

# socket programming report

## kelly casey

*by Kelly Casey*

---

**Submission date:** 22-May-2025 02:02PM (UTC+0100)

**Submission ID:** 260166133

**File name:** 31343\_Kelly\_Casey\_socket\_programming\_report\_kelly\_casey\_664977\_443895131.docx (82.08K)

**Word count:** 3871

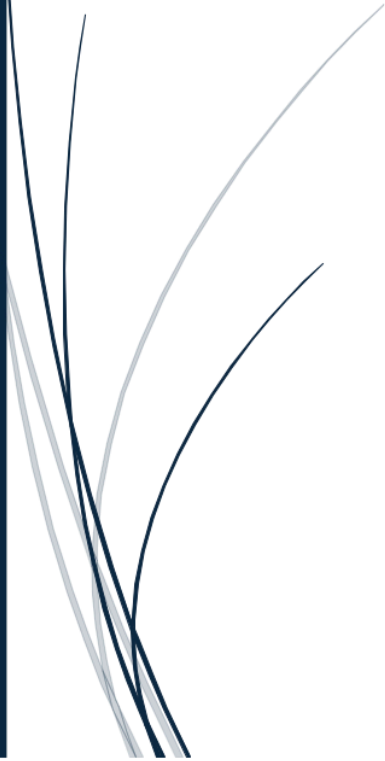
**Character count:** 23501



5/15/2025

# Understanding Key Protocols and Design Considerations in Networked Application Development

ACCA6028



Kelly Casey (2106451)

## Contents

|                                                       |    |
|-------------------------------------------------------|----|
| 1.Introduction .....                                  | 2  |
| 2.TCP and Reliable Delivery .....                     | 3  |
| TCP as a Reliable Transport Protocol.....             | 3  |
| The Three-Way Handshake.....                          | 3  |
| Reliability Mechanisms in TCP .....                   | 4  |
| The Two Generals' Problem .....                       | 4  |
| Relevance to Programmers and Application Design.....  | 5  |
| 3.MTU, PDUs, and Fragmentation .....                  | 5  |
| What is MTU? .....                                    | 6  |
| PDUs and the OSI Model .....                          | 6  |
| IP Fragmentation and Its Consequences.....            | 7  |
| Path MTU Discovery (PMTUD).....                       | 7  |
| Relevance to Programmers and Application Design.....  | 7  |
| 4.Synchronous vs. Asynchronous I/O.....               | 8  |
| Synchronous Communication .....                       | 9  |
| Asynchronous Communication .....                      | 9  |
| I/O Dispatch Models .....                             | 10 |
| Application Scenarios and Design Implications .....   | 11 |
| 5.Interoperability.....                               | 11 |
| What Is Interoperability? .....                       | 12 |
| Achieving Interoperability: Standards and Tools ..... | 12 |
| Challenges in Maintaining Interoperability .....      | 13 |
| Relevance to Programmers and Application Design.....  | 13 |
| Conclusion .....                                      | 14 |
| Bibliography .....                                    | 15 |

# 1.Introduction

In modern software development, designing and building efficient, reliable networked applications is an essential capability for software developers and systems engineers [1]. As systems become more interconnected, it's important to understand the core protocols and behaviours that drive network communication. This research-oriented report covers several key areas that directly affect programming and software design:

1. **TCP and Reliable Delivery:**

We start by exploring how the Transmission Control Protocol (TCP) guarantees ordered, error-checked data transfer, including its three-way handshake and the Two Generals' Problem that underlies reliability limits [2].

2. **MTU, PDUs, and Fragmentation:**

Next, we examine the Maximum Transmission Unit (MTU), how Protocol Data Units (PDUs) are sized, and why IP fragmentation, though necessary at times can impair performance and introduce security concerns.

3. **Synchronous vs. Asynchronous I/O:**

The report then compares blocking (synchronous) and non-blocking (asynchronous) communication patterns, looking at thread-pool and event-driven dispatch models and how they influence application responsiveness.

4. **Interoperability:**

Lastly, we discuss how to enable different systems to work together through standard protocols (e.g. HTTP, gRPC), common data formats (JSON, Protobuf), and sound API design, ensuring seamless integration across platforms.

