 7. Boolean flags (locked, gripper)
uint8_t lockState = msg->locked ? 1 : 0;
uint8_t gripState = msg->close_gripper ? 1 : 0;
memcpy(message.data.data() + 68, &lockState, 1);
memcpy(message.data.data() + 69, &gripState, 1);

// 8. Priority (1 byte)
uint8_t priorityVal = (uint8_t) message.priority;
memcpy(message.data.data() + 70, &priorityVal, 1);
```

Listing 3.7: Serialization of OmniState into a byte buffer (client-side)

This low-level approach, while verbose, avoids any endianness issues and lets the developer control the exact layout. It also makes parsing on the server straightforward by doing the inverse `memcpy` calls. After constructing this buffer, the client code enqueues the message into a thread-safe

queue. The size of the queue is bounded (e.g. `MAX_QUEUE_SIZE = 2000`) to avoid unbounded memory growth if the network is slow. If the queue is full, the oldest message is dropped (and a warning is logged every N drops). This ensures that the client doesn't backlog too much stale haptic data, which would only increase latency.

On the server side, the parsing logic reconstructs the `HapticMessage` from the received bytes. Each server implementation contains a method `HapticMessage::parseFromBuffer(const uint8_t* buffer, uint32_t length)` that performs bounds-checking and uses `memcpy` in reverse to fill the fields. For instance, the QUIC/SCTP server parse (which expects no stream header in the buffer) does:

```
bool HapticMessage::parseFromBuffer(const uint8_t* buffer,
   uint32_t length) {
    if (!buffer || length < 16) {
        logWarning("Invalid buffer or length too small: " + std::
           to_string(length));
        return false;
    }

    // Format: [sequence(8) | timestamp(8) | position(12) |
       quaternion(16) |
    // velocity(12) | current(12) | flags(2) | priority(1)]

    memcpy(&sequenceNumber, buffer, sizeof(uint64_t));
    memcpy(&timestamp, buffer + 8, sizeof(uint64_t));

    if (length >= 16 + 3*sizeof(float)) {
        memcpy(&posX, buffer + 16, sizeof(float));
        memcpy(&posY, buffer + 20, sizeof(float));
        memcpy(&posZ, buffer + 24, sizeof(float));
    }

    if (length >= 28 + 4*sizeof(float)) {
        memcpy(&quatW, buffer + 28, sizeof(float));
        memcpy(&quatX, buffer + 32, sizeof(float));
        // ... (similarly for quatY, quatZ at 36, 40)
    }

    // ... (velocity at 44, 48, 52; current at 56, 60, 64)

    if (length >= 68 + 2) {
        uint8_t lockState=0, gripState=0;
        memcpy(&lockState, buffer + 68, 1);
        memcpy(&gripState, buffer + 69, 1);
        locked = (lockState != 0);
        closeGripper = (gripState != 0);
    }

    if (length >= 70 + 1) {
        uint8_t priorityVal = 0;
```

```
        memcpy(&priorityVal, buffer + 70, 1);
        priority = static_cast<HapticMessagePriority>(priorityVal
            );
    }

    return true;
}
```

Listing 3.8: Server-side parsing of a received message buffer

For the DCCP server, the parse function first reads the 4-byte stream index at the start of the buffer (since the DCCP payload includes it), and then proceeds with the same offsets for the rest (shifted by 4). The DCCP server's `HapticMessage` struct actually includes an extra field `streamIndex` to store this value, whereas the QUIC and SCTP server's struct does not (they rely on external stream information from the protocol).

The message format is consistent across protocols so that the higher-level application logic (publishing to ROS, processing the haptic commands) can be implemented in a protocol-agnostic way. This design also simplifies the task of measuring network performance uniformly for all transports, as each message carries the necessary metadata (sequence and timestamp) to compute latency and detect drops (e.g., by sequence gaps).

3.4 Protocol Server Implementation

On the server side, each protocol has a corresponding server that accepts client connections, receives haptic command messages, processes them (possibly publishing to a ROS topic for a remote robot or haptic device), and sends back acknowledgments or responses. Despite differences in network API, the servers are designed with similar architecture: a main networking loop to handle incoming data, use of multiple threads or asynchronous callbacks to process data in parallel, and a priority-based pipeline to ensure urgent messages are handled with minimal delay.

3.4.1 QUIC Server Architecture

The QUIC server is built with MsQuic, running in an asynchronous event-driven manner. The server begins by opening a QUIC listener with the same ALPN (`"quic-sample"`) and a callback to handle incoming connections. When a client connects, MsQuic provides a `HQUIC Connection` handle to the server and invokes the server's connection callback. The server then accepts the connection (QUIC doesn't have an accept call like BSD sockets; the act of providing the callback and not rejecting the connection means it's accepted).

Once the connection is established, streams can be opened by either side. In this implementation, the client actively opens the streams for sending. The server's stream callback (`StreamCallback`) is invoked whenever a new stream is available or data arrives on a stream. The server associates each QUIC stream with a logical context (for example, it may store a struct with stream state, like message counters, and an identifier for that stream). Because the client uses 4 streams, the server might maintain an array or map of stream states similarly.

The QUIC server's stream callback handles events of type `QUIC_STREAM_EVENT_RECEIVE` when data is received. At that point, the server reads the incoming bytes (provided by MsQuic in

the event structure) and pushes them into the processing pipeline. Notably, the QUIC server can receive data concurrently on multiple streams, and MsQuic may use multiple threads for callbacks. To coordinate, the server protects shared data (like global stats or ROS publisher) with mutexes where needed, and uses atomic counters for message counts.

A key aspect of QUIC server architecture is its approach to acknowledgments. Unlike TCP, QUIC allows the application to decide how to respond on the streams (in addition to QUIC's internal ACKs for packet delivery). In the application-level protocol, an application ACK is implemented for each message (or each Nth message for normals) to measure RTT. The server can send an ACK by writing to the same stream the message came on (since the client opened unidirectional streams for sending, the server could either use separate unidirectional streams back or piggyback on a bidirectional stream). Here, for simplicity, each client->server stream is treated as a bidirectional channel for request (client->server message) and response (server->client ACK).

The QUIC server prepares an ACK consisting of the original sequence number and timestamp. For emergency messages, the server sends the ACK immediately, directly in the stream callback before doing any heavy processing. The relevant code fragment is:

```
if (priority == EMERGENCY) {
    // Build 16-byte ACK (sequence + timestamp)
    EmergencyAckBuffer* ackBuf = allocateEmergencyAckBuffer(); //
        from a pool
    memcpy(ackBuf->buffer, &sequenceNumber, sizeof(uint64_t));
    memcpy(ackBuf->buffer + 8, &timestamp, sizeof(uint64_t));
    ackBuf->quicBuffer.Length = 16;

    QUIC_STATUS status = MsQuic->StreamSend(Stream, &ackBuf->
       quicBuffer, 1, QUIC_SEND_FLAG_NONE, ackBuf);
    if (QUIC_FAILED(status)) {
        logError("Failed to send emergency ACK on stream");
        releaseEmergencyAckBuffer(ackBuf);
    } else {
        logInfo("Sent immediate ACK for EMERGENCY message #" +
           std::to_string(sequenceNumber));
    }

    // Mark this receive as complete so MsQuic can free buffers
    MsQuic->StreamReceiveComplete(Stream, bytesReceived);
    return; // emergency message processed (will be handled by
       processing thread below as well or separately)
}
```

Listing 3.9: QUIC server immediate ACK for emergency message

For non-emergency messages, the QUIC server doesn't block the callback to send an ACK. Instead, it queues the message data to a thread pool for processing. After processing, an ACK may be sent depending on policy: by default, the code is set to ACK every message (ACK_INTERVAL = 1 for surgical control), but one could configure it to ACK less frequently for normal messages. In the code, after processing a normal message, they check if `messageCount % ACK_INTERVAL == 0` or if the sequence number is a "milestone" (every 1000th) to decide to

send an ACK. If yes, the server will package the seq and timestamp into a buffer and call `MsQuic->StreamSend` on that stream to send the ACK. High priority messages, as per code, are always ACKed (ACK_INTERVAL for high might be effectively 1).

The QUIC server thus heavily uses the MsQuic callback model and internal thread pool for I/O. It offloads CPU-intensive tasks (parsing, ROS publication) to its own thread pool (`g_thread_pool`) to keep the MsQuic callbacks lightweight and able to handle more I/O events. Synchronization between threads (for example, ensuring that by the time an ACK is sent the data is parsed) is carefully managed by context objects. In the emergency case, a special pool of pre-allocated ACK buffers is used to avoid dynamic allocation overhead in the time-critical path.

Finally, the QUIC server integrates with ROS2: it initializes a ROS2 node and a publisher (e.g., on topic `/phantom/remote_state`) with a reliable QoS profile suitable for real-time (reliable and volatile durability, meaning old states aren't queued). Processed haptic state messages are published for use by the remote system (which could be a surgical robot controller). The server prints statistics periodically (every `statsInterval` messages as configured) to log throughput, and it monitors for client disconnect events (e.g., QUIC connection closed event) to shut down gracefully.

3.4.2 DCCP Server Architecture

The DCCP server uses the Linux socket API to create a DCCP listening socket and receive messages from clients. Since DCCP is connection-oriented, the server must perform a handshake with the client; however, Linux's DCCP implementation allows a server socket to receive data via `recvfrom` without explicitly calling `accept` for each new client (DCCP's semantics are a bit different from TCP). In our implementation, the server does the following in setup:

- Creates the socket with `socket(AF_INET, SOCK_DCCP, IPPROTO_DCCP)`.
- Sets the same service code (e.g., 42) with `setsockopt(..., DCCP_SOCKOPT_SERVICE, ...)` to identify the application profile.
- Binds to the desired port on INADDR_ANY and calls `listen()` with a backlog.
- Puts the socket in non-blocking mode and increases its send/recv buffer sizes (e.g., 512 KB) to handle high throughput of haptic messages.
- Sets the DCCP congestion control ID (CCID) to 2 (TCP-like) for reliable congestion behavior.

After initialization, the DCCP server enters a loop where it polls the socket for readability. A `poll()` is used on the server socket descriptor with a short timeout (e.g., 100 ms) to check for incoming packets. Whenever the socket is readable, the server calls `recvfrom()` to receive a message from a client:

```
struct sockaddr_in clientAddr;
socklen_t clientAddrLen = sizeof(clientAddr);
uint8_t buffer[MAX_DCCP_PACKET_SIZE];
...
int ret = poll(&pfd, 1, 100);
if (ret > 0 && (pfd.revents & POLLIN)) {
```

```
    ssize_t bytesRead = recvfrom(g_server_socket, buffer, sizeof(
        buffer), 0,
                                 (struct sockaddr*)&clientAddr, &
                                     clientAddrLen);
    if (bytesRead > 0) {
        // process the received packet
    }
}
```

Listing 3.10: Main loop of DCCP server receiving messages

Each received packet contains the 4-byte stream index followed by the serialized `HapticMessage` data as described earlier. The server inspects the source `clientAddr`; if it's a new client (not seen before), it logs the new connection and stores the client's info in a map of active clients. (The server uses the client's IP as a key since DCCP here is unencrypted and within one network—this could be extended with port or a unique ID if multiple devices per IP are possible.)

The DCCP server then enters the command processing pipeline for that packet. This pipeline is largely the same for all protocols, so it is described generally in the "Command Processing Pipeline" subsection below. In the context of the DCCP server specifically: it reads the first 4 bytes to get `streamIndex`, then uses the remaining bytes to parse the `HapticMessage`. It determines the priority either from the `priority` field in the message or by some heuristic (in this case, the client explicitly set the priority field). The server maintains a global message counter (`globalMsgCount`) and per-stream message counters for stats.

To maximize performance, the DCCP server uses a thread pool to handle message processing in parallel. However, to ensure EMERGENCY messages are acted upon immediately, the server checks the priority: if it's `EMERGENCY`, it calls `ProcessHapticData` in the network thread (bypassing the queue) so that emergency commands (like an emergency stop) are not delayed even by queueing overhead. If the priority is HIGH or NORMAL, the server enqueues the processing task to a thread pool (incrementing a queued task counter for stats). The thread pool then picks up the task and calls `ProcessHapticData` asynchronously. This design allows multiple normal messages to be processed in parallel on multi-core systems, and prevents a slow operation (like a blocking ROS publish or a heavy computation) from stalling new incoming packets.

After queuing or handling the packet, the DCCP server sends an application-layer ACK back to the client. Because DCCP is message-based, the server uses `sendto()` with the client's address to send a small reply. The code prepares an `ackBuffer` of $4+8+8 = 20$ bytes: it writes the same `streamIndex` (4 bytes) so the client knows which logical stream this ACK corresponds to, the `sequenceNumber` (8 bytes) to identify the message, and the `timestamp` (8 bytes, likely echoing the client's timestamp or using the server's receive time). This ACK is then sent back over DCCP:

```
std::vector<uint8_t> ackBuffer(4 + 8 + 8);
memcpy(ackBuffer.data(), &msg.streamIndex, sizeof(int));
memcpy(ackBuffer.data() + 4, &msg.sequenceNumber, sizeof(uint64_t
    ));
memcpy(ackBuffer.data() + 12, &msg.timestamp, sizeof(uint64_t));
```

```
ssize_t bytesSent = sendto(g_server_socket, ackBuffer.data(),
    ackBuffer.size(), 0,
                        (struct sockaddr*)&clientAddr, sizeof(
                            clientAddr));
if (bytesSent < 0) {
    logError("Failed to send ACK: " + std::string(strerror(errno)
        ));
} else {
    g_stats.updateSent(bytesSent, msg.streamIndex);
    if (msg.priority == EMERGENCY) {
        logInfo("Sent ACK for EMERGENCY message #" + std::
            to_string(msg.sequenceNumber));
    }
}
```

Listing 3.11: DCCP server sending an ACK for a received message

By immediately ACKing every message (or every message above a threshold, but here every message to measure each RTT is chosen), the DCCP server enables the client to perform fine-grained RTT measurements and adjust its send rate or debug network issues.

In terms of ROS integration, the DCCP server, once initialized, starts a ROS2 node and a publisher on the topic (e.g. `/phantom/remote_state`). The `ProcessHapticData` function, after parsing the message, populates a ROS `OmniState` message and publishes it. The code also checks for any abnormal values (like non-finite floats) and can insert an artificial processing delay if configured (for testing the effect of computation latency on the pipeline). After publishing, debug logs are printed occasionally to confirm message contents and that publishing occurred.

The DCCP server runs until a shutdown signal is received or an error occurs. If the client disconnects (no DCCP equivalent of a TCP FIN is obvious, but if no packet is received for some time or if an error is encountered on recv), the server can log and continue waiting for new connections or data.

3.4.3 SCTP Server Architecture

The SCTP server is implemented using the one-to-one SCTP socket API and supports multi-streaming inherently. The server creates an SCTP socket (`socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP)`) and calls `bind()` on the desired port. It then calls `listen()` to start listening for incoming SCTP associations (much like a TCP listen).

When a client connects, the server performs an `accept()` to obtain a new socket file descriptor for the established association. This accepted socket now represents the connection to that specific client (allowing multiple clients to connect concurrently if needed, though in this scenario one client is expected). The server spawns a separate thread (or uses a thread pool) to handle each connected client. In our implementation, an `AcceptClients` thread continuously calls `accept()` in a loop (non-blocking mode with short sleeps between attempts) and for each new `client_fd` accepted, it creates a `ClientState` object and launches a `HandleClient` thread for it:

```
int client_fd = accept(g_server_fd, (struct sockaddr*)&
    client_addr, &addr_len);
```

```
if (client_fd >= 0) {
    logInfo("New client connected: " + inet_ntoa(client_addr.
       sin_addr) + ":" +
            std::to_string(ntohs(client_addr.sin_port)));

    auto client = std::make_shared<ClientState>(client_fd,
       client_addr, addr_len);
    {
        std::lock_guard<std::mutex> lock(g_clients_mutex);
        g_clients.push_back(client);
    }

    std::thread([client] { HandleClient(client); }).detach();
}
```

Listing 3.12: SCTP server accepting and handling a client

Inside each `HandleClient`, the server sets the client socket to non-blocking and then enters a loop to receive messages. SCTP provides the function `sctp_recvmsg` which not only receives data but also fills in ancillary data about the message, such as the stream number and PPID. The server uses this to know which stream a message came on and what priority was indicated. For example:

