Technical Report: TCP Server-Client Application Using select()

1. Introduction

This report presents a detailed analysis of a client-server application implementing the TCP protocol with I/O multiplexing using the `select()` system call. The application demonstrates concurrent client handling on a single-threaded server that counts words from multiple clients simultaneously.

1.1 Project Objectives

- Implement a TCP-based communication system
- Handle multiple clients concurrently using `select()`
- Process and count words from client messages
- Maintain a global word count across all connected clients

---

2. TCP Protocol Overview

2.1 What is TCP?

TCP (Transmission Control Protocol) is a connection-oriented protocol that provides reliable, ordered, and error-checked delivery of data between applications. It operates at the Transport Layer (Layer 4) of the OSI model.

2.2 TCP Key Characteristics

Connection-Oriented: TCP establishes a connection before data transmission through a three-way handshake:

1. Client sends SYN (synchronize)
2. Server responds with SYN-ACK (synchronize-acknowledge)
3. Client sends ACK (acknowledge)

Reliability: TCP ensures data delivery through:

- Acknowledgment mechanisms
- Retransmission of lost packets
- Sequence numbering for proper ordering

Flow Control: TCP uses sliding window protocol to manage data flow between sender and receiver.

Error Detection: Checksums verify data integrity during transmission.

---

3. The select() System Call

3.1 Why Use select()?

Traditional blocking I/O forces a process to wait for a single operation to complete. In server applications handling multiple clients, this creates a problem: while waiting for data from one client, the server cannot service others.

Solutions:

- **Multiple processes/threads**: Resource-intensive
- **Non-blocking I/O + polling**: CPU-intensive, inefficient
- **I/O multiplexing (select)**: Monitor multiple file descriptors efficiently

3.2 How select() Works

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
          fd_set *exceptfds, struct timeval *timeout);
```

Parameters:

- `nfds` : Highest file descriptor number + 1
- `readfds` : Set of file descriptors to monitor for reading
- `writefds` : Set for writing (not used in our application)
- `exceptfds` : Set for exceptions (not used)
- `timeout` : Maximum wait time (NULL = block indefinitely)

Return Value: Number of ready file descriptors, 0 on timeout, -1 on error

3.3 File Descriptor Sets

Four macros manipulate `fd_set`:

```
FD_ZERO(&set);        // Clear all file descriptors from set
FD_SET(fd, &set);     // Add fd to set
FD_CLR(fd, &set);     // Remove fd from set
FD_ISSET(fd, &set);   // Test if fd is in set
```

3.4 select() Operation Flow

1. **Initialize**: Create a master set of all file descriptors to monitor
2. **Copy**: Before each `select()` call, copy master set (select modifies it)
3. **Block**: `select()` blocks until at least one descriptor is ready
4. **Check**: Iterate through descriptors using `FD_ISSET()` to find ready ones
5. **Process**: Handle ready descriptors (accept connections or read data)

---

4. Server Architecture

4.1 Server Initialization

```
server_fd = socket(AF_INET, SOCK_STREAM, 0);
```

- **AF_INET**: IPv4 protocol family
- **SOCK_STREAM**: TCP socket type (connection-oriented, reliable)

```
server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
server_addr.sin_port = htons(PORT);
```

- Binds to localhost (127.0.0.1) on port 1234
- `htons()`: Converts port to network byte order (big-endian)

4.2 Main Server Loop

The server operates in an infinite loop with the following logic:

Step 1: Copy Master Set

```
read_fds = master_fds;
```

Essential because `select()` modifies the fd_set to indicate ready descriptors.

Step 2: Wait for Activity

```
select(max_fd + 1, &read_fds, NULL, NULL, NULL);
```

Blocks until at least one file descriptor has data to read.

Step 3: Identify Ready Descriptors

```
for (int fd = 0; fd <= max_fd; fd++) {
    if (FD_ISSET(fd, &read_fds)) {
        // Handle this descriptor
    }
}
```

Step 4: Process Activity

Two cases exist:

Case A: New Connection (fd == server_fd)

```
client_fd = accept(server_fd, ...);
FD_SET(client_fd, &master_fds);
if (client_fd > max_fd) max_fd = client_fd;
```

- Accept new connection

- Add client socket to master set
- Update max_fd if necessary

Case B: Client Data (fd != server_fd)

```
read_size = recv(fd, buffer, sizeof(buffer) - 1, 0);
```

- Read data from connected client
- If `read_size <= 0` : Client disconnected, close socket and remove from set
- Otherwise: Process message and send response

4.3 Word Counting Logic