## 2.TCP and Reliable Delivery

The Transmission Control Protocol (TCP) is a foundational transport-layer protocol [3]that guarantees reliable data delivery over both public and private networks. As a connection-oriented service, TCP establishes a logical link between two endpoints before any application data is sent. Through its handshake process, sequence numbering, checksums, and retransmission strategies, it ensures that bytes arrive intact and in order. This dependable behaviour reinforces essential services like web browsing, email delivery, and file transfers. A solid grasp of TCP's mechanisms is invaluable for developers building networked software that must tolerate delays, losses, or reordering in the underlying network.

### TCP as a Reliable Transport Protocol

Unlike the connectionless UDP, which sends datagrams without guarantees, TCP provides ordered, error-checked delivery and ensures no data is lost or duplicated. It does this through a combination of mechanisms working together:

- connection setup and teardown
- sequence numbering
- checksums
- acknowledgments
- retransmissions

These safeguards are vital for scenarios where missing or corrupted data is unacceptable, such as financial transactions, authentication exchanges, or transferring structured documents. By managing connection state and dynamically controlling data flow, TCP delivers the reliability needed by these critical applications.

### The Three-Way Handshake

Before any data is exchanged, TCP endpoints perform a three-step handshake to establish a connection and agree on sequence numbers [4]:

- SYN (Synchronize): The client sends a segment with the SYN flag set and its Initial Sequence Number (ISN), signalling a connection request.
- SYN-ACK (Synchronize-Acknowledge): The server replies with both SYN and ACK flags set, acknowledging the client's ISN and providing its own ISN.
- ACK (Acknowledge): The client sends a final ACK, confirming the server's ISN.

Once this exchange completes, both sides share a synchronized sequence-number space, and the connection is ready for reliable, in-order data transfer.

## Reliability Mechanisms in TCP

Once the connection is in place, TCP keeps data delivery reliable through several coordinated mechanisms:

- **Sequence Numbers:** Each byte of data carries a unique sequence number. The receiver uses these to reorder any out-of-sequence segments before passing data to the application. [5]
- **Acknowledgments (ACKs):** After receiving data correctly, the receiver sends an ACK indicating the next expected sequence number. If the sender doesn't get an ACK within its timeout, it retransmits the unacknowledged data.
- **Checksums:** Every TCP segment includes a checksum that covers its header and payload. Corrupted segments fail the checksum check and are discarded.
- **Flow Control (Sliding Window):** The receiver advertises a window size that limits how many bytes the sender can transmit without receiving further ACKs, preventing buffer overflows.
- **Congestion Control:** TCP dynamically adjusts its send rate based on network feedback. Algorithms like TCP Tahoe, Reno, and Cubic increase throughput when conditions are good and back off when packet loss or delays suggest congestion.

Together, these features let TCP adapt to changing network conditions, recover from errors, and deliver data in order even when packets are lost, delayed, or arrive out of turn

## The Two Generals' Problem

Even with all its safeguards, TCP cannot overcome the fundamental uncertainty of unreliable networks, the same issue captured by the Two Generals' Problem [5]. In that thought experiment, two commanders try to coordinate an attack by sending messengers through hostile territory, but no messenger (or reply) can be guaranteed to arrive. As a result, there is no way to be sure both sides share the same information.

TCP doesn't solve this paradox however it does work around it. By using timeouts, retransmissions, and sequence-number acknowledgments, TCP drives the chance of lost data so low that most applications never notice a failure. Still, its reliability guarantees remain probabilistic, the protocol reduces risk to a practical minimum but cannot make perfect certainty possible returns an ICMP 'Fragmentation Needed' message.

## Relevance to Programmers and Application Design

When building networked applications, developers need to account for how TCP behaves in real-world environments:

- **Error Handling and Timeouts:** Even though TCP retries lost segments, your code should decide when to give up, alert users, or retry at the application level.
- **Resource Management:** Each open TCP socket consumes memory and file descriptors. High-scale services should use connection pools, keep-alive settings, or asynchronous I/O to avoid exhausting system resources.
- **Latency vs. Reliability:** TCP's guarantees add overhead and latency. For use cases like real-time gaming or VoIP, use UDP or a hybrid approach that trades some reliability for faster delivery.
- **Security Measures:** The TCP handshake can be abused (e.g., SYN floods). Protect your service with techniques such as SYN cookies, rate limiting, and appropriate firewall rules.

By designing with these TCP characteristics in mind, you'll ensure your applications remain robust, efficient, and secure across a range of network conditions.