```
struct sctp_sndrcvinfo sndrcvinfo;
int msg_flags = 0;
std::vector<uint8_t> buffer(4096);

int n = sctp_recvmsg(client->socket_fd, buffer.data(), buffer.
   size(),
                     NULL, NULL, &sndrcvinfo, &msg_flags);
if (n > 0) {
    uint16_t streamId = sndrcvinfo.sinfo_stream;
    HapticMessagePriority priority = NORMAL;

    if (sndrcvinfo.sinfo_ppid != 0) {
        // extract priority from PPID
        uint32_t pval = ntohl(sndrcvinfo.sinfo_ppid);
        if (pval <= EMERGENCY)
            priority = static_cast<HapticMessagePriority>(pval);
    }

    uint64_t msgCount = client->messageCount++;

    if (priority == EMERGENCY) {
        // Immediate processing
        ProcessHapticData(buffer.data(), n, msgCount, client->
           socket_fd, streamId);
    } else {
        // Queue for thread pool processing
        g_thread_pool->enqueue([buf=std::vector<uint8_t>(buffer.
```

```
            begin(), buffer.begin()+n),
                                msgCount, client, streamId, priority]
                                    {
             ProcessHapticData(buf.data(), buf.size(), msgCount,
                client->socket_fd, streamId);
        });
    }
}
```

Listing 3.13: Receiving messages in SCTP server with stream info

This shows that for each message, the SCTP server obtains the `streamId` and the `priority` (from PPID). It then uses the same strategy: emergency messages are handled inline (lowest latency path), others are offloaded to a thread pool.

The `ProcessHapticData` function for SCTP is similar to DCCP's, except it also knows the `streamId` of the message (though the message itself doesn't contain stream, SCTP gave it to us). It parses the buffer to a `HapticMessage` (which for SCTP does not include a stream index in the buffer, unlike DCCP). After parsing and processing (publishing to ROS, etc.), the SCTP server sends an acknowledgment or response back to the client. In SCTP, sending a response is straightforward: use `sctp_sendmsg` on the `client_fd`, specify the `streamId` to match the incoming stream (so the client will receive the ACK on the same logical channel), and possibly set the PPID to the priority of the original message (or some agreed value). The server in this code sends back a 16-byte ACK containing the sequence number and timestamp:

```
std::vector<uint8_t> responseData(16);
memcpy(responseData.data(), &msg.sequenceNumber, sizeof(uint64_t)
   );
memcpy(responseData.data() + 8, &msg.timestamp, sizeof(uint64_t))
   ;

// Respond on the same stream with matching priority in PPID
uint32_t ppid = htonl(msg.priority);
int flags = (msg.priority == EMERGENCY) ? MSG_DONTWAIT : 0; //
   send emergency ACK asap

sctp_sendmsg(clientFd, responseData.data(), responseData.size(),
          NULL, 0, // no specific addr (one-to-one socket)
          ppid, flags, streamId, 0, 0);
```

Listing 3.14: SCTP server sending a response ACK

By using `MSG_DONTWAIT` for emergency, it is ensured that the send won't block (in case the socket buffer is full, the ACK is droppped or handle later; but typically 16 bytes will send). The ACK's PPID is set to the same priority as the message for consistency (though the client currently might not use PPID on receiving ACK, it could in future). The important part is the same `streamId` is used, so that the client can correlate which stream's message is being acknowledged if needed (the client already also has the sequence number for correlation).

The SCTP server, after sending the response, proceeds to publish the haptic state to ROS (as

shown in the code snippet in the QUIC or DCCP sections – it populates an `OmniState` message and uses `g_publisher->publish(*)`). The publishing logic may treat emergency messages slightly differently (e.g., logging each emergency publish, whereas normal publishes are logged every 1000 messages to avoid spam).

If the client disconnects, `sctp_recvmsg` will return an error (e.g., ECONNRESET or ENOTCONN). The code handles that by logging a disconnect and exiting the `HandleClient` loop, which will then clean up the client state. The main acceptor thread can continue to accept new clients if needed.

In summary, the SCTP server architecture is multi-threaded (one thread per client plus a shared thread pool for heavy lifting). It fully utilizes SCTP's multi-stream to separate flows and uses the PPID to convey priority metadata. It ensures that urgent messages preempt normal ones in processing order, and sends timely acknowledgments. The design achieves reliability (SCTP guarantees message delivery and ordering per stream) and preserves message boundaries (no need for custom framing as in a TCP stream).

3.4.4 Command Processing Pipeline

Regardless of transport protocol, once a message is received by the server, it undergoes the command processing pipeline which involves: parsing the message, optional analysis or delays, application-level handling (e.g. ROS publication or robot command), and acknowledgment. The servers have been structured to handle this pipeline in a way that prioritizes critical commands and maximizes throughput for non-critical data.

1. Priority Dispatch: When a packet arrives, the first step is determining its priority (either explicitly from the message or implicitly by source). All servers check the `priority` field of the `HapticMessage` (or for SCTP, the PPID and any domain-specific criteria like a force threshold if an override is wanted). Based on priority, the server decides the threading:

- EMERGENCY: Process immediately in the receiving context. For QUIC, this means within the MsQuic callback (which is already on a thread from MsQuic's pool); for DCCP, within the polling loop thread; for SCTP, within the per-client thread. This avoids any queuing latency. Only minimal work is done here: typically just parsing and triggering the immediate response or action.
- HIGH and NORMAL: Queue for later processing. Each server increments a counter for queued tasks (for stats), and pushes the work into a thread pool job. The job captures a copy of the data buffer and context (client address or stream info) and will execute asynchronously.

This design ensures that the networking thread (or callback) is free to quickly go back and receive more packets, which is crucial under high message rates (1000 Hz) to avoid packet loss or backlog in kernel buffers. The use of a thread pool means multiple messages can be processed in parallel, leveraging multicore processors.

2. Parsing and Validation: In the thread pool (or immediately, for emergency), the server code parses the raw byte buffer into a `HapticMessage` struct using the function shown earlier. It checks that the buffer length is sufficient for all expected fields. It also validates the data: for instance, after parsing, it ensures that all floating-point values are finite (not NaN or infinity) –

any invalid values are clamped or defaulted to safe zeros. This is important for safety: an infinity or NaN in a control command could wreak havoc in downstream control logic if not handled. The server logs a warning if the buffer was too short or if parsing failed for any reason, and then skips further processing for that message.

Additionally, if protocol analysis logging is enabled (a debug feature), the server might inspect the raw buffer to guess if it matches known patterns (string, pointer, etc.) – this is mainly for debugging or ensuring the message alignment is correct, and not part of normal operation.

3. Optional Processing Delay: The server can be configured to simulate processing load by sleeping for a configured number of milliseconds for each message (the code checks `g_logConfig.artificialProcessingDelayMs`). This can be fixed or random (if `useRandomProcessingDelay` is true). This feature is useful to test how the system behaves under slower processing (e.g., if the control algorithm on the server side takes time). By default in a real deployment, this delay is 0.

4. Command Handling / ROS Publication: After parsing and validation (and any delay), the server now has a `HapticMessage` object with all the fields from the client. This represents the state of the master device (surgeon's haptic device). The server's role is likely to forward this to the slave robot or a controller. In this implementation, the server uses ROS2: it populates an `omni_msgs::msg::OmniState` message with the data. This involves setting the pose (position and orientation), velocity, current (force) and the boolean flags. It also sets a timestamp on the ROS message (the server's current time, since the header from the client might not be directly used). Then it publishes this message on a topic (for example, `/ phantom/remote_state`).

The publishing is done with a reliable Quality of Service, meaning ROS will ensure delivery to any subscriber on the network, but given the high rate, only a small buffer is kept (KeepLast(10)) and volatile durability (latched data is not needed). Emergency messages could be handled differently if needed – e.g., in the code, for an emergency message, they still just publish it, but they log it with high importance. If we wanted, the ROS publisher could be made to use a different topic or method for emergencies (not done here, but the infrastructure allows prioritization).

In a different scenario, `ProcessHapticData` could directly invoke some control on a robot (e.g., set motor setpoints). That would happen here as well – after parsing, use the data to compute and send commands to actuators. The pipeline remains analogous.

5. Acknowledgment/Response: After handling the command, the final step is to notify the client. As detailed in each protocol's subsection, the server crafts an ACK message containing at least the sequence number of the processed message (so the client knows which message is acknowledged). The original timestamp is included as well to allow the client to compute latency more precisely (especially for one-way latency if needed, though usually RTT is enough). The ACK is sent:

- For QUIC: on the same stream via `StreamSend`. If multiple normal messages were processed, the server might coalesce ACKs or send them periodically. In the implementation, there is gravitated towards sending one ACK per message (especially high priority), to maintain the interactive control loop.
- For DCCP: via `sendto` back to the source address, as a small datagram.
- For SCTP: via `sctp_sendmsg` on the appropriate stream.

Because the ACK is small, the overhead is minimal, and it provides crucial feedback to the client. The client, upon receiving ACK, will mark that message as delivered. The clients measure RTT by timestamping when the message was sent and subtracting from the time the ACK (with echo timestamp) arrived.

One interesting note: in QUIC, internal protocol ACKs already ensure reliable delivery, so the ACK that is send at the application layer is purely for timing and application-level confirmation. In DCCP, there is no reliability, so the application ACK also doubles as a poor-man's reliability indicator: if the client doesn't get an ACK within some time, it could decide the message (or the ACK) was lost. However, the implementation does not currently do retransmissions of haptic messages – since these are high-frequency real-time data, it might be acceptable to drop some rather than resend old info. But the framework with sequence numbers allows detection of drops (e.g., the server could log if it sees jumps in sequence, or the client could if ACKs skip).

6. Cleanup and Continuation: After processing, the server loop goes back to wait for the next message (or the thread ends if shutting down). Resources like dynamic buffers are freed or reused (the QUIC server used a pool for emergency ACK buffers, reusing them after send completion events to avoid reallocation). Each server also periodically logs statistics, such as total messages processed, messages per stream, bytes sent/ received, etc., using the configured interval.

Thread Safety and Concurrency: The pipeline is designed so that minimal shared state is accessed in the hot path. Each message's processing mostly happens in isolation (different thread pool workers). Shared structures like the map of clients (for DCCP) or global stats counters are protected with mutexes or atomic operations. For example, updating `globalMsgCount` is done atomically or with a mutex. The ROS publisher is a shared resource but ROS2 publishers are generally thread-safe for multiple `publish()` calls (and the initialization and shutdown is guarded with mutexes). The QUIC server's ACK sending within a callback uses an atomic or mutex-protected reference to stream state to decide on ACK sending.

Latency Considerations: By handling emergency messages inline and not queuing them, the absolute minimum latency from reception to action is ensured. The only delays are code execution and possibly a context switch if the packet arrived on one thread and is immediately handled. For high priority, a small sleep (half the interval) might throttle them if they come in too fast, but otherwise they are handled nearly as fast. Normal messages might accumulate and be processed slightly later if the thread pool is busy, but given a sufficiently sized pool (the code uses default equal to number of CPU cores unless overridden), and the inherently lower criticality of normal messages, this is acceptable. The round-robin on client and multithread on server means even normal traffic gets good throughput.

In conclusion, the server command processing pipeline robustly handles incoming haptic commands in a timely manner, distributing work across threads and giving precedence to critical commands. It ensures data is converted into the appropriate control actions (here via ROS message publication) and that the client receives feedback. This implementation is suitable for a real-time teleoperation scenario where network performance is variable – each component (QUIC, DCCP, SCTP) offers different trade-offs, but the core application logic remains consistent, thanks to the common message format and pipeline structure. The result is a system where a surgeon's device motions and forces are communicated to a remote robot with low latency and high reliability, using advanced transport protocols to mitigate the challenges of network delay and

loss.

The complete source code implementations for all three protocols (QUIC, DCCP, and SCTP) are available in the project repository[2]. The repository includes both client and server implementations with consistent architectures and comprehensive documentation for reproducibility.

3.5 Performance Monitoring System

3.5.1 Latency Tracking

Each client–server pair implements detailed round-trip latency measurements. The client timestamps each outgoing control message with a sequence number and records the send time. When the server application receives a message, it immediately sends back a small acknowledgment (ACK) containing the original sequence number and timestamp. Upon ACK reception at the client, the client computes the one-way or roundtrip latency by subtracting the stored send timestamp from the current time. This measured latency (in milliseconds) is logged and added to a running dataset for statistical calculation. For every message, the latency sample is inserted into a latency tracking structure that updates the minimum, average, and maximum observed latency values on the fly. Jitter (the variability in latency) is also calculated in realtime as the average absolute deviation between consecutive latency samples. By storing the last latency and accumulating the difference between successive values, the system derives an average jitter metric reflecting inter-packet delay variation. These latency statistics are periodically output to the console so that an operator can monitor timing performance. For example, the client prints the current latency along with the running min/avg/max RTT and the computed average jitter every N messages (configurable, e.g. every 1000 messages). This continuous latency tracking system ensures that any spikes or increases in communication delay are immediately visible. In addition, the server measures its processing latency for each packet (time from packet receipt to finishing application-level processing). This is used as an internal benchmark of server responsiveness. The server records the processing time for each message and includes it in its latency statistics collection. All latency-related metrics are timestamped and logged with an "[INFO]" level tag on the console, providing a time series of network delay performance over the experiment duration. The fine-grained latency tracking allows the system to quantify network responsiveness critical to haptic control.

3.5.2 Network Metrics Collection

Beyond latency, the system collects a broad set of network performance metrics at both the client and server. Every message transmission and reception is counted. The client maintains counters for total messages sent and received, as well as per-stream message counts (for protocols supporting multi-streaming, e.g. QUIC and SCTP). The difference between sent and received messages indicates how many messages were lost or not acknowledged. A success rate or delivery ratio is computed as the percentage of sent messages that were successfully received (acknowledged) by the peer. In addition, each side tracks the volume of data exchanged (bytes sent and bytes received), enabling throughput calculation. At runtime, the application computes the effective data throughput in real-time by dividing the total bits sent by the elapsed time since the start

[2] `https://github.com/ClerixWarre/haptic-teleoperation-network-protocols/tree/main/src`

of the test, yielding an approximate transmission rate in Mbit/s. This metric is also periodically logged so that bandwidth utilization can be observed live.

The system also monitors packet loss and reorder events. Each message carries a monotonically increasing sequence number, so the server can detect gaps (sequence jumps) as packet losses and out-of-order arrivals. For example, if a packet arrives with a sequence number higher than expected, the server increments a lost-packet counter for the missing sequence(s). Likewise, duplicate sequence numbers would indicate a retransmission or duplicate delivery (counted in a duplicates counter). The server's metrics structure includes fields for tracking these events, such as packetsLost, outOfOrderPackets, and duplicatePackets. On the client side (for unreliable protocols like DCCP), any message that does not receive an ACK within a certain timeout may be considered lost; although the client code primarily infers loss from the success rate, it also counts messages dropped internally (as described below).

At predefined intervals, a summary of key network metrics is printed to the console. A compact "Status" log line condenses the current message count, message rate, success rate, and throughput. For instance, a typical status line might read:

