A Robust Timeout Aware Socket Server With Proper Socket Configuration

1 Objective

To identify, analyze, and resolve the EAGAIN/EWOULDBLOCK error in the socket `accept()` operation by implementing proper socket configuration management between listening and connected sockets.

2 Problem Statement

Develop a concurrent TCP server that can handle multiple client connections simultaneously while properly managing socket configurations to avoid common pitfalls such as:

- **Premature Connection Timeouts**: The server incorrectly applies receive timeouts (`SO_RCVTIMEO`) to listening sockets, causing the `accept()` function to fail with `EAGAIN/EWOULDBLOCK` errors even when the server is functioning normally.
- **Inconsistent Socket Behavior**: Lack of clear separation between configuration requirements for listening sockets versus connected client sockets leads to unreliable server operation.
- **Poor Error Recovery**: Inadequate handling of temporary resource availability and timeout scenarios results in unnecessary server disruptions.

3 Code

3.1 Server Side Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <errno.h>
#include <signal.h>
#include <sys/time.h>

#define PORT 8080
#define BACKLOG 10
#define BUFFER_SIZE 1024

void configure_socket_options(int server_fd) {
```

```c
    int opt = 1;
    struct timeval timeout;

    // SO_REUSEADDR - Allow reuse of local addresses
    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)) < 0) {
        perror("setsockopt SO_REUSEADDR failed");
        exit(EXIT_FAILURE);
    }
    printf("SO_REUSEADDR enabled\n");

    // SO_REUSEPORT - Allow reuse of local ports (Linux specific)
    #ifdef SO_REUSEPORT
    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEPORT, &opt, sizeof(opt)) < 0) {
        perror("setsockopt SO_REUSEPORT failed");
    } else {
        printf("SO_REUSEPORT enabled\n");
    }
    #endif

    // SO_RCVTIMEO - Set receive timeout
    timeout.tv_sec = 5;  // 5 seconds
    timeout.tv_usec = 0;
    if (setsockopt(server_fd, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof(timeout)) < 0) {
        perror("setsockopt SO_RCVTIMEO failed");
    } else {
        printf("SO_RCVTIMEO set to 5 seconds\n");
    }

    // SO_SNDTIMEO - Set send timeout
    timeout.tv_sec = 5;  // 5 seconds
    timeout.tv_usec = 0;
    if (setsockopt(server_fd, SOL_SOCKET, SO_SNDTIMEO, &timeout, sizeof(timeout)) < 0) {
        perror("setsockopt SO_SNDTIMEO failed");
    } else {
        printf("SO_SNDTIMEO set to 5 seconds\n");
    }

    // TCP_NODELAY - Disable Nagle's algorithm for lower latency
    opt = 1;
    if (setsockopt(server_fd, IPPROTO_TCP, TCP_NODELAY, &opt, sizeof(opt)) < 0) {
        perror("setsockopt TCP_NODELAY failed");
    } else {
        printf("TCP_NODELAY enabled\n");
    }
}

void handle_client(int client_fd, struct sockaddr_in *client_addr) {
    char buffer[BUFFER_SIZE];
    char client_ip[INET_ADDRSTRLEN];
    int bytes_received;

    inet_ntop(AF_INET, &(client_addr->sin_addr), client_ip, INET_ADDRSTRLEN);
```

```
    printf("Client connected from %s:%d\n", client_ip, ntohs(
        client_addr->sin_port));

    while (1) {
        // Clear buffer
        memset(buffer, 0, BUFFER_SIZE);

        // Receive data from client
        bytes_received = recv(client_fd, buffer, BUFFER_SIZE - 1, 0);

        if (bytes_received > 0) {
            buffer[bytes_received] = '\0';
            printf("Received from client: %s", buffer);

            // Check for exit command
            if (strncmp(buffer, "exit", 4) == 0) {
                printf("Client requested disconnect\n");
                break;
            }

            // Send response back to client
            char response[BUFFER_SIZE];
            const char *prefix = "Server received: ";
            size_t prefix_len = strlen(prefix);
            if (prefix_len >= sizeof(response)) {
                response[0] = '\0';
            } else {
                int max_copy = (int)(sizeof(response) - prefix_len - 1)
                    ;
                if (max_copy < 0) max_copy = 0;
                int written = snprintf(response, sizeof(response), "%s
                    %.*s", prefix, max_copy, buffer);
                (void)written;
            }
            if (send(client_fd, response, strlen(response), 0) < 0) {
                perror("send failed");
                break;
            }
        } else if (bytes_received == 0) {
            printf("Client disconnected\n");
            break;
        } else {
            if (errno == EAGAIN || errno == EWOULDBLOCK) {
                printf("Receive timeout occurred\n");
                char *timeout_msg = "Server timeout - no data received\
                    n";
                send(client_fd, timeout_msg, strlen(timeout_msg), 0);
                break;
            } else {
                perror("recv failed");
                break;
            }
        }
    }

    close(client_fd);
    printf("Connection with client %s:%d closed\n", client_ip, ntohs(
        client_addr->sin_port));
```

```
}

int main() {
    int server_fd, client_fd;
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_len = sizeof(client_addr);

    // Create socket
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }
    printf("Server socket created\n");

    // Configure socket options
    configure_socket_options(server_fd);

    // Set up server address
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    // Bind socket to address
    if (bind(server_fd, (struct sockaddr *)&server_addr, sizeof(
        server_addr)) < 0) {
        perror("bind failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }
    printf("Socket bound to port %d\n", PORT);

    // Listen for connections
    if (listen(server_fd, BACKLOG) < 0) {
        perror("listen failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }
    printf("Server listening on port %d...\n", PORT);