## 3.MTU, PDUs, and Fragmentation

Networks impose both physical and logical limits on data transfer. One of the most important is the Maximum Transmission Unit (MTU) which is the largest packet size

a link can carry without splitting it. If a packet exceeds the MTU, it must be fragmented at the IP layer, which can slow transmission and introduce additional failure points. For developers, choosing the right packet size helps avoid needless fragmentation, improves throughput, and reduces the risk of packet loss or reassembly errors. Understanding MTU constraints is therefore key when optimising networked applications for speed and reliability.

## What is MTU?

The Maximum Transmission Unit (MTU) specifies the largest packet, in bytes, that a network link can carry in a single frame without fragmentation. For Example: standard Ethernet has an MTU of 1500 bytes (not counting headers). Any packet larger than this must be split into smaller fragments that each fit within the MTU limit.

Although MTU is a concept at the data-link layer (OSI Layer 2), its effects reach up into the network and transport layers (Layers 3 and 4). When IP packets or TCP/UDP segments exceed the MTU, they trigger fragmentation or path MTU discovery, which can add overhead, increase latency, and complicate packet reassembly. Understanding and respecting MTU constraints is therefore crucial for efficient, reliable data transfer in networked applications.

## PDUs and the OSI Model

To see how MTU limits affect real data, it helps to look at Protocol Data Units (PDUs) at each OSI layer:

- Layer 2 (Data Link): PDU is the Frame, which carries the full packet plus its own header and trailer.
- Layer 3 (Network): PDU is the Packet, which adds an IP header on top of the payload.
- Layer 4 (Transport): PDU is the Segment (for TCP) or Datagram (for UDP), which adds the transport-layer header.

Each layer's header eats into the MTU, so your application payload must be small enough that, once the Ethernet, IP, and TCP/UDP headers are added, the total stays under the MTU. This means subtracting roughly 14 bytes for Ethernet, 20 bytes for IP, and 20 bytes for TCP leaving about 1446 bytes for data. Being mindful of these cascading overheads lets you avoid fragmentation and keep your transfers efficient.

## IP Fragmentation and Its Consequences

When an IP packet is larger than the MTU, the network layer splits it into smaller fragments, each with its own IP header and a fragment offset field. While this lets oversized packets cross links with smaller MTUs, fragmentation brings drawbacks:

- **Extra Overhead:** Every fragment carries a full IP header, so more bytes are sent overall.
- **Greater Loss Impact:** Losing any single fragment forces the sender to retransmit the whole original packet.
- **Reassembly Cost:** The receiver must buffer and reassemble fragments, consuming memory and CPU [6].
- **Security Vulnerabilities:** Attackers can misuse overlapping or out-of-order fragments to bypass firewalls or IDS rules.

Because of these issues, fragmentation is generally avoided in modern networks through techniques like Path MTU Discovery or by keeping packet sizes below the smallest MTU on a path.

## Path MTU Discovery (PMTUD)

Most networks rely on Path MTU Discovery (PMTUD) to avoid IP fragmentation. PMTUD works by sending packets with the “Don’t Fragment” (DF) flag set. [6] If any router on the path can’t forward the packet because it exceeds its MTU, it drops the packet and returns an ICMP “Fragmentation Needed” message. The sender then lowers its packet size and retries until it finds a size that traverses every link without fragmentation.

Because some networks block ICMP for security, PMTUD can fail silently leading to “black hole” connections where packets vanish without warning. To work around this, many modern stacks use Packetization Layer PMTUD (PLPMTUD) [7], which probes different sizes at the transport layer instead of depending on ICMP replies.

## Relevance to Programmers and Application Design

When an application sends large payloads such as video streams, file uploads, or bulk API responses it needs to work around MTU limits to keep performance smooth and connections reliable. Key strategies include:

- **Application-level chunking:** Split big messages into smaller pieces yourself rather than relying on IP to fragment packets.

- Header overhead: Remember that protocols (TCP, HTTP/2, QUIC) add their own headers, so plan your payload size so the full packet stays under the MTU.
- Buffer sizing: Tune your TCP send and receive buffers [9] to match expected packet sizes and avoid stalls or extra retransmissions.
- Security checks: Always validate or reject suspiciously small or malformed fragments to guard against evasion attacks.
- Real-world testing: Exercise your service over links with different MTUs [10], VPNs, tunnels, cellular networks, to catch any hidden failures before deployment.

Essentially, paying attention to MTU constraints prevents hidden bugs, cuts down on retransmissions, and ensures your application performs reliably across diverse network environments.