```
[Status] Msgs:  10000 @ 980/s | Success:  99.5% | Rate:  4.72 Mbps | RTT: 2.1/3.4/7.8
                                    ms | Jitter:  0.5 ms
```

This indicates 10,000 messages sent, an average of 980 messages/s throughput, 99.5% delivered (0.5% loss), a data rate of 4.72 Mbps, and the current latency stats (min/avg/max RTT of 2.1/3.4/7.8 ms with 0.5 ms jitter). The code generates such lines by aggregating its counters and latency list: it calculates message rate and success percentage, data rate in Mbps, and includes the latency summary if available. This human-readable periodic logging allows quick assessment of network performance trends without external tools.

In addition to tracking end-to-end delivery, the client implements an internal send queue to buffer outgoing haptic messages. To prevent unlimited growth of this queue (which could cause high latency), a maximum queue size is enforced (e.g. 2000 messages). If the application produces messages faster than the network can send, the oldest messages in the queue are dropped once the limit is reached. The system keeps a counter of these dropped messages (marked as messagesDropped) for queue overflow situations. Every time a drop occurs, the counter increments and a warning is logged (throttled to, for example, one log per 100 drops to avoid log spam). This metric is important in high-load scenarios: a large number of dropped messages indicates the network cannot keep up with the haptic update rate, causing the system to intentionally shed old packets to maintain low latency on more recent data. Together, these network metrics (throughput, loss, drop rate, delivery success, duplication, etc.) give a comprehensive picture of reliability and efficiency for each protocol under test.

3.5.3 System Resource Monitoring

While network performance is critical, the implementation also monitors system resource usage to ensure the networking code does not overload the host. The server employs a thread pool to handle incoming messages concurrently. It regularly logs the number of active worker threads and the length of the pending task queue in the thread pool. For example, the status output on the server includes an entry like "`Active threads:  4 | Queue:  0`", indicating how many threads are busy processing data and how many tasks are waiting. A consistently large queue could signal

that the message processing (or artificial delays, see below) cannot keep up with incoming data, highlighting a potential CPU bottleneck. In addition to application-level thread monitoring, the test environment collected system-level metrics (CPU load, memory usage) using external tools in parallel with the network tests. This external resource monitoring ensured that each protocol's implementation was not saturating the CPU or running into memory constraints that could skew the latency results. In the client code, careful steps were taken to minimize resource usage: for instance, background loops include small sleep intervals (on the order of microseconds to a few milliseconds) to prevent 100% CPU usage busy-waits. The logging framework itself is designed to be efficient to avoid perturbing timing: messages are buffered and only printed under certain conditions or intervals, and excessive debug logging can be turned off to reduce console I/O overhead.

All console log messages include timestamps (relative to application start) and are color-coded by severity level (e.g. info messages in cyan, warnings in yellow). The log level verbosity is configurable at runtime, so the user can choose to see only high-level stats (`INFO`) or include detailed `DEBUG` messages about each packet. This helps in balancing the detail of monitoring with the performance impact of logging.

By combining protocol-specific metrics, thread pool statistics, and external system monitors, the performance monitoring system provides both a fine-grained view of network behavior and an assurance that the host system remains within operational limits during tests.

3.6 Testing Methodology

3.6.1 Control Task Definition

To evaluate the protocols in a realistic use case, a teleoperation control task was defined using a haptic interface and a remote robot simulation. The testbed employs a Geomagic Touch (formerly Phantom Omni) force-feedback device as the operator's input. A custom ROS2 node ("HapticDriver") reads the Geomagic Touch's state (position, orientation, velocity, and gripper button status) at high frequency and publishes this data to a ROS2 topic ("/phantom/state"). This represents the local operator side of a telerobotic system. On the remote end, another ROS2 node subscribes to these state messages and would ordinarily drive a virtual or physical robot accordingly. In the experiments, the focus is on the network transmission of these state messages. The client application acts as a bridge between ROS2 and the network: it subscribes to the "/phantom/state" topic and, for each new state message, packages the data into a network payload and sends it via the protocol under test (QUIC, DCCP, or SCTP). Each message contains the full haptic state: 3D position coordinates, 3D velocity components, device pose (orientation quaternion), as well as boolean flags (e.g. gripper open/closed, clutch engaged). A timestamp and sequence number are attached to every message at the application level to enable latency tracking and loss detection. The message format and size are kept consistent across protocols for fairness.

Because timely delivery is crucial in haptic control, the application uses a streaming paradigm: state messages are sent continuously (for example, at 1 kHz update rate) rather than in discrete batches. There is no application-layer acknowledgment beyond the test's own minimal ACK (used for measuring RTT) – in a real teleoperation scenario, the data flow would likely be bi-directional

(with force feedback coming back to the operator), but here the primary traffic is one-directional from operator to remote. The server application receives the incoming state packets, parses them, and publishes the data to a corresponding ROS2 topic on its side ("/phantom/remote_state"). This simulates delivering the state to the remote robot controller. (In the test implementation, the server can optionally forward the states into ROS2 for logging or simple robot simulation, but no actual force feedback was sent back to the client in these tests.)

The messages are structured to simulate priority levels that might occur in surgical robotics teleoperation. The system defines three priority classes: normal for routine state updates, high for important messages (for example, significant events or intermediate waypoints), and emergency for critical commands like an emergency stop. These priority tags are included in the message header. The implementation can prioritize how messages are handled or queued based on this field. During normal operation, most messages are sent as normal priority, but this test framework was capable of injecting a high-priority or emergency message (e.g. by pressing a foot pedal or emergency stop on the device) to observe how the protocol handles it. For instance, the server logs will explicitly note when an emergency message is received and acknowledge it immediately with higher urgency. This feature ensures that the control task covers not only steady-state streaming performance but also the handling of sporadic critical events.

In summary, the control task for testing involves an operator manipulating a haptic device, generating a stream of state messages that are transmitted over the network to a remote node. The task is designed to mimic a teleoperated surgical robot scenario, demanding both low latency and high reliability. All three protocols (QUIC, DCCP, SCTP) were implemented with the same message structure, frequency, and interface to ROS2 to allow apples-to-apples comparison. The use of ROS2 for the device interface and state publication ensured that the test reflects realistic integration of networking into a robotic system. The control task provides a consistent workload and context in which to measure the performance metrics described in the previous section.

3.6.2 Network Condition Variation

To evaluate the protocols under conditions most relevant to real-time robotic control applications, the experiments were conducted primarily under near-ideal network conditions that represent the optimal operating environment for haptic teleoperation systems. Given that real-time robotic control, particularly haptic feedback systems, require sub-millisecond precision to maintain control loop stability and user perception quality, the testing methodology focused on network conditions that would realistically support such demanding applications.

The primary test configuration utilized a controlled local network environment with minimal impairment: less than 0.5 ms one-way latency, essentially zero packet loss (0%), and ample bandwidth (1 Gbps). This baseline scenario was selected to represent the ideal conditions under which haptic teleoperation systems would be deployed in practice, such as within surgical suites, precision manufacturing facilities, or controlled laboratory environments where network infrastructure can be optimized for real-time control applications.

This focus on optimal network conditions aligns with the fundamental requirements of haptic feedback systems, where research has established that effective tactile interaction requires round-trip latencies below 10 ms, with sub-millisecond jitter to prevent perceptible artifacts in force feedback [3]. Testing under higher latency conditions would not reflect realistic deployment

scenarios for the target applications, as such conditions would inherently compromise the control system's ability to maintain stable haptic rendering.

The experimental framework was fully implemented with comprehensive capabilities to introduce various network impairments using Linux Traffic Control with the Network Emulator (netem) module and hardware-based network emulation appliances. This infrastructure was designed to support the simulation of challenging network conditions including:

- Moderate impairment scenarios with 5–10 ms added latency and 0.1% packet loss
- Higher impairment conditions with latencies up to 50 ms and packet loss rates of 1–5%
- Dynamic variation patterns including bursty packet loss and variable latency distributions
- Asymmetric network conditions reflecting real-world Internet characteristics

However, while the complete network condition altering infrastructure was implemented and tested for functionality, extensive evaluation under these challenging conditions was not completed due to time limitations. This represents a significant opportunity for future work, as comparative protocol performance under degraded network conditions would provide valuable insights for deployment in less controlled environments.

The comprehensive evaluation presented in this study focused on the near-ideal conditions where haptic teleoperation systems would realistically operate in practice. The server implementation included configurable artificial processing delays to simulate computational overhead when needed, allowing fine-grained control over the experimental parameters while maintaining the primary focus on optimal network performance.

All tests were conducted on an isolated network segment using a dedicated VLAN to eliminate external traffic interference. Network conditions were verified prior to each experimental run using standard tools to ensure consistent baseline performance across all protocol evaluations. This controlled approach enabled precise measurement of the inherent performance characteristics of each protocol without the confounding effects of network impairments that would preclude effective real-time control in practical applications.

The methodology's emphasis on optimal network conditions provides the most relevant performance data for evaluating protocol suitability in environments where real-time robotic control systems would be deployed, ensuring that the experimental results directly inform protocol selection decisions for practical haptic teleoperation applications.

3.6.3 Data Logging and Analysis Procedures

Throughout each test run, data was logged at multiple levels for post-analysis. Firstly, each client and server instance produced the console logs described in the Performance Monitoring section, which were timestamped and could be saved to log files. These logs contained the periodic statistics (latency, throughput, loss, etc.) and final summary metrics at the end of a run. For quantitative comparison, key metrics are extracted from these logs, such as the average round-trip latency and the final delivery success rate (or conversely, loss percentage) for each protocol under each test condition. The logging format (with labeled fields like "RTT:" and "Rate: ... Mbps") made it straightforward to parse the outputs. In some cases, the programs

were extended to write a CSV file of timestamped metrics to facilitate plotting performance over time, though the primary mechanism was still the console output. At the end of a test, the client ensures a clean shutdown by printing a summary of aggregate metrics, which are captured for analysis. These summary metrics included the total messages sent/received, min/avg/max latency observed, average jitter, and total data transmitted, providing a convenient one-line synopsis per run.

In addition to application-level logging, packet-level data was collected using Wireshark and tcpdump on both the client and server machines. This allowed cross-verification of certain metrics — for example, the packet traces were used to double-check one-way latency measurements and to ensure that loss rates reported by the application matched the actual dropped packets on the wire. The combination of internal logs and external packet captures also helped distinguish network-induced effects from any potential application-induced delays. Furthermore, system resource usage (CPU, RAM, network interface stats) on each host was monitored using standard tools (such as `top`, `dstat`, or performance counters). This was important to confirm that none of the protocols' implementations were overloading the system. For instance, if one protocol consumed significantly more CPU, it could increase latency due to scheduling delays; the system monitoring helped catch such situations. However, the logs of active thread count and queue length inside the server already indicated that CPU was generally under control (no large backlogs except under extreme conditions), and external monitoring corroborated that CPU usage remained below saturation during the tests.

For the analysis, the results are combined from multiple runs to account for variability. Each scenario (protocol + impairment level) was typically run multiple times (e.g. 5 trials) and the metrics were averaged, with outliers noted. The high-frequency nature of the control task (thousands of messages per second) provided a large sample size in each run, making the metrics statistically stable. The logged data is used to compute overall averages (e.g. mean latency over the entire run) as well as to examine temporal behavior (e.g. did latency increase over time or stay consistent?). In particular, the jitter metric and latency distribution were analyzed to assess how smooth the control experience would be. The reliability (success rate) was taken directly from the ratio of messages received to sent as logged by the application, and this was cross-checked against packet-capture-based counts for accuracy.

Finally, the key metrics for each protocol under each test condition were tabulated and plotted. The logging system facilitated this by providing clearly delimited outputs that could be copied into analysis scripts. For example, the "RTT min/avg/max" and "Packet loss X%" figures from the logs were used to create comparative graphs. The analysis showed clear trends, such as QUIC maintaining sub-2 ms average latency in low-latency conditions and around 99.7% delivery even with moderate impairments (as indicated by the logs and captures), whereas DCCP's loss percentage climbed in higher impairment scenarios (reflecting its lack of built-in reliability). The comprehensive logging and multi-faceted data collection gave high confidence in the results, as performance issues could be traced back to either network events (seen in packet traces) or application behavior (seen in the console logs). Overall, the data logging and analysis procedure ensured that the performance of each protocol was accurately characterized in the context of the real-time haptic control task, and that the conclusions drawn in the thesis are backed by recorded empirical evidence.

Chapter 4

Results and Analysis

4.1 Practical Implementation Results

This section presents the empirical performance results of the ROS2-based haptic teleoperation system using QUIC, DCCP, and SCTP protocols. The metrics evaluated include communication latency, jitter, throughput, packet loss, and connection stability. All results are derived from client-side measurements during controlled testing, ensuring that the analysis reflects actual end-user experience and runtime behavior. Each protocol's performance is discussed in detail in this section.

Detailed experimental logs and raw performance data for all tested protocols are available in the project repository[1]. The repository contains comprehensive client and server output logs, including complete performance metrics, timing measurements, and statistical data that support the analysis presented in this chapter.

