EPL421 HW4 Architecture report

Introduction:

The HTTPS server has been designed and implemented with a focus on configurability, scalability, and secure communication. The architecture leverages multithreading, SSL/TLS encryption, and a custom SSL connection queue to efficiently handle incoming client requests.

Configuration:

The server retrieves its configuration parameters from the "config.txt" file, which includes the number of threads, the port number, and the HTTP home directory containing served resources and storage.

Socket Initialization:

The server creates a TCP socket using the socket system call, followed by binding the socket to the port specified in the configuration file using the bind system call. Subsequently, the server enters listening mode using the listen system call, with the backlog parameter set to the number of threads specified in the configuration file.

SSL Connection Queue:

An SSL connection queue is initialized to facilitate efficient and thread-safe handling of incoming connections. This custom queue structure supports atomic enqueue and dequeue operations. Each node in the queue contains an SSL structure pointer and a file descriptor pointing to an established socket created through the accept system call.

Thread Pool Initialization:

The server initializes a thread pool based on the specified thread count in the configuration file. Threads within the pool await dequeue operations from the SSL connection queue. The implementation incorporates mutexes and condition variables to guarantee thread-safe operations during queue manipulation.

Main Thread Execution:

While worker threads await new connections in the queue, the main thread continuously accepts incoming connections on the designated port. For each accepted connection, the main thread establishes an SSL connection and atomically enqueues the new connection into the SSL connection queue. Subsequently, a waiting worker thread is signaled to dequeue the connection and resume processing.

Worker Thread Functionality:

Upon dequeuing a connection, worker threads handle client requests. The server robustly supports various HTTP methods, including GET, HEAD, POST, and DELETE, on resources defined in the HTTP home directory. This functionality ensures versatile and efficient resource management.

Additional Functionality:

Furthermore, our server implementation is designed to seamlessly interact with a variety of HTTP clients, including popular tools such as curl, Postman, and commercial web browsers. This

comprehensive compatibility ensures that developers and users can effortlessly communicate with the server using their preferred client applications. The server's robust design and adherence to HTTP standards enable it to handle requests from diverse sources, enhancing its versatility and accessibility in different development and testing environments.

Challenges Faced:

The implementation encountered challenges related to synchronization and coordination among threads, particularly during enqueue and dequeue operations in the SSL connection queue. Ensuring the thread-safe manipulation of the queue required careful consideration and testing.

Conclusion:

The HTTPS server's architecture showcases a modular design emphasizing configuration flexibility, secure communication through SSL/TLS, and efficient multithreaded processing. The incorporation of a dedicated thread pool management file and a thread-safe SSL connection queue contributes to a scalable and reliable solution. The server effectively handles diverse HTTP methods for resource management while maintaining a high level of security.