## 4. Synchronous vs. Asynchronous I/O

In networked applications, how and when data is exchanged has a direct impact on responsiveness and resource use. A fundamental choice is between synchronous and asynchronous I/O:

- Synchronous I/O: The calling thread waits for each operation to complete before moving on. This model is simple to reason about, but a blocked thread can't do anything else which limits scalability when many connections must be served concurrently.

- Asynchronous I/O: Operations return immediately, and completion is reported later via callbacks, futures, or events. This lets a small pool of threads service many connections without blocking, improving throughput under high load, however it adds complexity in control flow and error handling.

Understanding these paradigms and how they tie into I/O dispatch models (thread pools versus event loops) is essential when designing systems that must balance latency, throughput, and code maintainability.

## Synchronous Communication

Synchronous Communication happens when a sender issues a request and then blocks waiting until a response arrives before moving on. Operations occur one after another, each step depending on the prior one completing.

A familiar example is the HTTP/1.1 request–response cycle: the client sends a request and can't do anything else until the server replies. Traditional database queries, file reads/writes, and many RPC systems work the same way.

This model is straightforward to understand and implement, but it has drawbacks :

- Blocking Threads: While waiting, each thread sits idle, reducing overall throughput [11].
- Resource Underuse: Frequent I/O waits leave CPU and other resources underutilised.
- Limited Concurrency: In high-load scenarios, having one thread per request doesn't scale well.

Despite these drawbacks, synchronous I/O remains ideal for short-lived tasks, predictable workflows, or tools where simplicity and real-time response matter more than handling large numbers of concurrent operations.

## Asynchronous Communication

Asynchronous Communication lets you send off a request and keep running without waiting for the reply. When the response arrives, the application handles it via callbacks, event loops, promises, or message handlers. Because no thread is blocked, a single process can juggle many I/O tasks at once.

Examples include:

- WebSockets, enabling real-time, bidirectional messages between client and server

- Message brokers like RabbitMQ or Kafka, where producers and consumers work independently
- AJAX calls in browsers that fetch or send data without freezing the UI [12]

Key advantages:

- Improved Concurrency: Easily scales to handle large numbers of simultaneous operations.
- Resource Efficiency: Threads stay free to do work instead of waiting on I/O.
- Event-Driven Fit: Aligns well with architectures like microservices or interactive web apps.

however, asynchronous code is more complex: you need to manage callbacks, promises, or state machines, and debugging can get tricky when execution jumps between contexts.

## I/O Dispatch Models

I/O dispatch models determine how an application issues and handles I/O calls and directly shape whether your code behaves synchronously or asynchronously:

- Blocking I/O: Calls block the thread until data is read or written. It's easy to understand but ties up one thread per operation.
- Non-blocking I/O: Calls return immediately if no data is ready. Your code must poll or retry, which avoids blocking but can waste CPU cycles.
- Event-driven I/O (select/poll): The app registers sockets or file descriptors and waits for the OS to signal when they're ready [12]. You then handle each ready event.
- Multiplexed I/O (epoll/kqueue): Scalable, kernel-level event loops designed for thousands of connections with minimal overhead on Linux (epoll) or BSD/macOS (kqueue).
- Async/await: A language-level feature (in JavaScript, Python's asyncio, C#) that builds on event loops but lets you write code that looks synchronous, improving readability while still running non-blocking I/O under the hood.

Picking the right model is key: for example, Node.js uses a single-threaded, non-blocking event loop [13] to handle many simultaneous web connections efficiently, while Python's asyncio offers a similar pattern with async/await syntax, blending performance with clearer control flow.

## Application Scenarios and Design Implications

Choosing between synchronous and asynchronous communication shapes your application's architecture:

- Use Cases:
  - Real-time apps (chat, collaborative editing) thrive on asynchronous protocols like WebSockets.
  - Batch jobs or transactional systems often stick with synchronous calls to maintain strict ordering and consistency.
  - Simple REST APIs usually default to synchronous requests, whereas event-driven microservices rely on asynchronous messaging.
- Key Considerations:
  - Latency Sensitivity: Asynchronous I/O keeps interfaces responsive under load.
  - Error Handling: Async flows need careful retry and failure management to avoid lost or duplicate work.
  - Maintainability: Synchronous code is more straightforward to read but may require more threads to scale.