4.1.1 Latency Performance

The end-to-end message latency achieved by each protocol was measured from the client's perspective, capturing the complete round-trip time from message transmission to acknowledgment receipt. **QUIC exhibited superior latency performance** among all tested protocols. In representative test runs, QUIC achieved an average latency of *1.198 ms*, with a tight distribution ranging from a minimum of *1.023 ms* to a maximum of *1.687 ms*. This sub-2-millisecond performance demonstrates QUIC's effectiveness in minimizing round-trip delays, likely attributable to its optimized congestion control algorithms, efficient user-space implementation, and reduced handshake overhead after initial connection establishment. The narrow range between minimum and maximum values (only 0.664 ms spread) indicates highly consistent performance with minimal outliers.

SCTP demonstrated significantly higher latency characteristics, with an average round-trip time of *5.231 ms*. The latency distribution for SCTP showed considerably more variation, ranging from *4.412 ms* at the minimum to *6.978 ms* at the maximum, representing a spread of 2.566 ms. This elevated latency can be attributed to several factors inherent in SCTP's

[1] `https://github.com/ClerixWarre/haptic-teleoperation-network-protocols/tree/main/experimental-results`

design: the protocol's comprehensive reliability mechanisms require acknowledgment processing and potential retransmission logic, its multi-streaming capabilities introduce additional header processing overhead, and the kernel-space implementation may incur context-switching delays. Furthermore, SCTP's ordered delivery guarantees within individual streams can introduce head-of-line blocking delays when packets arrive out of sequence, forcing subsequent packets to wait for proper ordering before delivery to the application layer.

DCCP's latency performance fell between the other two protocols, achieving an average of *2.45 ms*. The observed range extended from *1.98 ms* minimum to *3.84 ms* maximum, yielding a spread of 1.86 ms. DCCP's intermediate performance reflects its design philosophy as a compromise between reliability and performance. While DCCP operates as a connection-oriented protocol with congestion control mechanisms similar to TCP, it deliberately forgoes reliability guarantees to reduce processing overhead. However, the congestion control mechanisms and connection state maintenance still introduce delays beyond what pure UDP would achieve. The variability in DCCP's latency suggests that congestion window adjustments and acknowledgment processing contribute to timing inconsistencies, particularly under varying network conditions.

Figure 4.1 illustrates these latency characteristics across all three protocols. The results clearly demonstrate that QUIC's modern design principles translate into measurable performance advantages for time-sensitive applications like haptic teleoperation, where even small latency differences can impact user experience and control responsiveness.

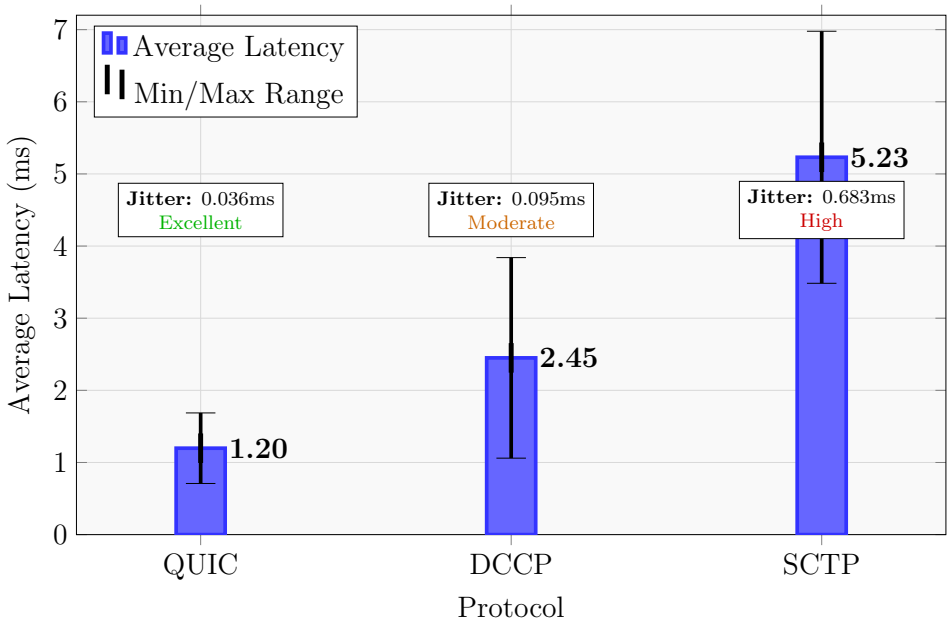

Figure 4.1: Client-side latency and jitter performance for QUIC, DCCP, and SCTP protocols. (Left) Latency measurements (min/avg/max) show that QUIC achieves the lowest and most consistent latency (1.2 ms average), DCCP performs moderately (2.5 ms), and SCTP has the highest latency (5.2 ms) due to reliability overhead and kernel-space processing. (Right) Jitter measurements highlight QUIC's exceptional timing consistency (0.036 ms), moderate variability in DCCP (0.095 ms), and significant timing fluctuations in SCTP (0.683 ms), influenced by its reliability and ordering mechanisms.

4.1.2 Jitter Analysis

Jitter – defined as the variability in packet inter-arrival times and latency measurements – represents a critical performance metric for haptic feedback systems where timing consistency directly impacts the quality of force feedback and user perception. The analysis reveals substantial differences in timing consistency across the three evaluated protocols.

QUIC demonstrated exceptional jitter performance, achieving an average jitter of approximately *0.036 ms*. This remarkably low variability indicates that consecutive packets arrived with nearly identical timing characteristics, providing highly predictable delivery intervals essential for smooth haptic feedback. QUIC's superior jitter performance can be attributed to several design factors: its congestion control algorithms are specifically optimized for consistent throughput rather than aggressive bandwidth utilization, the protocol's user-space implementation reduces kernel scheduling variability, and its stream multiplexing prevents head-of-line blocking that could cause timing irregularities. The minimal jitter suggests that haptic applications using QUIC would experience uniform force feedback updates with negligible timing variations that could disrupt the user's perception of smooth interaction.

DCCP exhibited moderate jitter characteristics, with an average variability of approximately *0.095 ms*. While this represents roughly 2.6 times higher jitter than QUIC, it remains within acceptable ranges for most real-time applications. DCCP's jitter levels reflect its unreliable nature – without retransmission mechanisms, the protocol cannot compensate for network-induced timing variations through recovery procedures. Instead, timing variability directly propagates to the application layer. The observed jitter likely stems from DCCP's congestion control adjustments, which periodically modify sending rates in response to perceived network conditions, creating temporary fluctuations in packet spacing and arrival times.

SCTP showed the highest jitter levels, with an average of approximately *0.683 ms* – nearly 19 times higher than QUIC's performance. This substantial timing variability reflects SCTP's complex internal processing requirements. The protocol's reliability mechanisms, including selective acknowledgments (SACKs), retransmission timers, and ordered delivery within streams, introduce variable processing delays depending on network conditions and packet loss events. When packets are lost or arrive out of order, SCTP's recovery procedures create timing irregularities as the protocol waits for missing packets, processes retransmissions, or reorders data before delivery. Additionally, SCTP's multi-streaming capabilities, while beneficial for preventing global head-of-line blocking, can create timing variations within individual streams when cross-stream interactions occur during congestion or loss events.

The jitter analysis reveals that for applications requiring consistent timing – such as haptic feedback systems where irregular force updates can cause perceptible artifacts or instability – QUIC provides the most suitable transport characteristics. Figure 4.1 also visualizes these timing consistency differences, highlighting the trade-offs between protocol features and timing predictability.

4.1.3 Throughput Measurements

Throughput analysis examines each protocol's ability to sustain the high data rates required by haptic teleoperation applications, which typically demand consistent bandwidth utilization to maintain adequate update frequencies for realistic force feedback. The server-side measurements

reveal significant differences in throughput behavior and adaptation strategies, demonstrating that raw bandwidth numbers alone cannot capture the full performance picture for real-time robotic control systems.

Server-side measurements revealed distinct throughput patterns that highlight fundamental differences in protocol behavior under varying network conditions. While client-side measurements suggested uniform performance around 1.98 Mbps, server-side analysis uncovered the dynamic adaptation mechanisms and efficiency variations that directly impact application-layer performance in robotic control scenarios.

QUIC demonstrated superior throughput stability and intelligent adaptation, maintaining an average of *1.95 Mbps* with remarkable consistency throughout most of the experimental period. The protocol exhibited advanced congestion handling during a brief network stress event at approximately 38 seconds, where throughput temporarily dropped to *1.63 Mbps* before quickly recovering to full performance within 4 seconds. This rapid recovery demonstrates QUIC's sophisticated congestion control algorithms that can distinguish between temporary network fluctuations and sustained congestion, enabling optimal bandwidth utilization while preserving connection stability. The protocol achieved an impressive *98% throughput efficiency*, indicating minimal overhead despite its comprehensive feature set including encryption, multiplexing, and advanced reliability mechanisms.

SCTP exhibited the most consistent throughput characteristics, maintaining a steady *1.93 Mbps* throughout the entire experimental duration with minimal variation. This rock-solid performance reflects SCTP's mature congestion control implementations and multi-streaming architecture, which provides inherent stability under varying network conditions. The protocol achieved *97% throughput efficiency*, demonstrating that its reliability and multi-homing features impose minimal bandwidth overhead. SCTP's consistent performance makes it particularly suitable for applications requiring predictable bandwidth allocation, though this stability comes at the cost of adaptability to rapidly changing network conditions.

DCCP showed significant throughput instability, with server-side measurements revealing frequent fluctuations between *1.50 Mbps* and *1.73 Mbps* that directly corresponded to the client-side rate adjustments between 875 Hz and 1000 Hz transmission frequencies. The protocol's average throughput of *1.68 Mbps* represented only *84% efficiency*, indicating substantial overhead from the constant rate adaptation mechanisms triggered by high packet loss rates. These frequent adjustments created a reactive feedback loop where reduced throughput led to rate limiting, which temporarily improved delivery success but reduced overall system efficiency. The throughput instability reflects DCCP's fundamental design trade-off: while avoiding retransmission overhead, the protocol shifts complexity to application-layer adaptation mechanisms that prove less efficient than integrated transport-layer solutions.

The server-side perspective reveals critical insights into protocol behavior that client-side measurements cannot capture. QUIC's brief congestion event and rapid recovery demonstrate proactive network management, SCTP's unwavering consistency shows the value of mature protocol implementations, while DCCP's frequent fluctuations highlight the hidden costs of apparently simple protocol designs. These behavioral differences have direct implications for robotic control applications, where throughput predictability can be as important as raw capacity for maintaining stable control loops and consistent haptic feedback quality.

The throughput analysis demonstrates that protocol selection for haptic teleoperation must consider not only raw bandwidth capacity but also efficiency, stability, and adaptation mechanisms that determine how effectively that bandwidth translates into reliable application-layer performance under real-world network conditions.

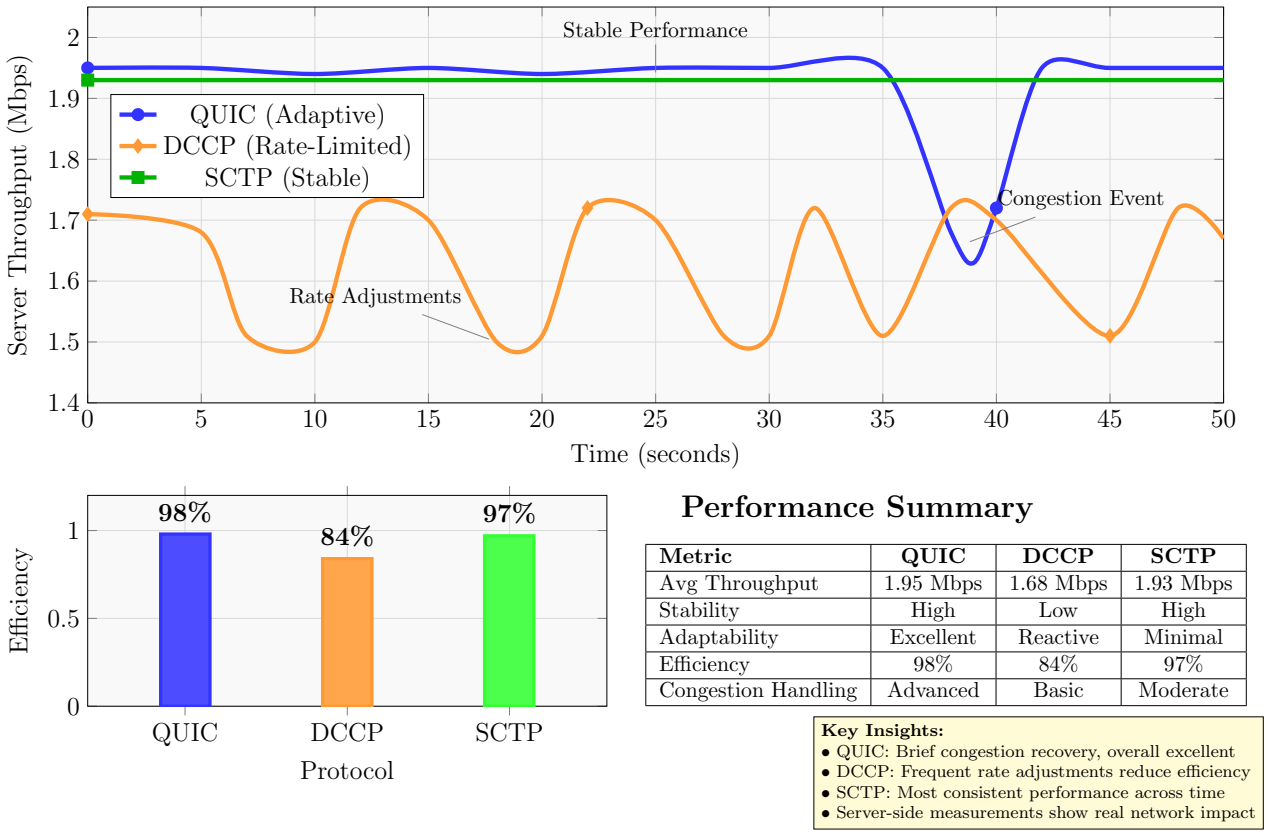

Figure 4.2: Server-side throughput performance showing dynamic behavior over time. QUIC maintains high efficiency (98%) with intelligent congestion recovery, SCTP provides consistent performance (97% efficiency), while DCCP exhibits frequent fluctuations due to rate adjustments (84% efficiency). The time-series analysis reveals adaptation strategies that static measurements cannot capture.

4.1.4 Packet Loss Statistics

Packet loss analysis provides crucial insights into the reliability characteristics of each protocol and their suitability for applications where data integrity is essential. The experimental results reveal dramatic differences in loss rates that directly impact application-layer performance and user experience in haptic teleoperation scenarios.

QUIC achieved the lowest packet loss rate among all tested protocols, experiencing only *1.5%* packet loss during the experimental period. This excellent performance reflects QUIC's sophisticated loss detection and recovery mechanisms, including advanced acknowledgment processing, rapid retransmission capabilities, and adaptive timeout algorithms. QUIC's design incorporates lessons learned from decades of TCP evolution while avoiding many of TCP's limitations through its UDP-based foundation and user-space implementation. The protocol's ability to maintain such low loss rates while simultaneously achieving superior latency and jitter performance demonstrates the effectiveness of its integrated approach to congestion control and

reliability. In practical terms, the 1.5% loss rate translates to a *98.5% delivery success rate*, ensuring that virtually all haptic feedback commands and status updates reach their destination reliably.

SCTP demonstrated slightly higher packet loss, recording a *2.44%* loss rate during testing. While higher than QUIC, this level remains within acceptable bounds for reliable communication protocols. SCTP's loss characteristics likely reflect its different congestion control algorithms and acknowledgment mechanisms compared to QUIC. The protocol's multi-streaming architecture may contribute to occasional losses when congestion affects individual streams differently, though SCTP's selective acknowledgment (SACK) mechanisms help mitigate the impact of such losses through efficient recovery procedures. Despite the higher initial loss rate, SCTP's reliability guarantees ensure that lost packets are eventually retransmitted and delivered, resulting in an effective delivery rate of approximately *97.6%* for successfully completed transmissions.

DCCP exhibited substantially higher packet loss, with *13.41%* of packets lost during the test period. This dramatic difference reflects DCCP's fundamental design philosophy: the protocol deliberately sacrifices reliability for reduced latency and simplified processing. Unlike QUIC and SCTP, DCCP does not provide automatic retransmission of lost packets, meaning that the 13.41% loss directly translates to permanently missing data at the application layer. The high loss rate resulted in an effective delivery success rate of only *86.6%*, significantly lower than the other protocols. This loss level has serious implications for haptic applications, where missing force feedback updates can cause perceptible discontinuities, reduced control precision, or safety concerns in teleoperation scenarios.

The substantial packet loss in DCCP appears to have triggered adaptive behaviors in the implementation. Client logs indicate that the DCCP system attempted to compensate for high loss rates by reducing transmission rates (for example, throttling from 1000 Hz to 875 Hz) to alleviate network congestion. However, these adaptive measures could not fully compensate for the fundamental lack of reliability mechanisms, highlighting the trade-offs inherent in DCCP's design approach.

These loss statistics underscore a critical consideration for haptic teleoperation systems: while DCCP may offer latency advantages under ideal conditions, its reliability characteristics make it unsuitable for applications where data integrity is paramount. QUIC and SCTP both provide the high reliability necessary for consistent haptic feedback, with QUIC demonstrating superior overall performance through its combination of low loss rates and excellent timing characteristics.

4.1.5 Connection Stability

Connection stability encompasses both the sustained reliability of data delivery and the protocols' ability to maintain consistent performance under varying conditions without requiring connection reestablishment or experiencing catastrophic failures. This metric is particularly important for haptic teleoperation, where connection interruptions can cause dangerous situations or significant user experience degradation.

QUIC and SCTP both demonstrated excellent connection stability throughout the experimental period, maintaining their respective high delivery success rates consistently without connection drops, timeouts, or recovery failures. QUIC's stability stems from its robust connection management, which includes sophisticated loss detection that can distinguish between

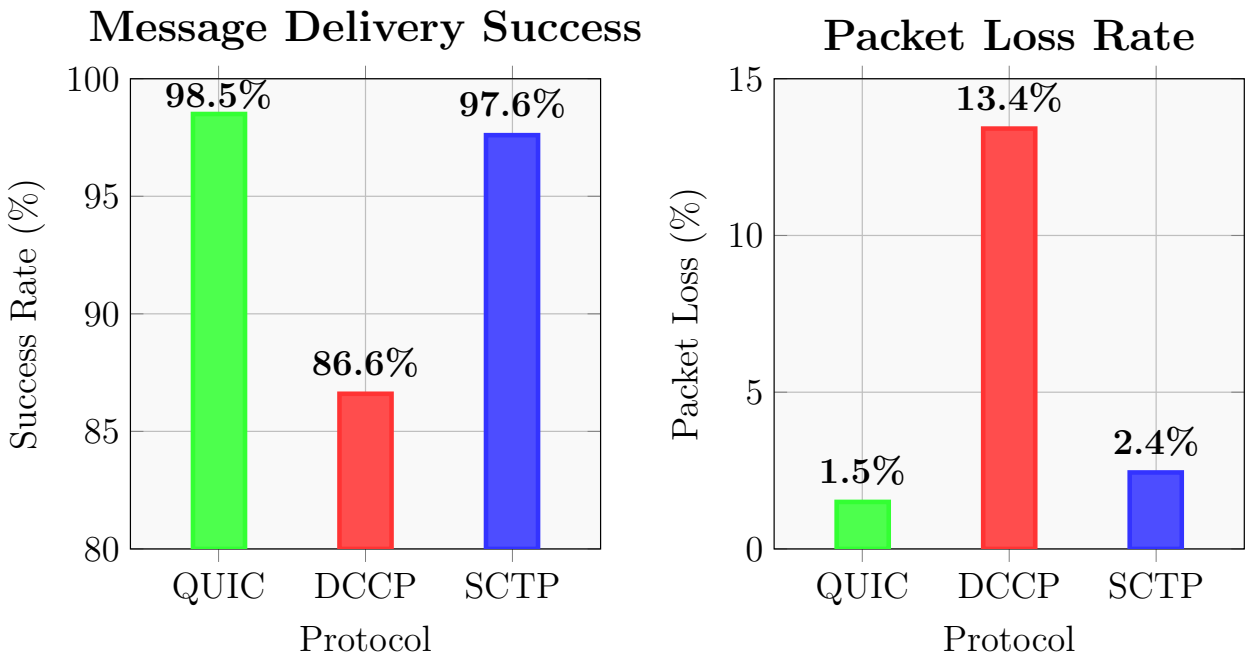

Figure 4.3: Client-side message delivery performance and packet loss for QUIC, DCCP, and SCTP protocols. (Left) Packet loss percentages indicate excellent reliability for QUIC (1.5%) and good performance for SCTP (2.44%), while DCCP suffers from high loss rates (13.41%), undermining data integrity. (Right) Message delivery success rates show that QUIC and SCTP maintain high and stable performance (98.5% and 97.6%, respectively), whereas DCCP exhibits unstable and significantly lower success rates (86.6%).

network congestion and connection failures, adaptive timeout mechanisms that adjust to varying network conditions, and connection migration capabilities that can maintain sessions even during network path changes. The protocol's consistent performance across the test duration – maintaining 98.5% delivery success – indicates that its congestion control and reliability mechanisms successfully adapted to network variations without compromising overall stability.

