[5IBRE-1] Basis of Networks

Raphaëlle Wats

December 13, 2025

Contents

1 Abstract

This document presents the design and implementation of a command-line interface (CLI) for a server and a corresponding client application that communicate through structured JSON messages.

The proposed architecture emphasizes modularity, scalability, and robustness. This document details the communication protocol, message framing strategy, and key implementation choices, highlighting how the system ensures correct message parsing, synchronization, and error handling.

2 Server

The server is written in **Python 3.14.2** and relies exclusively on Python's built-in standard library. While raw performance is a critical concern and Python is not traditionally considered a high-performance language, its ease of prototyping and expressive syntax enable rapid development and clear architectural design.

To mitigate performance limitations, the implementation makes extensive use of Python built-in features and standard library components that are internally implemented in optimized `C` code.

Furthermore, the architectural patterns can be directly adapted in production-grade implementations written in other languages that follow similar programming paradigms.

2.1 Python's built-in standard "technosystem"

- **json**:
  This module allows conversion into JSON-formatted string representation and fully complies with the **JavaScript Object Notationstandard**, ensuring **interoperability**.

- **socket**:
  This module provides a high-level interface to the underlying operating system's socket **API**. It abstracts platform-specific details, enabling **portable** and consistent network communication across different operating systems.

- **selectors**:
  This module provides a high-level and efficient I/O multiplexing abstraction. Its **DefaultSelector** class selects the most efficient **I/O multiplexing** mechanism available on the host platform. Its purpose is to monitor registered **file descriptors** and react to **readiness events**. Sockets are file descriptors and thus can be registered when configured in **non-blocking** mode, enabling an **asynchronous event-driven server architecture**.

2.2 Server's Core

```
def run(self):
    while self.running:
        # HANDLE NETWORK
        events = self.selector.select()
        for key, mask in events:
            sess = key.data
            if sess is None:
                self.handle_accept()
            else:
                if mask & selectors.EVENT_READ:
                    self.handle_tcpread(sess)
                if mask & selectors.EVENT_WRITE:
                    self.handle_tcpwrite(sess)
```

The call to **selector.select()** blocks the loop until one or more events are triggered by the operating system, which allows the server to remain **idle** without busy-waiting, consuming essentially zero CPU while waiting for I/O events.

This design follows the principles of event-driven multiplexing using **non-blocking** I/O. By handling multiple connections within a single thread, it allows the server to scale better without the overhead of **spawning** a large number of **OS threads** that induce heavy **context-switching** and **cache-miss** for the **CPU**. In Python (CPython), **the Global Interpreter Lock (GIL)** prevents multiple threads from executing Python bytecode in parallel. True **parallelism** is only possible when the GIL is temporarily released, such as during I/O operations or in C extensions that explicitly release the GIL

By contrast, this non-blocking, event-driven approach avoids these issues: the server waits for I/O readiness using the operating system's event notification mechanisms (e.g., select, poll, epoll, kqueue) without **busy-waiting**, keeping CPU usage minimal when idle. This architecture was chosen rather than a traditional multi-threaded approach that performs best when the language's runtime provides **lightweight threads** which are scheduled entirely in **user space** by the runtime (e.g., Erlang's BEAM or Oz's Mozart runtime), without relying on operating system **signals** or **kernel** threads. Python, however, does not natively support lightweight threads.

Python's **asyncio** library uses a similar mechanism under the hood, demonstrating that this pattern is both practical and efficient for high-concurrency applications.

2.3 Server Flow and Message Passing

This section describes the sequence of operations that occur when a client sends a request to the server. The goal is to provide a clear, step-by-step overview of how incoming data is received, processed, and ultimately dispatched to the appropriate application logic.

2.3.1 In the beginning was the word

Whenever a client attempts to connect to the server, the operating system marks the server's listening TCP socket as ready to accept a new connection. This generates a read-readiness event on the listening socket, causing the selector to wake up. The server then calls **accept()** to establish the connection. The newly accepted socket is configured in non-blocking mode and registered with the selector which subsequently monitors it for future read events.

Each time a connected client sends data, the operating system signals that the associated socket is readable. The selector wakes up the `handle_tcpread()` method is called and processes the incoming bytes.

Let's now describe the application's internal protocol used to handle TCP message traffic. When a client send a message to the server, they invokes the following method:

```
self.app.send_tcpmsg({
    TcpMsg.TYPE: TcpMsg.LOGIN,
    TcpMsg.DATA: {
        "username": "Roger",
        "password": "Rabbit"
    }
})
```

Before a structured application-level message can be transmitted over a TCP connection, it must be converted into a linear sequence of bytes. This process is known as ***serialization*** and is fully reversible through the corresponding ***deserialization*** process performed by the end user.

How does this high-level, structured **syntax sugar-free** abstraction ultimately get turned into **raw bytes frames** to be sent. Without commenting the quality of the chosen password, one might reasonably ask: ***Who framed Roger Rabbit?***

2.3.2 Serialization: *Who Framed Roger Rabbit?*

The application protocol uses a length-prefixed framing strategy in which each message is transmitted as a contiguous byte stream composed of a fixed-size header followed by a variable-size payload. The header consists of a 2-byte unsigned integer encoded in network byte order (big-endian) and specifies the length of the payload that follows. Message serialization is performed by converting structured data into a JSON string, which is then encoded into bytes using the UTF-8 codec. The length of this encoded payload is written into the header, and both parts are concatenated to form the complete message.

```
import json
from shared.constants import TcpMsg

def serialize_msg(msg):
    serialized = bytearray()
    encoded = json.dumps(msg).encode()
    header = len(encoded).to_bytes(2, 'big')
    serialized.extend(header)
    serialized.extend(encoded)
    return serialized
```

Deserialization follows the same structure in reverse: the receiver first reads the length prefix to determine the expected payload size, then collects the corresponding number of bytes, decodes them from UTF-8, and deserializes the JSON content to reconstruct the original data structure.