```
int countWords(char *str) {
    int count = 0;
    char *t = strtok(str, " ,;:");
    while (t != NULL) {
        count++;
        t = strtok(NULL, " ,;:");
    }
    return count;
}
```

This function tokenizes strings using delimiters (space, comma, semicolon, colon) and counts the tokens. The server maintains a `totalWords` counter accumulating counts from all clients.

---

5. Client Architecture

5.1 Connection Establishment

```
dfs_client = socket(AF_INET, SOCK_STREAM, 0);
connect(dfs_client, (struct sockaddr *)&adresse_serveur, ...);
```

The client creates a TCP socket and initiates connection to the server at 127.0.0.1:1234.

5.2 Communication Loop

```
while (1) {
    fgets(buffer, sizeof(buffer), stdin);  // Read user input
    send(dfs_client, buffer, strlen(buffer), 0);  // Send to server
    recv(dfs_client, serverReply, sizeof(serverReply), 0);  // Receive response
    printf("Server response: %s\n", serverReply);
}
```

The client operates in a simple request-response pattern:

1. Read input from user
2. Send message to server
3. Wait for server response
4. Display response

---

6. Communication Flow

6.1 Message Exchange Sequence

```
Client 1                Server              Client 2
   |                      |                     |
   |---- "hello" --------->|                     |
   |                   [count=1]                |
   |<-- "Words: 1" --------|                     |
   |                      |<----- "world" -------|
   |                   [count=2]                |
   |                      |---- "Words: 1" ----->|
   |---- "test" ---------->|                     |
   |                   [count=3]                |
   |<-- "Words: 1" --------|                     |
```

6.2 Data Flow Analysis

Server Perspective:

- Maintains single `totalWords` variable
- Each message increments global counter
- All clients see cumulative total in responses

Client Perspective:

- Simple synchronous communication
- Blocks waiting for server response after each send
- No awareness of other clients

---

7. Advantages of This Architecture

7.1 Using select()

Efficiency: Single process handles multiple clients without creating threads **Scalability**: Can manage hundreds of clients with minimal overhead **Simplicity**: No thread synchronization required **Deterministic**: Easier to debug than multi-threaded applications

7.2 Using TCP

Reliability: Guaranteed message delivery **Order Preservation**: Messages arrive in sent order **Error Detection**: Built-in checksums **Flow Control**: Prevents overwhelming receiver

---

8. Limitations and Improvements

8.1 Current Limitations

Buffer Issues: Fixed 1024-byte buffer may truncate large messages **No Error Recovery**: Server crash loses all state **Single-threaded**: CPU-intensive operations block all clients **No Authentication**: Any client can connect **Scalability Ceiling**: `select()` limited to FD_SETSIZE (typically 1024) file descriptors

8.2 Potential Improvements

Use poll() or epoll(): Better scalability for thousands of connections **Add Persistence**: Save word count to database or file **Implement Protocol**: Define message format (e.g., JSON, Protocol Buffers) **Add Security**: Implement TLS/SSL encryption **Handle Partial Reads**: TCP is stream-based, may need multiple recv() calls **Graceful Shutdown**: Signal handling for clean server termination **Per-Client Tracking**: Maintain separate statistics for each client

---

9. Testing Scenarios

9.1 Single Client Test

Procedure:

1. Start server
2. Connect one client
3. Send "hello world"
4. Verify response: "Words in this message: 2 | Total words (all clients): 2"

9.2 Multiple Client Test

Procedure:

1. Start server
2. Connect Client A, send "one two three" → Total: 3
3. Connect Client B, send "four five" → Total: 5
4. Client A sends "six" → Total: 6
5. Verify cumulative counting works correctly

9.3 Disconnect Test

Procedure:

1. Connect multiple clients
2. Type "exit" on one client
3. Verify server continues serving remaining clients
4. Check server logs for disconnection message

---

10. Conclusion

This application successfully demonstrates TCP socket programming with I/O multiplexing using `select()`. The server efficiently handles multiple concurrent clients on a single thread, maintaining shared state (total word count) across all connections.

Key Takeaways:

- `select()` enables concurrent I/O handling without threading
- TCP provides reliable, connection-oriented communication
- File descriptor management is critical for proper operation
- Simple architecture scales well for moderate client counts

Learning Outcomes:

- Understanding TCP three-way handshake and connection management
- Mastery of `select()` for multiplexed I/O operations
- Socket programming fundamentals (socket, bind, listen, accept, recv, send)
- Client-server architecture design patterns

This foundation prepares for more advanced topics like asynchronous I/O (epoll, kqueue), message protocols, and distributed systems design.

---

11. References

- Stevens, W. R., Fenner, B., & Rudoff, A. M. (2003). *UNIX Network Programming, Volume 1: The Sockets Networking API*
- POSIX.1-2008 Standard - select() specification
- RFC 793 - Transmission Control Protocol
- Linux man pages: socket(2), select(2), tcp(7)