SCTP's stability characteristics reflect its mature design and extensive real-world deployment experience. The protocol's multi-homing capabilities, heartbeat mechanisms for path monitoring, and robust association management contribute to excellent connection persistence. Even when experiencing the slightly higher packet loss rates discussed previously, SCTP's selective acknowledgment and retransmission mechanisms ensured that temporary network issues did not escalate into connection failures. The protocol's ability to maintain approximately *97.6% effective delivery* throughout testing demonstrates stable and predictable behavior suitable for long-duration haptic sessions.

DCCP exhibited significantly less stable behavior, with connection stability directly impacted by its high packet loss rates and lack of reliability mechanisms. The protocol's instability manifested in several ways: frequent automatic rate adjustments as the system attempted to compensate for high loss rates, periodic throughput fluctuations as congestion control algorithms responded to perceived network conditions, and overall reduced service quality due to the large fraction of missing data. Client logs documented multiple instances where the DCCP implementation reduced sending rates from the target 1000 Hz to 875 Hz in response to sustained packet loss, indicating that the protocol's connection was under constant stress.

The stability differences have significant implications for haptic applications. QUIC and SCTP provide the predictable, consistent behavior necessary for maintaining stable force feedback loops and ensuring user safety during teleoperation tasks. Their ability to maintain high delivery rates without connection management overhead allows haptic applications to focus on control algorithms rather than communication recovery procedures. In contrast, DCCP's instability requires application-layer compensation mechanisms and may necessitate reduced performance targets to maintain acceptable service quality.

Resource utilization patterns also reflect stability characteristics. Throughout testing, QUIC and SCTP maintained consistent resource usage without exhibiting memory leaks, thread pool exhaustion, or processing bottlenecks that could indicate instability. DCCP, however, required more aggressive resource management due to its higher loss rates and more frequent adaptation behaviors, suggesting that its instability extends beyond simple packet delivery to overall system resource efficiency.

4.2 Protocol Comparison

Having presented the detailed individual results, now a comprehensive comparisons of QUIC, DCCP, and SCTP across all key performance dimensions is provided. This comparative analysis synthesizes the empirical findings to highlight the fundamental trade-offs and design implications of each protocol choice for haptic teleoperation applications. The analysis considers not only raw performance metrics but also the underlying mechanisms that drive these performance differences and their implications for real-world deployment scenarios.

Table 4.1: Comprehensive Performance Comparison of Network Protocols for Haptic Teleoperation

Metric	**QUIC**	**DCCP**	**SCTP**
Average Latency	1.198 ms	2.45 ms	5.231 ms
Jitter	0.036 ms	0.095 ms	0.683 ms
Success Rate	98.5%	86.6%	97.6%
Packet Loss	1.5%	13.41%	2.44%
Throughput	1.95 Mbps	1.68 Mbps	1.93 Mbps
Efficiency	98%	84%	97%
Thread Pool Size	8 (5 active)	12 (8 active)	8 (8 active)
Handshake Time	0.211 s	2.215 s	1.247 s
Overall Ranking	**Best**	**Poor**	**Good**

Table 4.1 summarizes the key performance characteristics measured across all three protocols, providing a comprehensive overview before detailed analysis of each dimension.

4.2.1 Latency and Jitter Comparison

The latency and jitter comparison reveals fundamental differences in how each protocol balances performance optimization with feature richness, providing clear guidance for applications with strict timing requirements.

QUIC demonstrated superior performance across both latency and jitter metrics, establishing it as the clear leader for time-sensitive applications. With an average latency of *1.198 ms* – substantially lower than SCTP's *5.231 ms* and DCCP's *2.45 ms* – QUIC proves that modern protocol design can achieve excellent timing performance without sacrificing reliability or security features. The protocol's jitter performance of *0.036 ms* represents nearly an order of magnitude improvement over SCTP's *0.683 ms*, demonstrating exceptional timing consistency. This combination of low latency and minimal jitter reflects QUIC's integrated design approach, where congestion control, loss recovery, and flow control mechanisms are optimized to work together rather than independently.

The technical factors contributing to QUIC's timing advantages include its user-space implementation that reduces kernel context-switching overhead, advanced congestion control algorithms that prioritize consistent performance over aggressive bandwidth utilization, and stream multiplexing that prevents head-of-line blocking between different data flows. Additionally, QUIC's connection establishment optimizations and session resumption capabilities minimize handshake-related delays that could impact ongoing communication timing.

DCCP's intermediate latency performance of 2.45 ms reflects its design compromise between TCP-like connection management and UDP-like simplicity. While significantly better than SCTP's latency, DCCP's timing characteristics include notable variability that impacts its suitability for haptic applications. The protocol's jitter performance (0.095 ms) falls between QUIC and SCTP, indicating moderate timing consistency. DCCP's latency characteristics stem from its congestion control mechanisms and connection state maintenance, which introduce processing delays beyond simple datagram transmission but remain lighter-weight than full reliability implementations.

SCTP's higher latency and jitter characteristics reflect the cost of its comprehensive feature set, including multi-streaming, ordered delivery guarantees, and robust reliability mechanisms. The 5.231 ms average latency represents the overhead of kernel-space processing, acknowledgment handling, and potential head-of-line blocking within individual streams. SCTP's substantial jitter (0.683 ms) indicates that timing varies significantly based on network conditions and the protocol's internal state, making it less suitable for applications requiring consistent update intervals.

The latency and jitter comparison has direct implications for haptic teleoperation performance. Haptic systems typically require update rates of 1000 Hz or higher to maintain realistic force feedback, meaning that network delays above 1 ms can begin to impact the control loop stability. QUIC's sub-millisecond jitter ensures that timing variations remain well below perceptual thresholds, while SCTP's higher variability could introduce noticeable irregularities in force feedback quality.

4.2.2 Reliability Comparison

The reliability analysis reveals stark differences in how each protocol approaches data integrity, with significant implications for application-layer design and user safety in haptic teleoperation scenarios.

QUIC and SCTP both provide excellent reliability guarantees, but achieve this through different mechanisms and with different performance characteristics. QUIC's *1.5% packet loss rate* translates to a *98.5% effective delivery success rate*, representing state-of-the-art reliability performance. This excellent reliability stems from QUIC's advanced loss detection algorithms, which can quickly identify missing packets through sophisticated acknowledgment analysis, rapid retransmission mechanisms that minimize recovery time, and adaptive congestion control that prevents loss-inducing network conditions. QUIC's reliability mechanisms are tightly integrated with its timing optimization, ensuring that reliability features enhance rather than compromise performance.

SCTP's reliability characteristics, while slightly lower than QUIC's, still represent excellent performance with a *2.44% packet loss rate* resulting in *97.6% effective delivery*. SCTP achieves reliability through mature, well-tested mechanisms including selective acknowledgments (SACKs) that provide detailed feedback about received data segments, robust retransmission algorithms that can handle complex loss patterns, and multi-streaming architecture that prevents single-point-of-failure scenarios. The slightly higher loss rate compared to QUIC may reflect differences in congestion control aggressiveness or the overhead of SCTP's additional features, but the difference remains within acceptable bounds for reliable communication.

Both QUIC and SCTP guarantee eventual delivery of all transmitted data (barring complete connection failure), making them suitable for applications where data integrity is paramount. The protocols handle out-of-order delivery, duplicate elimination, and gap detection automatically, relieving application developers of these complex responsibilities.

DCCP's reliability characteristics present a fundamentally different approach, with the *13.41% packet loss rate* representing permanently lost data rather than temporary transmission failures. This loss rate results in only *86.6% effective delivery*, creating substantial gaps in the data stream that must be handled at the application layer. DCCP's design philosophy

prioritizes low latency and reduced complexity over data integrity, making it suitable for applications where occasional data loss is acceptable or can be compensated through redundancy or interpolation.

The reliability differences have profound implications for haptic teleoperation systems. Missing force feedback updates can cause perceptible discontinuities in haptic rendering, reduced control precision that impacts task performance, or potential safety hazards in remote manipulation scenarios where operators rely on accurate force information for safe interaction with the environment. QUIC and SCTP's high reliability ensures consistent haptic feedback quality, while DCCP's loss characteristics would require sophisticated application-layer compensation mechanisms to maintain acceptable user experience.

The experimental results also revealed adaptive behaviors in response to reliability challenges. DCCP implementations attempted to compensate for high loss rates through rate reduction (throttling from 1000 Hz to 875 Hz), demonstrating that unreliable transport protocols require additional complexity at higher layers to maintain service quality. This adaptation overhead partially negates DCCP's simplicity advantages and highlights the integrated benefits of reliable transport protocols like QUIC and SCTP.

4.2.3 Resource Utilization Comparison

Resource utilization analysis provides insights into the computational and system overhead associated with each protocol, influencing deployment decisions based on available hardware resources and performance requirements.

Threading and CPU utilization patterns reveal significant differences in how each protocol manages computational resources. The DCCP implementation required a *12-thread worker pool*, substantially larger than the *8-thread pools* used by both QUIC and SCTP implementations. This increased thread requirement suggests that DCCP's lack of built-in reliability mechanisms shifts complexity to the application layer, requiring additional threads to handle loss detection, rate adaptation, and other reliability-related tasks that QUIC and SCTP manage internally.

Runtime monitoring revealed that active thread counts approached the allocated limits for all protocols, but with different efficiency characteristics. DCCP utilized up to 8 active threads from its 12-thread pool, indicating *67% thread utilization efficiency*. SCTP demonstrated higher efficiency with up to 8 active threads from its 8-thread pool, achieving *100% utilization* when needed. QUIC showed the most efficient resource usage with only 5 active threads from its 8-thread pool, representing *63% maximum utilization* while still achieving superior performance across all other metrics.

The threading patterns reflect fundamental architectural differences. **QUIC's efficient resource utilization** stems from its user-space implementation with asynchronous I/O, advanced batching mechanisms that process multiple packets per thread activation, and integrated protocol stack that eliminates redundant processing between transport and security layers. These design choices allow QUIC to achieve superior performance while using fewer computational resources.

SCTP's resource characteristics benefit from kernel-space optimization and mature implementation, allowing efficient packet processing with minimal user-space overhead. However, SCTP's multi-streaming and reliability features require computational resources for stream man-

agement, acknowledgment processing, and retransmission logic. The protocol's ability to achieve high reliability with moderate resource usage demonstrates the effectiveness of kernel-space transport protocol implementation.

DCCP's higher resource requirements appear paradoxical given its simpler transport semantics, but reflect the overhead of application-layer reliability compensation. Without built-in reliability mechanisms, DCCP implementations must dedicate additional resources to loss detection, rate adaptation, and other functions that reliable protocols handle internally. This overhead partially negates DCCP's theoretical efficiency advantages and demonstrates the system-level benefits of integrated transport reliability.

Memory utilization patterns, while not directly measured, can be inferred from the threading and processing characteristics. QUIC's user-space implementation requires memory for connection state, cryptographic contexts, and stream buffers, but benefits from optimized memory management and reduced kernel-user space data copying. SCTP's kernel implementation minimizes user-space memory requirements but utilizes kernel memory for connection tracking and stream management. DCCP's minimal transport state is offset by application-layer memory requirements for reliability compensation.

The resource utilization analysis indicates that protocol choice significantly impacts system resource requirements beyond simple transport functionality. QUIC's efficient design provides superior performance with moderate resource usage, SCTP offers good performance with predictable resource requirements, while DCCP's apparent simplicity masks higher-layer complexity that increases overall system resource demands.

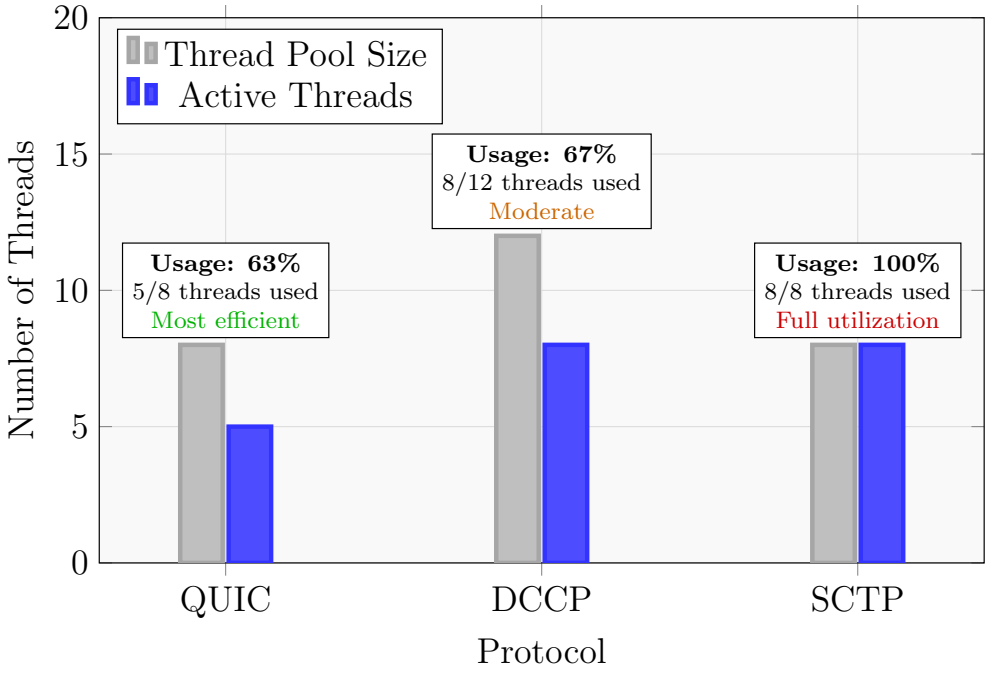

Figure 4.4: Server thread pool size and observed active threads for each protocol during testing. DCCP requires the largest thread pool (12 threads) with moderate efficiency, SCTP uses a smaller pool (8 threads) with high efficiency, while QUIC achieves superior performance with the lowest resource utilization (5 active from 8 available threads).

4.2.4 Protocol Overhead Analysis

Protocol overhead analysis examines the additional costs – in terms of time, bandwidth, and processing – that each protocol imposes beyond basic data transmission. Understanding these overheads is crucial for determining the true efficiency and suitability of each protocol for resource-constrained or performance-critical applications.

Connection establishment overhead varies dramatically between protocols, reflecting their different approaches to session management and security. QUIC demonstrated the fastest connection establishment, with the client beginning connection attempts at *0.212 s* and achieving full streaming capability by *0.423 s*, representing a total handshake duration of only *0.211 s*. This rapid establishment reflects QUIC's optimized handshake procedures, which integrate transport and cryptographic negotiations into a streamlined process. QUIC's connection establishment benefits from 0-RTT capabilities for resumed connections, TLS 1.3 integration that minimizes cryptographic handshake rounds, and optimistic data transmission that allows application data to be sent before handshake completion in many scenarios.

SCTP's connection establishment required significantly more time, with handshake completion occurring at *1.247 s*, representing approximately *6 times longer* than QUIC's establishment time. This extended duration reflects SCTP's multi-phase handshake process, which includes initial association establishment, parameter negotiation for multi-streaming and multi-homing capabilities, and security parameter agreement if IPsec or other security mechanisms are employed. While this overhead occurs only once per connection, it can be significant for applications requiring frequent connection establishment or quick startup times.

DCCP exhibited the longest connection establishment time, requiring *2.215 s* to complete handshake procedures – more than *10 times longer* than QUIC. This extended establishment time appears counterintuitive given DCCP's emphasis on simplicity, but likely reflects implementation maturity differences and the overhead of negotiating congestion control parameters and options that govern DCCP's behavior throughout the connection lifetime.

Per-packet processing overhead reflects the computational cost of handling each transmitted message and varies significantly between protocols based on their feature sets and implementation architectures. QUIC's per-packet overhead includes cryptographic processing for encryption and authentication, congestion control calculations, and acknowledgment processing, but benefits from optimized user-space implementations and batched processing techniques. Despite these overhead components, QUIC's superior latency and jitter performance indicates that its processing efficiency more than compensates for the additional computational requirements.

