

Ethan Shellem-Prince Assignment 1 Report

by Ethan Shellem-Prince

Submission date: 13-May-2025 04:45PM (UTC+0100)

Submission ID: 259102132

File name: 31725_Ethan_Shellem-Prince_Ethan_Shellem-Prince_Assignment_1_Report_664976_1501307185.docx (139.69K)

Word count: 3027

Character count: 18151

Socket Programming and Concurrency Report

Module Code: ACCA6028

Ethan Shellem-Prince

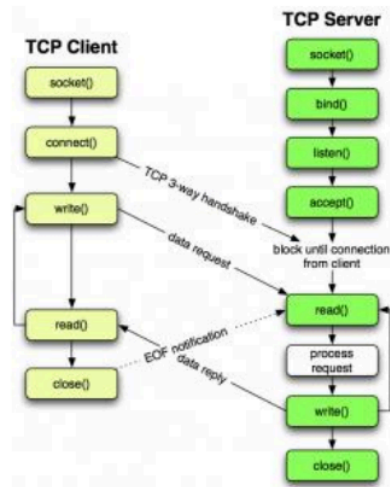


Table of Contents

1.	Introduction	3
2.	TCP Reliability: The Three-Way Handshake and the Two Generals' Problem	3
2.1	2.1 The Three-Way Handshake	4
2.2	Acknowledgement and Retransmission	4
2.3	The Two Generals' Problem	4
2.4	Relevance to Software Design	4
3.	MTU, PDU, and IP Fragmentation in the OSI Model	5
3.1	Understanding MTU and PDU	5
3.2	Fragmentation and Its Implications	5
3.3	Impact on Application and Protocol Design	6
3.4	OSI Model Context	6
4.	Synchronicity and Asynchronicity in Network Protocols	6
4.1	Synchronous Communication	7
4.2	Asynchronous Communication	7
4.3	I/O Dispatch Models	7
4.4	Design Implications for Developers	8
5.	Interoperability in Networked Applications	8
5.1	What Interoperability Means	8
5.2	Standards and Protocols	9
5.3	Techniques to Achieve Interoperability	9
5.4	Relevance to Software Engineering	9
6.	Conclusion	10
7.	References	11

1. Introduction

The internet has given us a new way of thinking and behaving. But its real power comes from the fact that there is a global and transparent information network where people can exchange information. The ready transmission across different distributed systems is fundamental to the smooth operation of most every application in this high-tech digital age. Performance and reliability of networked applications depend heavily on the underlying protocols and the principles guiding how data is transmitted. When users are engaged in real-time conversation, is remote access to a service provided and does data synchronization systemwide occur spontaneously?

As a software engineer, you must understand how these protocols work and also evaluate their design and implementation to build efficient, secure, and scalable applications.

This report examines the following key areas in networked communication. Its theoretical underpinnings, how the Transmission Control Protocol [1] (TCP) operates to ensure reliable delivery including a three-way handshake mechanism that guarantees this. The impact on maximum transmission unit (MTU) size, PDUs (protocol data units) and IP fragmentation, the distinction between synchronous and asynchronous communication in network protocols and what that means for I/O dispatch models. Interoperability, what is in networked systems and ways to make systems seamlessly communicate across different platforms.

Each of these sections provides examples that not only show what might be expected theoretically about each topic but also gives insight as to how good and bad software engineering can follow from understanding these concepts.

The report discusses how these principles inform design decision-making, affect application performance, and how they contribute to the overall user experience. All areas are examined in detail, with the aim of demonstrating how a deep understanding of network behaviour enables design and maintainability as well future-proofing systems.

2. TCP Reliability: The Three-Way Handshake and the Two Generals' Problem

With various mechanisms afforded to it, the Transmission Control Protocol (TCP) is widely considered the backbone of the Internet's transport layer.

It provides routine, ordered, and error-checked data delivery, ensuring stable and continuous communication between networked devices. As opposed to the User Datagram Protocol [2] (UDP), whose partnership with IP allows it to send packets to any recipient without establishing a connection first, TCP is connection-oriented protocol. For example, it establishes a session between sender and receiver before data can begin flowing down that route and sets up the necessary connection parameters between both parties. This defers the actual data transmission until the session is established.

The reliability mechanisms of TCP make it an essential part of applications where data integrity is important, such as email, web browsing and file transfers.

2.1 The Three-Way Handshake

When a TCP connection is established, the protocol employs a three-way handshake mechanism to the sender.

- SYN (synchronize): The client sends a TCP segment with the SYN flag indicating that it would like to establish an association, and it offers its initial sequence number (ISN).
- SYN-ACK (synchronize-acknowledge): The server replies with a SYN-ACK segment, acknowledging the acceptance of the client's ISN, plus it has its own ISN.
- ACK (acknowledge): The client sends an ACK back to the server in reply to its own request. Once this exchange is complete, both sides are synchronized, and reliable communication begins.