By weighing these factors, you can select the model that delivers both the performance and the reliability your application context demands.

## 5. Interoperability

In a word defined by tightly linked systems, interoperability is essential for any networked application. It describes the capacity of diverse software components across different platforms, languages, vendors, or architectures to exchange data and collaborate without conflict. For developers, prioritising interoperability means designing APIs, data formats, and communication protocols that can evolve independently while still integrating smoothly, ensuring applications remain adaptable, scalable, and resilient to future changes.

## What Is Interoperability?

Interoperability means that different systems can exchange data in a way that's both accurate and meaningful. In practice, this requires that information sent by one component can be correctly interpreted and processed by another, even when they were built with different technologies or in separate teams. Common interoperability scenarios include:

- Web applications talking to cloud-based APIs
- Mobile clients interacting with backend services
- IoT sensors sending data to central servers
- Microservices communicating within a distributed architecture

Without interoperability, systems end up in silos, making integration complex, limiting features, and driving up maintenance efforts over time.

## Achieving Interoperability: Standards and Tools

Interoperability relies on following open standards and industry conventions that define how systems send, format, and interpret data:

- Common Protocols: HTTP, FTP, SMTP, MQTT, and similar protocols offer a shared language for exchanging messages in a predictable way, regardless of the technologies on each side.
- Data Formats: JSON, XML, and Protocol Buffers provide structured encoding that any compliant parser can read. JSON's human-readable syntax and broad support make it the de facto choice for most REST APIs [15].
- API Specifications: REST, GraphQL, and gRPC define clear contracts for how services expose functionality. REST maps resources to URIs and uses standard HTTP methods [16]; GraphQL lets clients request precisely the data they need; gRPC delivers strongly typed RPCs over HTTP/2 using Protobuf.

- Service Discovery: In distributed systems, tools like Consul, etcd, or Kubernetes DNS enable services to locate one another dynamically, removing hardcoded addresses and simplifying scaling.
- Authentication & Authorization: OAuth 2.0, OpenID Connect, and JWT tokens standardise how clients prove identity and obtain permissions, ensuring secure interactions without inventing custom schemes.

By building on these widely adopted standards, applications can plug into existing ecosystems, evolve independently, and still communicate reliably across platforms and languages.

## Challenges in Maintaining Interoperability

Even with solid standards, interoperability still faces hurdles [17]:

- Versioning: APIs change over time. Without a clear versioning scheme like including a version in the URI or using custom headers, clients can break when fields are added, removed, or renamed.
- Schema Drift: When data models evolve independently, mismatches occur. Employing schema definitions (OpenAPI, JSON Schema) and automated validation helps catch these issues early. [17]
- Protocol Gaps: Legacy or proprietary protocols don't always play well with modern stacks. Bridging them often requires custom adapters or middleware layers.
- Performance Overhead: Translating between formats or routing through service meshes can add latency and CPU load, especially when joining very different systems.

Addressing these challenges calls for thoughtful API design, strict schema governance, and ongoing coordination between teams to keep services aligned.

## Relevance to Programmers and Application Design

For developers, interoperability isn't just a protocol choice it's a core design principle. It means creating systems that:

- Expose clear, well-documented interfaces
- Keep components loosely coupled
- Rely on platform-agnostic standards
- Include automated tests against real external services

- Allow new integrations without major rewrites

In practice, someone might pick REST + JSON for maximum compatibility or gRPC + Protobuf for high-performance, strongly typed APIs, and use an API gateway or middleware to bridge any gaps. By designing this way, the application can easily connect with third-party services, support multiple client platforms, and adapt as the technology landscape evolves.

## Conclusion

Building modern networked applications requires a solid understanding of how data moves and is managed across diverse systems. TCP underpins reliable communication, its three-way handshake establishes synchronized sequence numbers, while checksums and retransmissions guard against loss or corruption. Although TCP cannot guarantee perfection an impossibility highlighted by the Two Generals' Problem it reduces failures to a level acceptable for most real-world use cases.

Equally important is awareness of packet-sizing constraints. The Maximum Transmission Unit (MTU) determines how large each frame can be before IP fragmentation kicks in, and techniques like Path MTU Discovery help avoid the performance and security penalties that come with splitting packets. At the application layer, this translates into careful message chunking and buffer management to keep traffic flowing efficiently.