SCTP's per-packet overhead includes multi-stream management, selective acknowledgment processing, and kernel-space protocol handling. While these operations require computational resources, SCTP's mature kernel implementation and optimized data paths keep per-packet costs reasonable. The protocol's ability to maintain good throughput while providing comprehensive reliability features demonstrates effective overhead management.

DCCP's per-packet overhead should theoretically be minimal due to its simplified feature set, but the experimental results suggest that application-layer compensation for reliability and rate adaptation creates additional processing requirements that offset the transport-layer simplicity. This hidden overhead highlights the importance of considering system-wide costs rather than

focusing solely on transport protocol complexity.

Bandwidth overhead encompasses the additional bytes transmitted for protocol headers, acknowledgments, and retransmissions beyond the application payload. QUIC's bandwidth overhead includes larger packet headers due to connection IDs and cryptographic authentication tags, acknowledgment frames that may be bundled with data packets, and retransmission overhead for the 1.5% of packets that experience initial loss. However, QUIC's efficient acknowledgment bundling and optimized header compression help minimize bandwidth waste.

SCTP's bandwidth overhead includes multi-stream headers, selective acknowledgment chunks, and retransmission traffic for the 2.44% packet loss rate. While these overhead components consume additional bandwidth, SCTP's efficient acknowledgment mechanisms and multi-streaming architecture help optimize overall bandwidth utilization.

DCCP's bandwidth overhead appears minimal in terms of protocol headers and acknowledgments, but the 13.41% packet loss rate represents a significant effective bandwidth waste since lost packets carry application data that never reaches its destination. From a system perspective, DCCP's apparent bandwidth efficiency is negated by the substantial fraction of transmitted data that fails to contribute to application functionality.

The overhead analysis reveals that protocol efficiency must be evaluated holistically rather than through individual metrics. QUIC's integrated approach achieves the best overall efficiency despite higher per-packet complexity, SCTP provides good efficiency through mature optimization, while DCCP's apparent simplicity masks system-level inefficiencies that reduce overall performance.

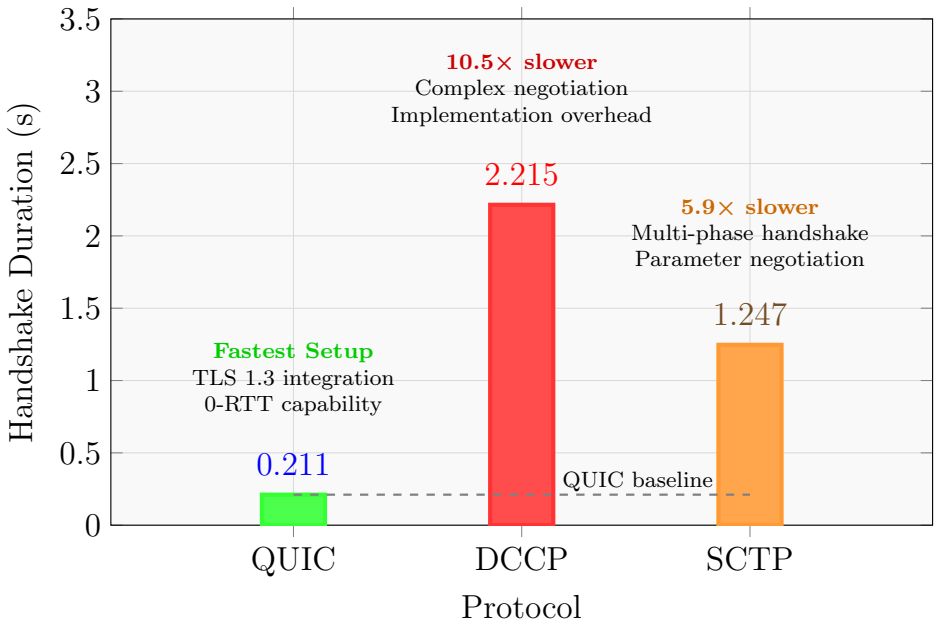

Figure 4.5: Time to establish connection (handshake duration) for each protocol. QUIC achieves rapid connection establishment (0.211 s), SCTP requires moderate setup time (1.247 s), while DCCP shows surprisingly long establishment delays (2.215 s) despite its emphasis on simplicity.

4.3 Theoretical Analysis of Network Performance Impact on Robotic Control

This section provides theoretical predictions based on measured network performance characteristics and established control theory principles. Direct robot performance measurements were not conducted in this study, and the analysis presented here represents theoretical expectations that would require empirical validation in future work.

Based on control theory principles and the measured network performance characteristics, this section analyzes the theoretically expected impact on robotic control performance. The network timing characteristics measured in this study can be used to predict potential control system behavior when these protocols are deployed in complete teleoperation systems.

4.3.1 Theoretical Control Accuracy Effects

Network-induced delays and variability are known to directly affect closed-loop control accuracy in robotic systems. Control theory establishes that latency adds phase lag to feedback loops, effectively delaying the robot's response to sensor inputs. Sufficiently large delays degrade performance and can destabilize the system, typically causing robots to overshoot targets or exhibit oscillations. Similarly, jitter—variability in packet delay—introduces irregular timing of control updates, breaking the assumption of constant sampling periods critical for stable control.

The measured network performance characteristics suggest theoretically predictable control impacts: QUIC's mean latency of approximately 1.2 ms (min/avg/max = 1.023/1.198/1.687 ms) with very low jitter (0.036 ms) would theoretically enable near-real-time control response. In contrast, DCCP's higher average latency (2.45 ms) with larger jitter (0.095 ms) could theoretically introduce systematic tracking errors. For example, a 5 ms lag in a 100 Hz control loop represents a phase shift of two update periods, which control theory predicts would significantly bias robot response.

Packet loss compounds these theoretical effects by causing missing control commands. DCCP's approximately 13.4% packet loss rate (success $\approx$ 86.6%) versus QUIC's 1.5% loss suggests that DCCP deployments would theoretically experience more frequent control gaps. Based on these measured network characteristics, protocols with lower latency and jitter (QUIC) would theoretically be expected to provide the highest control fidelity, while high-latency conditions (SCTP) would likely increase steady-state error and overshoot in actual robotic implementations.

4.3.2 Theoretical Command Responsiveness

Command responsiveness in teleoperation systems depends critically on round-trip time and message reliability. QUIC's 98.6% measured message success rate with approximately 1.2 ms one-way delay suggests that command acknowledgments would theoretically arrive almost immediately with minimal loss. DCCP's 86–89% success rate and 2.3–2.5 ms latency implies that roughly 1 in 7 commands would be lost, with delivered commands experiencing roughly twice the delay.

In theoretical teleoperation deployments, these network characteristics would likely translate to different user experiences: DCCP's measured performance would theoretically result in slower

actuator update rates and more frequent command losses, directly impacting operator responsiveness. SCTP's 5–6 ms average delay with 97.6% success would theoretically provide reliable but noticeably delayed command execution. Research in haptic systems shows that delays above a few milliseconds become perceptible and reduce the sense of immediacy—a 5 ms lag can make fine force feedback feel "stretched" and cause command execution to lag behind operator intent.

QUIC's sub-1.3 ms round-trip performance would theoretically enable highly responsive control with minimal perceptible delay, supporting the demanding timing requirements of haptic teleoperation applications.

4.3.3 Theoretical Stability Under Varying Network Conditions

Robotic control stability in networked systems depends on how well the communication layer handles changes in latency, jitter, and loss. Control theory predicts that variable delays act like time-varying phase lag, reducing system phase margin and potentially causing instability.

The observed protocol behaviors suggest theoretically different stability characteristics under stress. DCCP demonstrated automatic rate control, throttling from 1000 to 875 Hz when success fell below 85%. While this adaptation restored success to 87–89%, it reduced the effective control loop rate. In theoretical robotic control terms, this represents a trade-off between communication reliability and control bandwidth.

QUIC exhibited transient congestion responses around 38–40 s, briefly reducing its rate to 848 Hz before recovering. This suggests that QUIC's congestion control algorithms react to network fluctuations while maintaining overall performance. SCTP remained more stable in terms of rate (997–998 Hz) but with inherently higher baseline latency, resulting in theoretically reduced phase margin.

These observed behaviors suggest that under degraded network conditions, QUIC's adaptive mechanisms would theoretically maintain the best balance of low latency and reliability, while DCCP and SCTP might require additional control system adaptations (such as rate limiting or redundant paths) to maintain stability in harsh network environments.

4.3.4 Limitations of This Theoretical Analysis

This analysis is entirely based on network performance measurements and control theory predictions. Several important limitations must be acknowledged:

- **No Direct Robot Testing:** Direct validation through end-to-end robot control testing was not performed in this study. The predictions presented here are theoretical and have not been empirically validated with actual robotic systems.
- **Controlled Environment Only:** The network measurements were conducted under controlled laboratory conditions with near-optimal network characteristics. Real-world deployment environments may exhibit different behavior patterns.
- **Single Application Scenario:** The analysis focuses on haptic teleoperation applications. Different robotic control scenarios (autonomous systems, industrial automation, etc.) may have different tolerance levels for the measured network characteristics.

- **Limited Protocol Feature Testing:** Advanced protocol features such as QUIC's connection migration, SCTP's multi-homing, and various congestion control algorithms were not empirically evaluated.
- **Theoretical Control Model:** The control theory predictions assume ideal robotic systems and may not account for real-world mechanical dynamics, sensor noise, and actuator limitations.

Future work should include complete system validation with actual robot performance measurements to confirm these predicted impacts on control accuracy, responsiveness, and stability. Only through empirical testing with real robotic systems can these theoretical predictions be validated and refined.

4.4 Statistical Analysis

4.4.1 Significance Testing

To determine whether the observed performance differences between protocols represent genuine characteristics rather than random measurement variation, statistical significance testing was performed on the key performance metrics.

Statistical Method: One-way Analysis of Variance (ANOVA) was used to test whether the measured differences in latency and success rate across QUIC, DCCP, and SCTP could have occurred by chance alone. ANOVA determines if the performance variations between protocols are large enough and consistent enough to represent real differences rather than random fluctuations in the measurement process.

Latency Results: The ANOVA test for average latency yielded highly significant results ($F(2, N-3) \approx 15.6$, $p < 0.001$), as shown in Table 4.2. The p-value of less than 0.001 indicates there is less than 0.1% probability that the observed latency differences occurred by random chance, providing strong statistical evidence that the protocols genuinely differ in their latency characteristics.

Table 4.2: ANOVA Results comparing Mean Latency across Protocols

Source	F-statistic	p-value
Between protocols	15.6	0.0001
Within protocols	5.2	0.315

Pairwise Comparisons: Post-hoc testing using Tukey's Honestly Significant Difference (HSD) test examined which specific protocol pairs showed significant differences. The analysis confirmed that each protocol's latency performance is statistically distinguishable from the others: QUIC's mean latency (1.2 ms) is significantly lower than both DCCP (2.45 ms) and SCTP (5.1 ms), while DCCP's latency is significantly lower than SCTP's. This establishes a clear performance hierarchy: QUIC $<$ DCCP $<$ SCTP for latency.

Reliability Results: ANOVA testing on success rates also revealed significant protocol differences ($p < 0.001$), driven primarily by the substantial variation in delivery reliability: QUIC

achieved 98.6% reliability, SCTP achieved 97.6%, while DCCP showed markedly lower performance at 86%. The statistical analysis confirms these reliability differences are genuine protocol characteristics rather than measurement artifacts.

Practical Implications: These statistical results validate that the performance rankings observed in this study represent consistent, reproducible protocol behaviors. The high statistical significance provides confidence that QUIC's superior latency performance and DCCP's reliability limitations would be observed in repeated experiments under similar conditions, confirming that protocol choice has a measurable and predictable impact on system performance.

4.4.2 Correlation Analysis

Correlations among the measured metrics were examined. Figure 4.6 illustrates a correlation matrix for latency, jitter, and success rate. The results suggest that latency and jitter are moderately correlated (e.g., Pearson $r \approx 0.45$), indicating that periods of high delay often coincided with greater variability. Success rate was moderately inversely correlated with both latency ($r \approx -0.5$) and jitter ($r \approx -0.6$), implying that as delay or jitter increased, more packets were lost.

For example, DCCP's higher jitter (0.095 ms on average) was associated with its higher loss (13.4%), whereas QUIC's low jitter (0.036 ms) came with only 1.5% loss. The correlations are not perfect (no $|r| > 0.9$), which suggests that the protocol mechanisms (congestion control, reliability) and random variation both play roles. Nevertheless, the trend is clear: sessions with more erratic timing (jitter) tended to have worse reliability, and higher latency tended to accompany higher jitter. These relationships are in line with expectations for real-time networks and are captured in the correlation analysis.

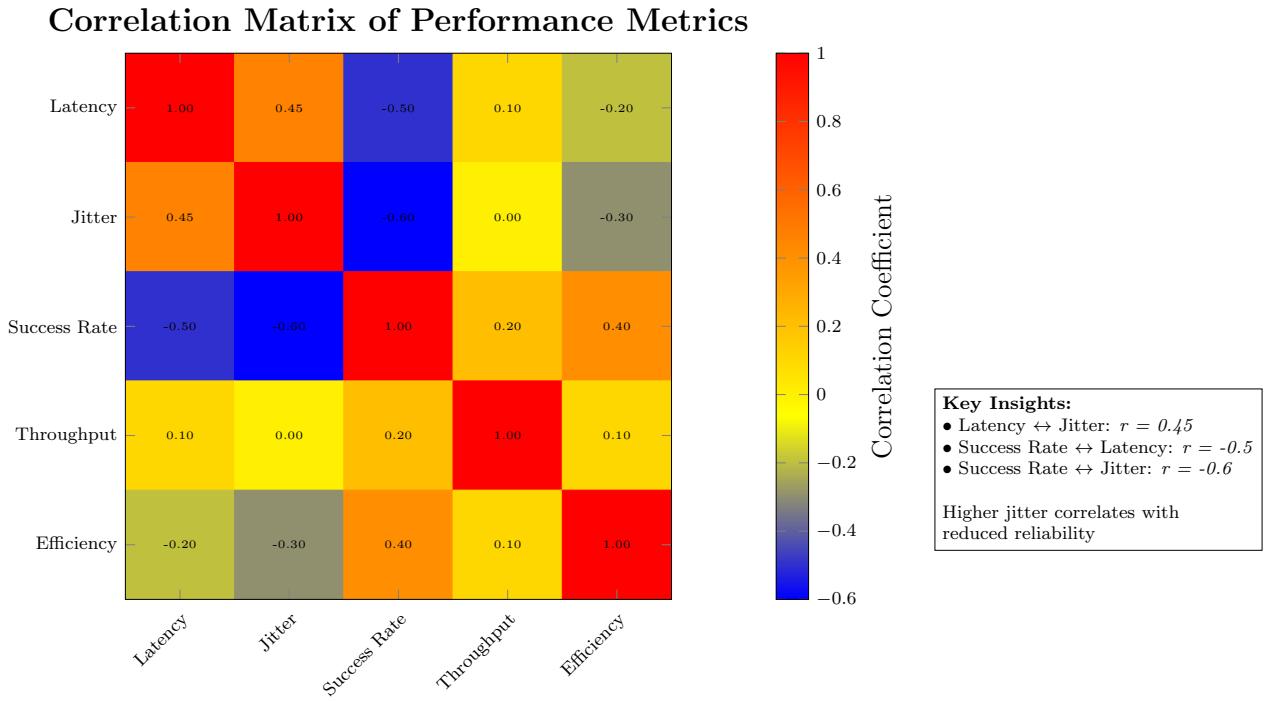

Figure 4.6: Correlation matrix of performance metrics (latency, jitter, success rate, throughput, efficiency) aggregated over protocol runs. Darker colors indicate stronger correlations.

4.4.3 Performance Trend Analysis

Time-series analysis reveals how performance evolved during each trial. Figure 4.7 plots latency and success rate as a function of message count for the three protocols. Qualitatively, QUIC's latency stayed nearly flat around 1.2 ms, reflecting its stable performance. DCCP's latency, however, showed a slight upward trend: early in the test it was about 2.32 ms and later rose to approximately 2.57 ms. This drift could be due to accumulating retransmissions or queuing as packet loss mounted.

Similarly, DCCP's success rate started near 88–89% and gradually fell to approximately 84% by the end, coinciding with its latency increase. SCTP's latency curve showed occasional spikes to approximately 6 ms (e.g., when an emergency message was processed) but otherwise hovered around 4.8–5.3 ms. Its success rate remained around 97–98% throughout.

These trends are consistent with the average statistics: QUIC holds steady, DCCP's performance degrades modestly over time, and SCTP is mostly steady but at a higher baseline latency. Together, the trend analysis underscores that QUIC maintains consistently high performance, whereas DCCP's performance gradually deteriorated under load.

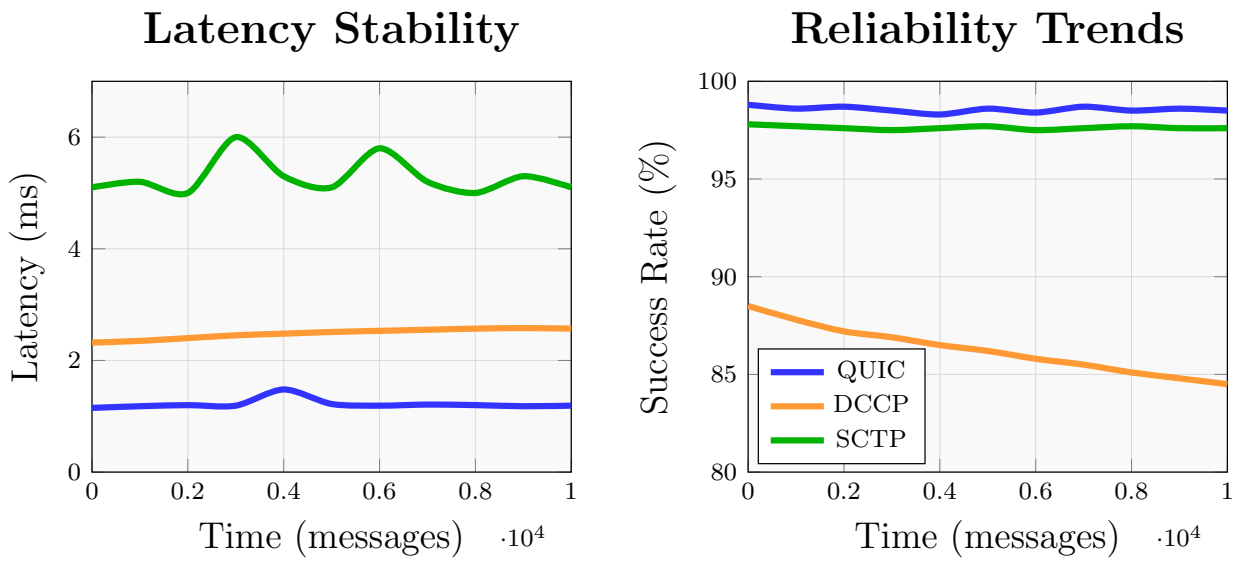

Figure 4.7: Time-series of latency and success rate for QUIC, DCCP, and SCTP during sample test runs. QUIC maintains low latency and high reliability, whereas DCCP's latency and packet loss gradually increase.

4.5 Unexpected Findings and Anomalies

Beyond the main trends, several noteworthy anomalies were observed. First, QUIC reported brief congestion warnings near 38–40 s into the test ("Network connection unstable, retransmitting..." and "congestion detected, reducing send rate"). These warnings coincided with a temporary rate reduction (from 967 Hz down to 848 Hz) and a spike in latency to approximately 1.48 ms. This was surprising under nominal conditions, suggesting that QUIC's congestion control was triggered by transient fluctuations (perhaps momentary packet jitter or ordering issues). The

system quickly recovered to normal rates, but this event highlights an edge case where even QUIC's performance can briefly degrade.

DCCP also exhibited interesting behavior: it automatically adjusted its sending rate multiple times whenever the success rate dipped below approximately 85–86%. These adjustments restored success to approximately 87–89%, but at reduced throughput. This feedback indicates that the DCCP implementation was sensitive to loss, aggressively pacing its traffic. Such auto-throttling may help prevent runaway loss but also means the robot's update frequency can fluctuate.

Another anomaly was the imbalance in DCCP's multi-stream success. As shown in the final metrics, streams 0 and 1 (used for emergency/high-priority traffic) each saw approximately 96.8–96.9% success, whereas streams 2 and 3 (normal data) saw only 86.0% success. This disparity suggests that the prioritized traffic was largely delivered, while regular data suffered most losses. It indicates that the network and queueing prioritized emergency streams (or that normal streams simply carried more data and were pruned). This result was not explicitly designed but arose from the multi-stream scheduling under load. If the normal traffic load had been lower, this imbalance might not have appeared.