This handshake not only initiates the connection but also establishes synchronisation of sequence numbers and negotiation of TCP parameters such as the window size. These elements are vital for enabling orderly delivery and flow control throughout the session.

2.2 Acknowledgement and Retransmission

Now that the link is established, the TCP relies on positive acknowledgement plus timeout-based retransmission to achieve reliable delivery. If the receiver fails to Acknowledge receipt of a transmitted segment, the sender assumes it has been lost and will send it again. Other mechanisms such as Sliding Windows, congestion control (e.g., TCP Reno [3] or Cubic) and Checksum verification further improve reliability.

These built-in features make TCP highly suitable for the demands of high-reliability services. But they can also introduce significant latency, particularly in the case of heavy load and / or slip-configured media—an important consideration for real-time applications such as multiplayer games, where latency must be minimal for optimal user experience.

2.3 The Two Generals' Problem

The Two Generals' Problem [4] is a well-known theoretical dilemma in computer science that highlights the challenge of coordination over an unreliable communication medium. On opposite hills, two separate generals who must agree a time to attack their common enemy. They can only talk through messengers who might get intercepted. No matter how many acknowledgments are sent, certainty is never achieved because one side never can know whether other party received last message.

TCP, self-evident from its mathematical nature that it cannot solve this paradox. It leverages approaches such as the three-way handshakes and continuous acknowledgment, which both make very reasonable assumptions. This kind of behaviour is sufficient for most real-world networking environments.

2.4 Relevance to Software Design

To developers the knowledge of TCP behaviour is crucial for creating stable network applications featuring good performance. As an example:

- Applications should manage dropped or timed-out connections in a manner clear to the user.

- Developers can also opt to reuse connections (for instance, by using Persistent HTTP connections) if they know the cost of performing a handshake anew.
- Applications that do large transfers of data ought to stream the output both to avoid memory bloat and to benefit as much as possible from TCP flow control.

However, developers should realize that TCP is not suitable for all uses. Real-time systems may instead choose UDP or hybrids (e.g., QUIC [11]) which reduce latency but at the cost of loss recovery functions that are traditionally associated with reliable transport.

3. MTU, PDU, and IP Fragmentation in the OSI Model

As data moves along a network it has to be broken down into discrete packets. This is required by the hardware and protocol standards that underpin the network. Two concepts of significance in this situation are the Maximum Transmission Unit [5] (MTU) and the Protocol Data Unit [6] (PDU). Because of IP fragmentation, MTU and the PDU have given material support to the efficiency, credibility, and communication protocol design.

3.1 Understanding MTU and PDU

Maximum Transmission Unit, known as (MTU), is the largest size in bytes that can be sent over a link layer without requiring packets to be fragmented. For example, an ethernet has a MTU of 1500 bytes. If the size of the packet exceeds this limit, it will have to be broken down into smaller pieces--bringing about more complexity and potential extra processing overhead.

The Protocol Data Unit (PDU) is a generic term for the data formats used by each layer of OSI model [6] as follow.

- At the Application Layer: Message
- At the Transport Layer: Segment (TCP) or Datagram (UDP)
- At the Network Layer: Packet
- At the Data Link Layer: Frame

Every PDU contains the payload and protocol-defined headers. These headers will increase the size of the data unit, which must fit within MTU parameters. Therefore, when designing a message payload, developers must consider the header overhead introduced by each protocol layer when designing payload structures.

3.2 Fragmentation and Its Implications

This issue is commonly found in practical networking scenarios. Whatever the cause, whenever the Internet Protocol tries to transmit a packet over a network segment with an MTU too small to accommodate it, the layer will perform fragmentation. This means that the packet is split into smaller fragments, each of which will be reassembled at the destination.

However, fragmentation presents many drawbacks:

- Increased latency due to processing overhead.
- Higher risk of failure: if one fragment is lost, the entire packet is discarded.
- Security vulnerabilities: attackers can exploit fragmentation in techniques like overlapping fragments to evade intrusion detection systems.

Modern network practices favour avoiding fragmentation altogether. Techniques such as Path MTU Discovery [5] (PMTUD) allow a system to determine the smallest MTU along the route and adjust packet sizes dynamically.

3.3 Impact on Application and Protocol Design

Understanding MTU constraints is essential for software engineers building networked applications. For example:

- Developers must carefully construct UDP payloads to avoid fragmentation, particularly because UDP does not guarantee retransmission.
- Applications transmitting large files (e.g., FTP, video streaming) should implement chunking mechanisms at the application layer rather than relying on IP fragmentation.
- Real-time protocols, such as VoIP, must account for jitter and packet loss — which is made worse by fragmentation.