Nevertheless, due to the stream-oriented nature of TCP and the possibility of partial reads, the receiving endpoint cannot always deserialize incoming data upon reception. Instead, received bytes are **buffered** then the application extract all complete messages according to the framing format, removing each successfully reconstructed message from the buffer. Once extracted, messages are forwarded to the **Dispatcher**, which is responsible for validating their structure and semantic correctness ensuring that messages are not **malformed** or **malicious** before invoking the corresponding application-level handlers.

2.3.3 Who pull request to patch this request dispatcher ?

Let's now describe the Dispatcher's design pattern, each supported message type is represented by a **symbolic** constant (e.g., **ROOM_JOIN**) and is associated with a dedicated **handler function** implementing the corresponding **behavior**.

```
class Dispatcher:
    def __init__(self, server):
        self.server = server
        self.handlers = {
            TcpMsg.ROOM_JOIN: room_join,
            TcpMsg.ROOM_ECHO: room_echo,
            TcpMsg.ROOM_LIST: room_list
        }

    def dispatch(self, sess, msg):
        msgtype = msg.get(TcpMsg.TYPE)
        if msgtype is None:
            return

        handler = self.handlers.get(msgtype)
        if handler:
            handler(self.server, sess, msg)
```

Upon receiving a message, the dispatcher extracts the message type field, retrieves the corresponding handler from the **lookup table**, and invokes it with the necessary execution context.

This approach makes the system easily extensible: supporting a new message type only requires defining a new constant and handler function and registering it in the dispatcher, without modifying the dispatching logic.

3 Client

The client application is implemented using the **Textual** Python framework, a modern toolkit for building terminal user interfaces. In recent years, there has been a noticeable rise in popularity for such frameworks, often referred to as *modern TUIs*. The term is largely a trend label for the next evolution in user interfaces, reflecting shifts in front-end development beyond traditional graphical user interfaces (GUIs).

3.1 Modern TUIs: What else ?

Modern TUIs offer several advantages, including efficient remote interaction over protocols such as SSH, enabling rich interfaces even on low-bandwidth connections; flexible styling and layout using CSS-like syntax without requiring HTML or JavaScript knowledge; and integration with web technologies, allowing interfaces to be served directly in a browser.

3.2 Architecture

The client architecture separates network communication from the user interface by leveraging a background **networker** daemon thread and a foreground TUI front-end. The networker adopts a design analogous to the server: it uses a non-blocking TCP socket monitored by a selector. This architecture is flexible and could be extended in the future to support peer-to-peer communication or additional protocols such as UDP for voice or video streaming.

```
1 def _boot_networker(self):
2     self.networker.start()
3     self.dispatcher = self.set_interval(
4         0.05, self._dispatch_tcpmsg
5     )
6     self.push_screen(RoomScreen())
```

The foreground thread is managed by the **Textual** framework, which internally runs an event loop responsible for rendering the interface and handling user input asynchronously. Once the connection to the server is established, the client starts the networker daemon thread and sets up a periodic interval using Textual's `set_interval` mechanism. The interval drains the thread-safe incoming message queue and dispatches messages to the currently active screen by invoking its `handle_tcpmsg` method. Each screen implements this method, allowing polymorphic handling of messages depending on the active view.

3.3 Conclusion

This separation of concerns ensures that network communication does not block the graphical user interface, while Textual's reactive event loop provides smooth and responsive interaction.

Although the original assignment instruction imply graphical user interface, I've made the decision to implement a modern TUI, as I consider it a next-generation graphical interface paradigm. Having prior experience in front-end GUI development, I wanted to explore a new approach for educational purposes, which aligns with the primary goal of this assignment. I was amazed by this experience and highly recommend considering Textual for future project.

References

[1] T. Bray. The JavaScript object notation (JSON) data interchange format. Technical report, IETF, 2017.

[2] W. Eddy. Transmission Control Protocol (TCP). Technical report, IETF, 2022. The modern standard for TCP, explaining its stream-oriented nature.

[3] OSS Insight. Trends in terminal user interface (TUI) frameworks. `https://ossinsight.io/collections/tui-framework/trends/`, 2025. Accessed: 2025-12-16.

[4] Python Software Foundation. Global interpreter lock (GIL). `https://docs.python.org/3/glossary.html#term-global-interpreter-lock`, 2025. Internal mechanism for thread synchronization in CPython.

[5] Python Software Foundation. *Python 3.14.2 Documentation*. Python Software Foundation, 2025. Standard Library: selectors and socket modules.

[6] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*, volume 2. John Wiley & Sons, 2000. Details the Reactor and Dispatcher patterns for high-performance networking.

[7] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *Unix Network Programming, Volume 1: The Sockets Networking API*. Addison-Wesley Professional, 2004. Reference for I/O multiplexing and select/poll/epoll logic.

[8] Textualize. Textual: Modern TUI framework for Python. `https://www.textualize.io/`, 2025. Accessed: 2025-12-16.

[9] Wikipedia contributors. Protocol data unit — Wikipedia, the free encyclopedia, 2025. Accessed: 2025-12-16. Discusses framing and encapsulation.

[10] F. Yergeau. UTF-8, a transformation format of ISO 10646. Technical report, IETF, 2003.