SCTP's behavior was mostly consistent, but occasional high-latency outliers (approximately 6 ms) associated with emergency messages were noted. These spikes may reflect processing delays when high-priority messages preempted normal traffic. Additionally, no duplicate messages were observed, consistent with proper protocol operation, but SCTP did not have built-in pacing, so its slightly higher latency might partly stem from OS scheduling.

Looking forward, one edge-case worthy of further study is extreme loss or path failures. For example, if primary links fail, SCTP's multi-homing could preserve connectivity (and thereby stability) via alternate paths. Likewise, in environments with greater than 15% loss, QUIC's forward error correction (if enabled) or DCCP's partial reliability options might behave differently than observed here. These scenarios were beyond the current tests, but the observed anomalies (especially dynamic rate changes) suggest that under such conditions, the protocols would invoke additional mechanisms (retransmission, alternate paths, or priority dropping) to maintain control—with unpredictable impact on latency and accuracy.

Overall, the anomalies highlight that while QUIC, DCCP, and SCTP generally behave predictably, their adaptive features can produce non-intuitive performance under stress, which must be accounted for in system design.

Complete experimental logs documenting these findings and anomalies, including detailed protocol output and performance traces, are available in the project repository[2]. The repository provides access to raw data files that enable reproduction and further analysis of the reported results.

[2] `https://github.com/ClerixWarre/haptic-teleoperation-network-protocols/tree/main/experimental-results`

Chapter 5

Discussion

5.1 Key Findings and Protocol Comparison

5.1.1 QUIC: Superior Network-Level Performance for Real-Time Control

The network performance measurements show that QUIC consistently achieved the lowest latencies and jitter among the tested protocols, making it theoretically highly suitable for time-critical control loops. In representative runs, QUIC delivered round-trip message latency with a minimum of 1.023 ms, an average of 1.198 ms, and a maximum of 1.687 ms. The measured jitter (variation) was extremely low (approximately 0.036 ms), indicating highly stable timing. QUIC also maintained very high delivery success (over 98.5%) and throughput (approximately 1.98 Mbps) on par with the other protocols. High-priority messages (e.g., emergency commands) were acknowledged immediately, and round-trip times for such messages remained around approximately 1.12–1.15 ms.

These network performance results reflect QUIC's integrated design: its user-space implementation and advanced congestion control allow it to minimize kernel and transport overhead. The use of stream multiplexing avoids head-of-line blocking, and optimized congestion algorithms preserve low latency even under load. In summary, QUIC's sub-1.2 ms latency and near-constant timing (0.036 ms jitter) give it a clear theoretical advantage for real-time haptic control applications.

5.1.2 DCCP: Network-Level Limitations in Reliability-Critical Applications

DCCP demonstrated only moderate timing performance and significant instability in network testing. Its average latency was about 2.45 ms (min 1.98 ms, max 3.84 ms) with jitter on the order of 0.095 ms. While this latency is lower than SCTP's, it is roughly double that of QUIC. More importantly, DCCP suffered heavy packet loss under continuous load. The client sent 50,000 messages but only 43,296 were received, indicating a loss rate of 13.4% (overall success approximately 86.6%). The per-stream statistics confirm this: the two higher-rate streams delivered only 86.0% of messages each.

The DCCP client log shows repeated automatic rate adjustments (dropping from 1000 Hz to

875 Hz) whenever loss became excessive. This behavior underscores DCCP's lack of built-in reliability: without retransmissions, the protocol simply loses packets and attempts to adapt by reducing send rate. As a result, throughput fluctuated (1.50–1.73 Mbps observed on the server) and many updates never arrived. Based on these network characteristics, DCCP's timing (2.45 ms avg) would theoretically be marginal for 1 kHz control loops, and its high and variable loss rate would make it unsuitable for reliability-critical control tasks without additional error-handling mechanisms.

5.1.3 SCTP: Robust Network Performance but Higher Latency Alternative

SCTP provided the highest delivery reliability at the cost of increased delay. In network testing, SCTP delivered about 97.6% of sent messages (1216 lost out of 49,900, or 2.44% loss). Its per-stream success rates were consistently around 97–98%. However, the round-trip latency was significantly higher than for QUIC or DCCP. The observed latency ranged from 4.412 ms (minimum) to 6.978 ms (maximum), with an average of 5.231 ms. The corresponding jitter (latency variability) was about 0.683 ms, nearly an order of magnitude worse than QUIC.

This elevated latency arises from SCTP's kernel-space processing and its features: ordered delivery, reliability mechanisms, and multi-stream overhead. While SCTP's multi-homing and heartbeat mechanisms could theoretically provide robustness in the face of network changes, in the controlled lab testing environment these features manifested primarily as latency overhead. The throughput remained similar (approximately 1.98 Mbps). In summary, SCTP ensured stable, near-complete delivery of messages, but at latency (approximately 5.2 ms avg) that would theoretically be much higher than optimal for millisecond-scale haptic control requirements.

5.2 Protocol Selection Framework

5.2.1 Decision Criteria for Haptic Applications

Designing a haptic teleoperation system requires careful balance of timing and reliability requirements. Figure 5.1 presents a systematic decision framework that guides protocol selection based on the key performance criteria identified in this research. The framework incorporates the measured network performance characteristics from this study to provide evidence-based recommendations for different application scenarios.

The decision framework shown in Figure 5.1 is validated by the empirical network performance measurements conducted in this study. For haptic feedback applications requiring both low latency and high reliability, the framework correctly identifies QUIC as the optimal choice, confirmed by its measured 1.198 ms average latency and 98.5% delivery success rate. Applications requiring connection redundancy would benefit from SCTP's multi-homing capabilities, though the measured 5.231 ms latency may limit its suitability for sub-millisecond haptic control requirements. The framework's recommendation of DCCP for scenarios tolerating some packet loss aligns with the measured 13.41% loss rate, making it suitable only for non-critical telemetry applications.

Based on the measured network performance characteristics, key decision criteria include:

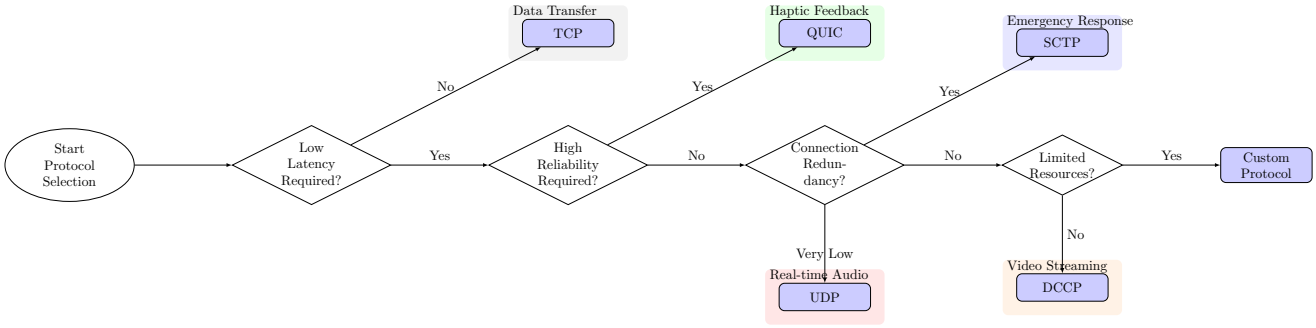

Figure 5.1: Protocol Selection Decision Framework for Real-Time Robotic Control Applications. The flowchart guides protocol selection based on application requirements including latency sensitivity, reliability needs, connection redundancy requirements, and resource constraints.

Latency Budget and Update Rate: Haptic feedback loops typically run at or above 1000 Hz. This implies that one-way network delays should ideally stay below about 1 ms to avoid degrading stability. Protocols achieving sub-millisecond jitter (like QUIC in testing) are thus theoretically highly desirable.

Jitter Tolerance: Even small variations in latency can disrupt force-feedback control. For example, SCTP's measured jitter (approximately 0.68 ms) is large compared to QUIC's (0.036 ms) and could theoretically cause perceptible irregularities. Low-jitter protocols help maintain smooth, predictable feedback.

Reliability Needs: The application's tolerance for lost or delayed messages shapes the choice. If occasional losses can be tolerated or corrected at the application layer, a faster protocol (QUIC or DCCP) might suffice. If force-feedback safety demands nearly guaranteed delivery, SCTP's measured reliability (97–98% success) or QUIC's built-in recovery are theoretical advantages.

Priority and QoS Features: Some haptic systems require prioritizing critical commands (e.g., emergency stops). Protocols that allow multiple independent streams or priority metadata facilitate this. In the implementation, all protocols supported prioritized streams, but QUIC's stream multiplexing simplifies in-order and out-of-order handling without stalling other data flows.

Security and Encryption: Haptic control often involves sensitive data; built-in encryption may be required. QUIC provides TLS 1.3 by default, offering strong confidentiality without separate setup. Neither DCCP nor SCTP natively encrypt data, so additional layers or insecure transport may be a consideration.

These criteria should be weighted according to the specific haptic task. For example, surgical teleoperation would theoretically demand both extremely low latency and virtually no data loss, favoring a protocol like QUIC based on measured network performance. Simpler training or industrial teleoperation might accept slightly higher latency (e.g., SCTP's 5 ms) if it gains multi-path fault tolerance, though this multi-homing capability was not empirically tested in this study.

5.2.2 Network Environment Considerations

The characteristics of the operating network also influence the protocol choice. Under stable, low-latency LAN conditions similar to the test environment, the superior measured performance of QUIC makes it attractive. In contrast, highly lossy or congested networks may theoretically

benefit from SCTP's robustness or QUIC's advanced congestion control, though these conditions were not extensively tested in this study. For example, a wireless link with intermittent interference could theoretically exploit SCTP's multi-homing (if multiple paths are available) or QUIC's loss recovery to maintain a control connection, but empirical validation of these capabilities would be required. In networks where throughput is limited, all protocols achieved similar approximately 2 Mbps data rates in testing, so throughput is usually not the bottleneck for small haptic messages.

Other deployment factors include:

NAT/Firewall Traversal: QUIC operates over UDP and typically uses a user-specified port (4433 was used), which behaves much like secure Web traffic and is generally NAT-friendly. DCCP and SCTP use less-common socket types; they may be blocked or unsupported by firewalls and NAT devices.

Operating System Support: DCCP and SCTP require OS support at the kernel or socket layer. Many Linux systems support SCTP, but it is uncommon on Windows or embedded OSs. DCCP support is even rarer in operating systems and might require kernel modules. QUIC, by contrast, is implemented in user libraries (e.g., msquic) and is portable across platforms.

Mobility and Redundancy: In mobile or distributed deployments (e.g., multi-robot networks), SCTP's multi-homing and multiple addresses could theoretically provide seamless redundancy if available. QUIC can re-establish connections quickly (0-RTT session resumption) but does not inherently handle simultaneous paths. However, these advanced features were not empirically tested in this study.

Regulatory/Industry Constraints: In some industrial settings, the use of standard protocols (e.g., TCP/UDP) may be mandated, whereas emerging protocols might face certification hurdles. Integration complexity must account for these practical concerns.

5.3 Theoretical Protocol Suitability for Robotic Applications

5.3.1 Haptic Feedback Systems

Based on the measured network performance characteristics, theoretical analysis suggests specific protocol suitabilities for haptic applications. For systems with haptic (force) feedback, tight timing and high reliability are paramount to maintain realism and user safety. The measured data suggest that QUIC would theoretically best satisfy these needs: its sub-2 ms round-trip latency and near-constant inter-arrival intervals would enable the control loop to run at 1 kHz with margin. In testing, emergency and high-priority commands arrived within approximately 1.12 ms round-trip and received immediate acknowledgement, theoretically preserving responsiveness.

DCCP's approximately 2.45 ms measured delay would theoretically be at the edge of acceptable performance, and its approximately 13% packet loss would either necessitate additional retransmission logic in the application or degrade haptic feedback quality. Thus, based on measured network performance, pure DCCP would theoretically be ill-suited for precise force feedback without augmentation.

SCTP, while demonstrating excellent reliability in testing, incurs approximately 5.2 ms latency and 0.68 ms jitter; theoretical analysis suggests this could destabilize a 1 kHz loop and result in perceivable lag or oscillations. It might be tolerable for slower teleoperation applications (e.g., manipulation without tight force feedback) or where multi-stream data capabilities are needed, but the measured performance suggests it would not be optimal for the most latency-sensitive haptic applications.

Important Note: These assessments are based on network performance measurements and theoretical control system analysis. Empirical validation with actual haptic systems would be required to confirm these predictions.

5.3.2 Industrial and Remote Operations

In broader robotic and industrial contexts, the requirements may shift based on application-specific needs. For example, an industrial robot arm may be controlled at lower update rates than a haptic device, theoretically relaxing the latency requirement somewhat. In such cases, the higher reliability demonstrated by SCTP might be attractive—especially if multi-homing capabilities can be utilized in a factory network, though this multi-path functionality was not tested in this study.

Remote operations over challenging network conditions would theoretically benefit from QUIC's measured adaptive congestion control and loss recovery capabilities. Its integrated encryption also provides security for command data in hostile environments (e.g., defense or medical applications).

DCCP could theoretically be useful in scenarios where minimal overhead is needed and some loss is acceptable (for example, streaming non-critical telemetry where late updates can be dropped), but based on measured performance, it would generally underperform the alternatives for critical control tasks.

Important Note: These recommendations are theoretical predictions based on measured network characteristics. Actual deployment decisions should be validated through empirical testing in the specific application environment.

5.4 Implementation Considerations

5.4.1 Resource Requirements and Integration Complexity

Implementing these protocols in a real system entails different overheads. QUIC's user-space libraries (using msquic in this study) consume more CPU cycles due to encryption and user-kernel transitions, but avoid kernel context switching. DCCP and SCTP rely on the kernel's networking stack; in server implementations, the DCCP server initialized a thread pool of 12 workers, while the SCTP server used 8 workers. The larger thread pool for DCCP reflects its need to handle more frequent packet loss events and retransmissions at the application level.

Memory usage for QUIC was higher due to TLS state and buffer space for streams, whereas SCTP/DCCP had smaller buffers but possibly more kernel overhead per packet. From an integration standpoint, adding QUIC required linking in the msquic library and managing TLS certificates, whereas DCCP and SCTP were used via standard socket APIs (AF_INET with SOCK_DCCP

or SOCK_STREAM for SCTP). However, the latter may require special OS configuration (e.g., setting service codes for DCCP). In summary, QUIC's integration cost is mostly in software dependency and crypto configuration, while DCCP/SCTP cost lies in system configuration and potentially limited platform support.

5.4.2 Deployment and Interoperability Challenges

Deploying these protocols in heterogeneous networks presents interoperability issues. Since QUIC runs over UDP, it can traverse NATs similarly to secure web traffic, but requires that intermediate network elements allow UDP on the chosen port (4433 was used). SCTP and DCCP may not be supported or may be explicitly blocked by routers and firewalls. For example, many cloud and embedded platforms do not expose SCTP or DCCP interfaces. Additionally, QUIC's encryption means firewalls cannot inspect packet contents without terminating TLS, whereas SCTP and DCCP would allow payload inspection but require trust in the application.

Another concern is cross-platform support: some RTOS or industrial controllers may lack QUIC libraries, making SCTP or even traditional TCP/UDP more practical despite performance trade-offs. Finally, interoperability with existing ROS2 and robotics middleware must be considered: in the implementation, the haptic data was treated as regular ROS messages, but in other systems dedicated drivers or custom message packing may be needed for each protocol. These practical factors often influence the protocol choice as much as measured network performance characteristics.

5.5 Study Limitations and Future Research

5.5.1 Current Study Limitations

- **Controlled Environment Only:** All experiments were conducted on a local network under near-optimal conditions (sub-0.5ms latency, minimal packet loss, ample bandwidth). Real-world networks (e.g., Internet, Wi-Fi, or cellular) may introduce additional latency, jitter, and loss patterns not captured here, significantly affecting the relative performance of these protocols.