Engineers also need to understand encapsulation overhead: using protocols like HTTPS over TCP over IP can significantly reduce the effective payload size. This is especially relevant in IoT or wireless environments, where bandwidth is limited.

3.4 OSI Model Context

Within the OSI model, MTU is typically associated with the Data Link Layer (Layer 2), while fragmentation occurs at the Network Layer (Layer 3). PDUs traverse through all layers, accumulating or shedding headers at each stage. A robust understanding of these interactions enables developers to:

- Debug complex network issues using tools like Wireshark [10].
- Design protocols with appropriate payload sizes.
- Create applications resilient to network variability.

4. Synchronicity and Asynchronicity in Network Protocols

When the choice of network programming is synchronous, or asynchronous, it can dramatically impact application performance, responsiveness, and scale. The two paradigms

of interaction scheduling are decided by how to handle things such as sending or receiving data--especially when waiting for network responses. Developers should understand that efficient systems must handle waiting gracefully. With clever design practices, it's possible to create lightweight network applications that operate reliably over long periods with minimal maintenance.

4.1 Synchronous Communication

With synchronous communication, the sender waits (blocks) for an operation to complete. For instance, in socket programming a synchronous `recv()` call halts execution until data has been received. This model is easy to implement and understand, making it suitable for applications that have predictable, linear workflows:

- File transfers (e.g., FTP)
- Email clients (e.g., SMTP or POP3)
- Batch-processing systems.

Synchronous models can lead to I/O blocking, where threads remain idle while waiting for data. In a single-threaded system, this can cause performance bottlenecks or an entire application to freeze.

4.2 Asynchronous Communication

Operations like reading or writing from a socket in asynchronous communication return immediately. The actual result is processed later through callbacks, event loops or signals. In this way of adapting non-blocking applications, the program has the freedom to keep running other tasks while waiting for I/O operations to be completed.

Asynchronous models are ideal for:

- Real-time communication (e.g., chat apps)
- High-performance servers (e.g., HTTP, game servers)
- GUI applications where blocking the main thread would freeze the interface.

A synchronous communication usually entails a more complex architecture (such as event-driven loops) but offers improved throughput and scalability--both vital in handling thousands of concurrent connections.

4.3 I/O Dispatch Models

The mechanism for handling asynchronous operations is typically referred to as an I/O dispatch model. Common models include:

- Thread-per-connection: A separate thread manages each connection. Simple to implement but scales poorly due to thread overhead.
- Event-driven (Reactor pattern): Uses `select()`, `poll()`, or `epoll()` [7] (Linux) to monitor multiple sockets and respond to events without blocking.
- Proactor pattern [7]: The OS completes I/O operations in the background and notifies the application upon completion.

Modern network frameworks (e.g., Boost.Asio [8] in C++, libuv, or Node.js) use event-driven models for efficiency, particularly when building scalable, non-blocking servers.

4.4 Design Implications for Developers

Choosing between synchronous and asynchronous models influences everything from architecture to user experience:

- Responsiveness: Asynchronous I/O prevents UI lockups in desktop or mobile applications.
- Scalability: Large-scale systems (e.g., microservices, REST APIs) require async I/O to efficiently manage resources.
- Complexity: Asynchronous systems require careful handling of race conditions, shared state, and error propagation.

In C++, engineers can combine multi-threading with non-blocking sockets to strike a balance between responsiveness and control. Libraries like Boost.Asio or newer standards like C++20's coroutines provide tools to manage this complexity elegantly.

5. Interoperability in Networked Applications

Interoperability involves systems, applications and components that are often built using different platforms or technologies, as well as sharing data effectively.

The term "interoperability," is crucial to ensure that systems can work together both independently and in cooperation with one another in complex distributed environments for example the internet and cross-platform mobile applications.

5.1 What Interoperability Means

Interoperability means that systems which use different programming languages, frameworks, or platforms can communicate effectively via the network.

For example, a server program written in C++ should be able to communicate with a client written in Python provided they both follow a common protocol and data format agreed upon in advance by all parties concerned. This is particularly relevant in today's computing environments, where services travel between a variety of operating systems and hardware architectures as well as across multiple cloud providers.

Interoperability can be broadly divided into two main categories: horizontal and vertical. Horizontal interoperability refers to the ability of systems or services on the same architectural level for example peer microservices in a distributed system, to work together seamlessly.

Vertical interoperability concerns interaction between different layers within a software architecture, for example between a front-end client app and its back-end server. Both types are essential to building scalable and maintainable networked systems.

5.2 Standards and Protocols

Interoperability is primarily achieved through adherence to open standards and well-defined interfaces. Examples include:

- HTTP/HTTPS for web communication
- JSON [9] or XML for data serialization
- REST or gRPC for APIs.
- UTF-8 for character encoding

By using these standardized technologies, developers ensure that their applications can interact with others, even when developed independently or maintained by different organizations.

