TCP Server-Client Report Using select()

1. Introduction

This report analyzes a client-server application using TCP protocol with `select()` for I/O multiplexing. The server handles multiple clients concurrently, counting words from their messages and maintaining a global counter.

---

2. TCP Protocol

What is TCP?

TCP (Transmission Control Protocol) is a connection-oriented protocol providing reliable, ordered data delivery at the Transport Layer.

Key Features

- **Connection-Oriented**: Three-way handshake (SYN → SYN-ACK → ACK)
- **Reliable**: Acknowledgments and retransmissions ensure delivery
- **Ordered**: Data arrives in sequence
- **Error Checking**: Checksums verify integrity

---

3. The select() System Call

Why Use select()?

Traditional blocking I/O can only handle one client at a time. `select()` allows monitoring multiple file descriptors simultaneously without threads.

How It Works

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

Key Operations:

- `FD_ZERO(&set)` - Clear set
- `FD_SET(fd, &set)` - Add descriptor
- `FD_ISSET(fd, &set)` - Check if ready
- `FD_CLR(fd, &set)` - Remove descriptor

Operation Flow

1. Maintain master set of all file descriptors
2. Copy master set (select modifies it)

3. Block until activity detected
4. Check which descriptors are ready
5. Process ready descriptors

---

4. Server Implementation

Initialization

```
server_fd = socket(AF_INET, SOCK_STREAM, 0);  // Create TCP socket
bind(server_fd, ...);                          // Bind to 127.0.0.1:1234
listen(server_fd, 5);                          // Start listening
```

Main Loop Logic

1. Wait for Activity

```
read_fds = master_fds;  // Copy master set
select(max_fd + 1, &read_fds, NULL, NULL, NULL);  // Block until ready
```

2. Check All Descriptors

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

3. Handle Two Cases

New Connection (fd == server_fd):

```
client_fd = accept(server_fd, ...);
FD_SET(client_fd, &master_fds);  // Add to monitoring set
```

Client Data (fd != server_fd):

```
recv(fd, buffer, ...);  // Read message
// If recv <= 0: client disconnected, close and remove
// Otherwise: count words, update total, send response
```

Word Counting

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

Tokenizes string and counts words. Server maintains `totalWords` across all clients.

---

5. Client Implementation

Connection

```
dfs_client = socket(AF_INET, SOCK_STREAM, 0);
connect(dfs_client, ...);  // Connect to 127.0.0.1:1234
```

Communication Loop

```
while (1) {
    fgets(buffer, stdin);              // Read input
    send(dfs_client, buffer, ...);     // Send to server
    recv(dfs_client, serverReply, ...); // Wait for response
    printf("Server response: %s\n", serverReply);
}
```

Simple request-response pattern: send message → wait for reply → display.

---

6. Communication Flow Example

```
Client A          Server (totalWords)          Client B
   |                    |                          |
   |--- "hello" ------->| count: 1                 |
   |<-- "Words: 1" -----| total: 1                 |
   |                    |<------ "world test" -----|
   |                    | count: 2, total: 3       |
   |                    |------- "Words: 2" ------>|
   |--- "foo" ---------->| count: 1                |
   |<-- "Words: 1" -----| total: 4                 |
```

---

7. Advantages

select() Benefits:

- Single process handles multiple clients
- No threading complexity
- Efficient: waits for activity instead of polling
- Easier to debug than multi-threaded code

TCP Benefits:

- Guaranteed delivery
- Ordered messages
- Built-in error detection

---

8. Limitations & Improvements

Current Limitations

- Fixed buffer size (1024 bytes)
- No persistence (data lost on crash)
- Single-threaded (blocks on CPU-intensive tasks)
- Limited to ~1024 connections (FD_SETSIZE)

Possible Improvements

- Use `epoll()` or `poll()` for better scalability
- Add database for persistence
- Implement authentication
- Handle partial TCP reads properly
- Add graceful shutdown with signal handling

---

9. Testing

Single Client: Connect, send "hello world", verify count = 2

Multiple Clients:

- Client A: "one two" → total: 2
- Client B: "three" → total: 3
- Verify cumulative counting

Disconnection: Exit one client, verify others continue working

---

10. Conclusion

This application demonstrates TCP socket programming with I/O multiplexing using `select()`. The server efficiently manages multiple concurrent clients without threading, maintaining shared state across all

connections.

Key Learnings:

- `select()` enables single-threaded concurrent I/O
- TCP provides reliable connection-oriented communication
- Proper file descriptor management is essential
- Architecture scales well for moderate client counts

Skills Gained:

- Socket API (socket, bind, listen, accept, recv, send)
- I/O multiplexing with select()
- Client-server design patterns
- TCP protocol understanding