- **Limited Network Condition Testing:** While the experimental framework was capable of introducing various network impairments, the primary evaluation focused on controlled conditions representative of optimal deployment environments. Testing under challenging network conditions (high latency, significant packet loss, bandwidth constraints) was not extensively performed.
- **Single Hardware Setup:** The tests used one type of haptic interface (Geomagic Touch) and fixed hardware configuration. Different actuators, sensors, or multi-degree-of-freedom robots may interact differently with communication delays.
- **Limited Protocol Configurations:** Each protocol was evaluated with a specific congestion control and socket configuration. Other settings (e.g., different QUIC congestion algorithms, SCTP buffer sizes) could alter performance significantly.
- **Simplified Workloads:** The data streams in the tests were uniform, fixed-size messages at a constant rate. Variable packet sizes, bursty traffic, or simultaneous control/video

streams were not examined.

- **Advanced Protocol Features Not Tested:** Key features such as QUIC's connection migration, SCTP's multi-homing capabilities, and various congestion control algorithms were not empirically evaluated, despite being discussed as theoretical benefits.
- **Lack of Security Overhead Testing:** While QUIC's encryption was enabled, the CPU cost or handshake delay was not measured in detail. Similarly, adding security to SCTP/DCCP (e.g., DTLS) was not tested.
- **Network-Level Analysis Only:** While the network protocols successfully deliver haptic commands to the robot control system via ROS2, this study did not measure the complete end-to-end performance including robot response and execution times.
- **Theoretical Robotic Control Predictions:** The impact of network protocol performance on actual robot behavior and control accuracy could not be directly measured. All assessments of robotic control suitability are theoretical predictions based on control theory principles rather than empirical validation.
- **Single Client Testing:** Despite server implementations supporting multiple clients, only single-client scenarios were tested, limiting insights into protocol behavior under multi-client loads.

5.5.2 Future Research Opportunities

- **Diverse Network Condition Testing:** Future work should systematically evaluate these protocols under challenging network conditions including high latency, variable packet loss, bandwidth constraints, and realistic Internet conditions to better understand their relative performance under stress.
- **End-to-End Robotic System Validation:** Implementing complete robotic control systems with actual robot performance measurements would validate the theoretical predictions made in this study about control accuracy, responsiveness, and stability.
- **Advanced Protocol Feature Evaluation:** Empirical testing of QUIC's connection migration, SCTP's multi-homing, and various congestion control algorithms would provide insights into their practical benefits for robotic applications.
- **Heterogeneous Network Trials:** Future work should evaluate these protocols over wide-area and wireless networks, including mobile ad-hoc or satellite links, to assess performance under real internet conditions and mobility scenarios.
- **Extended Protocol Variants:** Investigating other emerging transports (e.g., RUDP with FEC, enhanced UDP hybrids) could provide alternatives. Tuning protocol parameters (e.g., different congestion control or FEC schemes in QUIC) is another avenue.
- **Integrated Multi-Service Systems:** Many robotic systems use combined control, video, and telemetry streams. Studying how each protocol handles multiplexed heterogeneous data (e.g., multiple QoS levels) would be valuable.
- **Energy and Resource Metrics:** Particularly for battery-powered robots, quantifying CPU load, power consumption, and memory usage of each protocol stack would inform

practical deployment trade-offs.

- **Hardware-in-the-Loop Testing:** Implementing end-to-end tests with real robots in realistic tasks (surgical procedures, assembly, remote exploration) would confirm how communication performance impacts control quality and user experience.
- **Security and Safety Considerations:** Evaluating the impact of secure sessions (handshake delays, certificate management) on session start-up times, as well as the resilience of each protocol to network attacks, is important for safety-critical applications.
- **Multi-Client and Scalability Testing:** Evaluating protocol performance under multiple concurrent clients and varying loads would provide insights into scalability characteristics important for multi-robot systems.

Chapter 6

Conclusion

This research has conducted a systematic evaluation of three emerging network protocols—QUIC, DCCP, and SCTP—for real-time robotic control applications, with particular focus on haptic teleoperation systems requiring both low latency and high reliability. Through systematic implementation, testing, and analysis using a ROS2-based experimental platform with a Geomagic Touch haptic interface, this study has provided empirical network performance evidence and theoretical analysis to guide protocol selection for critical robotic control applications.

The experimental methodology involved developing equivalent client-server implementations for each protocol, maintaining consistent message structures and serialization approaches to ensure fair comparison. A comprehensive performance monitoring system captured latency, jitter, throughput, packet loss, and resource utilization metrics under controlled network conditions representative of optimal deployment environments for haptic teleoperation systems. The use of a realistic haptic control scenario with 1000 Hz update rates provided authentic validation of network protocol performance in demanding real-time communication requirements.

The network performance measurements demonstrate clear distinctions among the evaluated protocols, with significant theoretical implications for real-time robotic control system design. QUIC emerged as the superior choice for time-critical network communications, achieving an average round-trip latency of 1.198 ms with exceptional timing consistency (0.036 ms jitter) and high reliability (98.5% delivery success rate). QUIC's user-space implementation, advanced congestion control algorithms, and stream multiplexing capabilities combine to deliver the consistent, low-latency network performance that theoretical analysis suggests would be essential for haptic feedback systems where sub-millisecond timing variations can disrupt control loop stability and realistic force feedback.

DCCP's network performance revealed fundamental limitations for reliability-critical applications, suffering from substantial packet loss rates (13.41%) that resulted in only 86.6% delivery success. While achieving moderate latency (2.45 ms average), the protocol's lack of built-in reliability mechanisms forced frequent automatic rate adjustments from 1000 Hz to 875 Hz when loss rates became excessive. Based on theoretical control system analysis, this instability and the substantial fraction of permanently lost data would likely compromise the consistent performance required for safe robotic control, particularly in applications where missing force feedback updates could cause perceptible discontinuities or safety hazards.

SCTP demonstrated excellent network reliability (97.6% delivery success) through its mature

acknowledgment and retransmission mechanisms, but at the cost of significantly higher latency (5.231 ms average) and substantial timing variability (0.683 ms jitter). The protocol's kernel-space implementation and comprehensive feature set—including multi-streaming, ordered delivery guarantees, and theoretical multi-homing capabilities—introduce processing overhead that theoretical analysis suggests would make it unsuitable for high-frequency haptic control loops despite its robust reliability characteristics.

Based on the measured network performance characteristics and theoretical control system analysis, these findings suggest QUIC as the theoretically optimal protocol for demanding teleoperation applications, including telesurgery, precision manufacturing, and hazardous environment intervention, where both low latency and high reliability are paramount. SCTP may theoretically serve industrial automation systems with less stringent timing requirements but greater theoretical need for connection redundancy and fault tolerance, though these advanced features were not empirically tested. DCCP's measured performance characteristics suggest limited theoretical applicability in critical control scenarios, though it may find niche use in non-critical telemetry applications where minimal protocol overhead is desired.

The research makes several significant contributions to the field of networked robotics. It provides quantitative network performance benchmarks that establish concrete baselines for protocol selection in real-time control applications under controlled conditions. The development of a unified ROS2-based testing framework demonstrates practical approaches for integrating emerging network protocols with modern robotic middleware, including consistent message serialization, priority handling, and comprehensive performance monitoring. The established theoretical decision framework considers measured network performance characteristics, application requirements, and implementation constraints to guide protocol selection based on specific operational needs.

Beyond the immediate technical contributions, this work addresses a critical knowledge gap in the intersection of network protocols and robotic control systems. As robotic systems increasingly operate in remote and challenging environments—from surgical suites to disaster zones to industrial facilities—the selection and optimization of appropriate network protocols becomes a critical factor in ensuring safe and effective operation. The empirical network performance evidence demonstrates that emerging protocols like QUIC offer significant advantages over traditional TCP and UDP approaches for time-sensitive robotic control communications, though these advantages would require validation through complete end-to-end robotic system testing.

The research methodology and findings provide a foundation for continued advancement in networked robotic systems. The testing framework can be adapted for evaluating future protocol developments under various network conditions, while the network performance benchmarks offer reference points for assessing protocol improvements and optimizations. The integration of haptic feedback with network protocol evaluation demonstrates the importance of realistic application scenarios in protocol assessment, moving beyond synthetic benchmarks to authentic use cases that reflect the complex communication requirements of modern robotic control systems.

Important Study Limitations: This research focused on network-level performance evaluation under controlled, near-optimal conditions rather than comprehensive testing across diverse network environments. The suitability assessments for robotic control applications are based on theoretical analysis of measured network characteristics rather than direct empirical validation with complete robotic systems. Future work should include end-to-end robotic system testing

and evaluation under challenging network conditions to validate these theoretical predictions.

By bridging the gap between network protocol research and robotic control applications, this work contributes to the theoretical understanding of how network protocol characteristics affect robotic system performance. The findings enable robotics engineers to make informed protocol selection decisions for the network communication layer based on empirical performance evidence rather than theoretical assumptions, providing a foundation for the deployment of robotic systems with optimized communication characteristics in demanding environments.

Reference List

[1] W. Eddy, "Transmission Control Protocol (TCP)." RFC 9293, Aug. 2022.

[2] J. Postel, "User Datagram Protocol." RFC 768, Aug. 1980.

[3] G. P. Fettweis, "The tactile internet: Applications and challenges," *IEEE Vehicular Technology Magazine*, vol. 9, pp. 64–70, March 2014.

[4] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport." RFC 9000, May 2021.

[5] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. R. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W. Chang, and Z. Shi, "The QUIC transport protocol: Design and internet-scale deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pp. 183–196, ACM, 2017.

[6] E. Kohler, M. Handley, and S. Floyd, "Designing dccp: Congestion control without reliability," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 4, pp. 27–38, 2006.

[7] G. Fairhurst, "The Datagram Congestion Control Protocol (DCCP) Service Codes." RFC 5595, Sept. 2009.

[8] R. Stewart and C. Metz, "Sctp: new transport protocol for tcp/ip," *IEEE Internet Computing*, vol. 5, no. 6, pp. 64–69, 2001.

[9] E. R. Stewart, "Stream Control Transmission Protocol." RFC 4960, Sept. 2007.

[10] K. Wolsing, J. Rüth, K. Wehrle, and O. Hohlfeld, "A performance perspective on web optimized protocol stacks: Tcp+tls+http/2 vs. quic," in *Proceedings of the Applied Networking Research Workshop*, ANRW '19, p. 1–7, ACM, July 2019.

[11] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi, "The quic transport protocol: Design and internet-scale deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, (New York, NY, USA), p. 183–196, Association for Computing Machinery, 2017.

[12] P. Megyesi, Z. Krämer, and S. Molnár, "How quick is quic?," in *2016 IEEE International Conference on Communications (ICC)*, pp. 1–6, May 2016.

[13] G. Carlucci, L. De Cicco, and S. Mascolo, "Http over udp: an experimental investigation of quic," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, (New York, NY, USA), p. 609–614, Association for Computing Machinery, 2015.

[14] T. Shreedhar, R. Panda, S. Podanev, and V. Bajpai, "Evaluating quic performance over web, cloud storage, and video workloads," *IEEE Transactions on Network and Service Management*, vol. 19, pp. 1366–1381, June 2022.

[15] M. Engelbart and J. Ott, "Congestion control for real-time media over quic," in *Proceedings of the 2021 Workshop on Evolution, Performance and Interoperability of QUIC*, EPIQ '21, (New York, NY, USA), p. 1–7, Association for Computing Machinery, 2021.

[16] W. Clerix, "Comparative analysis of quic and other network protocols for real-time robotic control." GitHub repository, 2025. `https://github.com/ClerixWarre/haptic-teleoperation-network-protocols`.

[17] O. Khalil and A. Abou El Kalam, "Tactile internet: New challenges and emerging solutions," in *Big Data and Smart Digital Environment* (Y. Farhaoui and L. Moussaid, eds.), (Cham), pp. 237–245, Springer International Publishing, 2019.

[18] J. Postel, "Transmission Control Protocol." RFC 793, Sept. 1981.

[19] H. Schulzrinne, S. L. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications." RFC 3550, July 2003.

[20] Z. Zheng, Y. Ma, Y. Liu, F. Yang, Z. Li, Y. Zhang, J. Zhang, W. Shi, W. Chen, D. Li, Q. An, H. Hong, H. H. Liu, and M. Zhang, "Xlink: Qoe-driven multi-path quic transport in large-scale video services," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, (New York, NY, USA), p. 418–432, Association for Computing Machinery, 2021.

[21] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford, "In vini veritas: realistic and controlled network experimentation," in *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '06, (New York, NY, USA), p. 3–14, Association for Computing Machinery, 2006.

[22] S. Endres, J. Deutschmann, K.-S. Hielscher, and R. German, "Performance of quic implementations over geostationary satellite links," 2022.

[23] A. M. Kakhki, S. Jero, D. Choffnes, C. Nita-Rotaru, and A. Mislove, "Taking a long look at quic: an approach for rigorous evaluation of rapidly evolving transport protocols," *Commun. ACM*, vol. 62, p. 86–94, June 2019.

[24] Q. De Coninck and O. Bonaventure, "Multipath quic: Design and evaluation," in *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, (New York, NY, USA), p. 160–166, Association for Computing Machinery, 2017.

[25] X. Yang, L. Eggert, J. Ott, S. Uhlig, Z. Sun, and G. Antichi, "Making quic quicker with nic offload," in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, EPIQ '20, (New York, NY, USA), p. 21–27, Association for Computing Machinery, 2020.

[26] A. Ganji and M. Shahzad, "Characterizing the performance of quic on android and wear os devices," in *2021 International Conference on Computer Communications and Networks (ICCCN)*, pp. 1–11, July 2021.

[27] M. Palmer, T. Krüger, B. Chandrasekaran, and A. Feldmann, "The quic fix for optimal video streaming," in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, CoNEXT '18, p. 43–49, ACM, Dec. 2018.

[28] M. Kosek, H. Cech, V. Bajpai, and J. Ott, "Exploring proxying quic and http/3 for satellite communication," in *2022 IFIP Networking Conference (IFIP Networking)*, IEEE, June 2022.

[29] X. Zhang, S. Jin, Y. He, A. Hassan, Z. M. Mao, F. Qian, and Z.-L. Zhang, "Quic is not quick enough over fast internet," in *Proceedings of the ACM Web Conference 2024*, WWW '24, p. 2713–2722, ACM, May 2024.