Finally, the choice between synchronous and asynchronous I/O models shapes an application's responsiveness and scalability. Blocking calls are simple but can tie up threads, while non-blocking, event-driven approaches let a few workers handle many

connections. Together with these dispatch strategies, interoperability through open protocols, standard data formats, and clear API design ensures that web apps, microservices, mobile clients, and IoT devices can exchange information seamlessly. By grounding software architecture in these networking principles, reliable transport, efficient packet handling, appropriate I/O models, and platform-agnostic interfaces developers can build systems that perform consistently, adapt easily, and integrate smoothly in today's interconnected landscape.

Exploring these topics has shown just how much networking principles shape real-world software design. Ideas that once seemed like abstract theory now directly inform decisions about reliability, performance, and scalability. This insight provides confidence in crafting architectures that meet both technical requirements and business objectives. In turn, my strengthened understanding of networking has fundamentally changed how I approach building and evolving software systems.

## Bibliography

- [1] A. S. Tanenbaum and D. Wetherall, *Computer Networks*, 2011: Pearson, 2011.
- [2] J. Postel, "Transmission Control Protocol," Internet Engineering Task Force (IETF), September 1981. [Online]. Available: <https://tools.ietf.org/html/rfc793>. [Accessed 9 5 2025].
- [3] J. F. a. R. K. W. Kurose, *Computer Networking: A Top-Down Approach*, Pearson: Boston, 2016.
- [4] R. Braden, "Requirements for Internet Hosts – Communication Layers," IETF, October 1989. [Online]. Available: <https://tools.ietf.org/html/rfc1122>. [Accessed 6 2 2025].
- [5] J. a. E. J. Barwise, *The Liar: An Essay on Truth and Circularity*, Oxford: : Oxford University Press, 1987.
- [6] C. Partridge, *Gigabit Networking*, 1994.

- [7] J. a. D. S. Mogul, "Title: Path MTU Discovery," IETF, November 1990. [Online]. Available: <https://tools.ietf.org/html/rfc1191>. [Accessed 8 4 2025].
- [8] M. M. a. J. Heffner, "datatracker.," march 2007. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc4821>. [Accessed 17 4 2025].
- [9] D. Borman, "TCP Extensions for High Performance," Internet Engineering Task Force (IETF), 2025 August 2014. [Online]. Available: <https://tools.ietf.org/html/rfc7323>. [Accessed 3 5 2025].
- [10] I. Pepelnjak, "MPLS MTU Problems and Fragmentation," IP Space, 2020. [Online]. Available: <https://blog.ipspace.net>. [Accessed 26 4 2025].
- [11] A. S. a. W. D. : Tanenbaum, Computer Networks, Boston: Pearson, 2011.
- [12] Mozilla, "Concurrency model and the event loop," Mozilla Developer Network, 2023. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop>. [Accessed 13 5 2025].
- [13] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events," in *Pattern Languages of Program Design*, Boston, Addison-Wesley, 1996, p. 529–545.
- [14] N. Foundation, " About Node.js Event Loop," Node.js, 2023. [Online]. Available: <https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/>. [Accessed 3 5 2025].
- [15] D. Crockford, "The application/json Media Type," IETF, july 2006. [Online]. Available: <https://tools.ietf.org/html/rfc4627>. [Accessed 26 3 2025].
- [16] L. a. A. M. Richardson, RESTful Web APIs, Sebastopol: O'Reilly Media, 2013.
- [17] S. Ghosh, "Interoperability in Heterogeneous Networks," *IEEE Internet Computing*, vol. 22, no. 4, p. 91–95, : 2018.
- [18] T. e. a. Wright, "OpenAPI Specification 3.," OpenAPI Initiative, 2018. [Online]. Available: <https://swagger.io/specification>. [Accessed 5 3 2025].
- [19] W. R. Stevens, TCP/IP Illustrated, Volume 1: The Protocols, Boston: Addison-Wesley, 1994.
- [20] V. Paxson, "End-to-End Internet Packet Dynamics," *EEE/ACM Transactions on Networking*, vol. 7, no. 3, p. 277–292, 1999.
- [21] A. S. a. W. D. anenbaum, Computer Networks, Boston: Pearson, 2011.

[22] J. F. a. R. K. W. Kurose, Computer Networking: A Top-Down Approach, Pearson, 2016.

# socket programming report kelly casey

## GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

A good report overall, well written. Probably a little too dependent on bullet points, which hurts depth in some areas. Nonetheless, a strong submission.

78/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