    // Handle SIGCHLD to avoid zombie processes
    signal(SIGCHLD, SIG_IGN);

    while (1) {
        // Accept incoming connection
        client_fd = accept(server_fd, (struct sockaddr *)&client_addr,
            &client_len);
        if (client_fd < 0) {
            perror("accept failed");
            continue;
        }

        // Fork to handle client in separate process
        pid_t pid = fork();
        if (pid == 0) {
            close(server_fd);
            handle_client(client_fd, &client_addr);
            exit(0);
        } else if (pid > 0) {
```

```
            close(client_fd);
        } else {
            perror("fork failed");
            close(client_fd);
        }
    }

    close(server_fd);
    return 0;
}
```

3.2 Client Side Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <errno.h>
#include <sys/time.h>

#define SERVER_IP "127.0.0.1"
#define PORT 8080
#define BUFFER_SIZE 1024

void configure_client_socket(int sockfd) {
    int opt = 1;
    struct timeval timeout;

    // SO_RCVTIMEO - Set receive timeout
    timeout.tv_sec = 10;
    timeout.tv_usec = 0;
    if (setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof(
        timeout)) < 0) {
        perror("setsockopt SO_RCVTIMEO failed");
    } else {
        printf("Client SO_RCVTIMEO set to 10 seconds\n");
    }

    // SO_SNDTIMEO - Set send timeout
    timeout.tv_sec = 10;
    timeout.tv_usec = 0;
    if (setsockopt(sockfd, SOL_SOCKET, SO_SNDTIMEO, &timeout, sizeof(
        timeout)) < 0) {
        perror("setsockopt SO_SNDTIMEO failed");
    } else {
        printf("Client SO_SNDTIMEO set to 10 seconds\n");
    }

    // TCP_NODELAY - Disable Nagle's algorithm
    opt = 1;
    if (setsockopt(sockfd, IPPROTO_TCP, TCP_NODELAY, &opt, sizeof(opt))
        < 0) {
        perror("setsockopt TCP_NODELAY failed");
    } else {
```

```
43          printf("TCP_NODELAY enabled\n");
44      }
45  }
46
47  int main() {
48      int sockfd;
49      struct sockaddr_in server_addr;
50      char buffer[BUFFER_SIZE];
51      char message[BUFFER_SIZE];
52
53      // Create socket
54      if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
55          perror("socket creation failed");
56          exit(EXIT_FAILURE);
57      }
58      printf("Client socket created\n");
59
60      // Configure client socket options
61      configure_client_socket(sockfd);
62
63      // Set up server address
64      server_addr.sin_family = AF_INET;
65      server_addr.sin_port = htons(PORT);
66
67      if (inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr) <= 0) {
68          perror("invalid address");
69          close(sockfd);
70          exit(EXIT_FAILURE);
71      }
72
73      // Connect to server
74      if (connect(sockfd, (struct sockaddr *)&server_addr, sizeof(
            server_addr)) < 0) {
75          perror("connection failed");
76          close(sockfd);
77          exit(EXIT_FAILURE);
78      }
79      printf("Connected to server %s:%d\n", SERVER_IP, PORT);
80
81      printf("Type messages to send to server (type 'exit' to quit):\n");
82
83      while (1) {
84          printf("Client: ");
85          fflush(stdout);
86
87          // Read input from user
88          if (fgets(message, BUFFER_SIZE, stdin) == NULL) {
89              break;
90          }
91
92          // Send message to server
93          if (send(sockfd, message, strlen(message), 0) < 0) {
94              perror("send failed");
95              break;
96          }
97
98          // Check if user wants to exit
99          if (strncmp(message, "exit", 4) == 0) {
```

```
            printf("Disconnecting from server...\n");
            break;
        }

        // Clear buffer
        memset(buffer, 0, BUFFER_SIZE);

        // Receive response from server
        int bytes_received = recv(sockfd, buffer, BUFFER_SIZE - 1, 0);
        if (bytes_received > 0) {
            buffer[bytes_received] = '\0';
            printf("Server: %s", buffer);
        } else if (bytes_received == 0) {
            printf("Server disconnected\n");
            break;
        } else {
            if (errno == EAGAIN || errno == EWOULDBLOCK) {
                printf("Receive timeout - no response from server\n");
            } else {
                perror("recv failed");
                break;
            }
        }
    }

    close(sockfd);
    printf("Connection closed\n");

    return 0;
}
```

4 Problems Faced During Development Process

- Encountered errors with `struct timeval` and `timeout`, resolved by including `sys/time.h`.
- Faced issues with `TCP_NODELAY` enum, resolved by including `netinet/tcp.h`.
- Addressed a string formatting issue:
  - **Before**:

```
sprintf(response, BUFFER_SIZE, "Server received: %s", buffer);
```

  - **After**:

```
int written = snprintf(response, sizeof(response), "%s%.*s",
    prefix, max_copy, buffer);
(void)written;
```

5 Conclusion

The implementation of the concurrent TCP server successfully addresses the critical socket configuration challenge that initially caused the `EAGAIN/EWOULDBLOCK` errors.

5.1 Key Points

1. **Problem Resolution**: The implementation successfully addresses the critical socket configuration challenge that initially caused the `EAGAIN/EWOULDBLOCK` errors.
2. **Architectural Improvements**:
   - **Modular Design**: Implemented clear separation between `configure_listening_socket()` and `configure_client_socket()` functions.
   - **Proper Resource Management**: Incorporated robust error handling, signal management for zombie processes, and comprehensive cleanup procedures.
   - **Concurrent Handling**: Utilized process forking to efficiently manage multiple simultaneous client connections.