5.3 Techniques to Achieve Interoperability

To support interoperability, developers should:

- Use language-neutral formats such as JSON, Protobuf, or XML for data exchange.
- Design RESTful [12] APIs that use clear, consistent URI structures and HTTP methods.
- Avoid platform-specific dependencies or undocumented behaviour.
- Version APIs to manage backward compatibility when changes occur.
- Implement robust error handling and response codes to aid integration and debugging.

Interface Definition Languages (IDLs), such as those used in gRPC or Thrift, provide an additional layer of abstraction, allowing code generation across multiple languages from a single specification.

5.4 Relevance to Software Engineering

For networked applications, poor interoperability can lead to vendor lock-in, integration failures, and maintenance challenges. Conversely, good interoperability enables:

- System integration in large organisations
- Reusability of components
- Extensibility for future development
- Third-party development (e.g., plugins, external clients)

When designing networked systems, engineers must weigh interoperability against performance. For instance, while JSON is human-readable and widely supported, binary formats like Protobuf offer better efficiency but require more effort to implement and maintain.

6. Conclusion

Modern networked applications demand proficiency in not only programming but also a deep grasp of the underlying communication protocols and architectural principles. Several fundamental ideas in network communication have been reviewed by this report, beginning with TCP's three-way handshake at an effort to solve coordination challenges illustrated by the Two Generals' Problem. It highlighted how TCP's mechanisms, though not perfect, do at least provide a robust framework for real-world applications to depend on to deliver data intact across networks.

We also considered how system throughput and efficiency were influenced by technical parameters such as the Maximum Transmission Unit (MTU) and Protocol Data Units (PDUs). This has implications for performance and error handling. The distinction between synchronous and asynchronous communication went on to show how each model is suitable for different application types and system needs, especially at the levels of thread management or I/O dispatch.

Finally, the report discussed interoperability and stressed once again the significance of open standards and coherent interfaces in making it possible that diverse systems can communicate seamlessly with one another.

These topics are not remote concepts but concrete concerns for a developer of software. IT should form the basis for decision-making in terms of application design, scalability, and user experience. Follow these principles carefully and you can create a solid, efficient interconnected system which will be able to meet the needs of today's multifaceted digital environments.

7. References

- [1] J. Postel, "Transmission Control Protocol," RFC 793, IETF, 1981. <https://datatracker.ietf.org/doc/html/rfc793> [Accessed: 02/04/25]
- [2] J. Postel, "User Datagram Protocol," RFC 768, IETF, 1980. <https://datatracker.ietf.org/doc/html/rfc768> [Accessed: 02/04/25]
- [3] S. Floyd and T. Henderson, "The NewReno Modification to TCP's Fast Recovery Algorithm," RFC 3782, IETF, 2004. <https://datatracker.ietf.org/doc/html/rfc3782> [Accessed: 02/04/25]
- [4] J. Halpern and R. Moses, "Knowledge and common knowledge in a distributed environment," Journal of the ACM, vol. 37, no. 3, pp. 549–587, 1990.
- [5] J. Mogul and S. Deering, "Path MTU Discovery," RFC 1191, IETF, 1990. <https://datatracker.ietf.org/doc/html/rfc1191> [Accessed: 05/04/25]
- [6] A. S. Tanenbaum and D. J. Wetherall, *Computer Networks*, 5th ed., Pearson, 2010.
- [7] D. Schmidt and S. Vinoski, "The Reactor and Proactor Design Patterns," C++ Report, vol. 10, no. 7, 1998.
- [8] Boost C++ Libraries, "Boost.Asio Documentation," [Online]. Available: https://www.boost.org/doc/libs/1_82_0/doc/html/boost_asio.html [Accessed: 15/04/25]
- [9] T. Bray et al., "The JavaScript Object Notation (JSON) Data Interchange Format," RFC 8259, IETF, 2017. <https://datatracker.ietf.org/doc/html/rfc8259> [Accessed: 17/04/25]
- [10] Wireshark Foundation, "Wireshark User Guide," [Online]. Available: <https://www.wireshark.org/docs/> [Accessed: 20/04/25]
- [11] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000, IETF, 2021. <https://datatracker.ietf.org/doc/html/rfc9000> [Accessed: 26/04/25]
- [12] L. Richardson and S. Ruby, *RESTful Web Services*, O'Reilly Media, 2007.

Ethan Shellem-Prince Assignment 1 Report

GRADEMARK REPORT

FINAL GRADE

68/100

GENERAL COMMENTS

A reasonable report, fairly well-written, though far too reliant on bullet points, and thus lacking a little depth in some areas. You may also like to consider the sources selected - I get the sense these are primarily from a web search rather than use of a library/journal catalogue, and the report has therefore missed a few key sources I would have expected to see.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11