**Reflection on Implementation and Specification Updates**

When I started coding the QUIC streaming protocol, I realized that some of my initial design assumptions did not match reality. First, I thought I could read data directly from a QUIC stream with a simple “read” call. In practice, the aioquic library does not provide a straightforward reader for each stream. Instead, I had to keep a buffer of incoming bytes for every stream. Whenever data arrived, I appended it to that buffer and then checked whether I had enough bytes to form a complete PDU before parsing it. This meant I updated the specification to explain exactly how to manage and parse a byte buffer rather than assuming a high-level read API.

Another issue was timing outgoing packets. My original design said to wait for the connection to be ready after sending each PDU. I expected a method like “wait\_closed” to flush packets immediately. During implementation, I found that this method was either unavailable or did not work as expected on the client side. The simpler solution is to insert a short “await asyncio.sleep(0)” immediately after sending. This pause tells the event loop to process outgoing QUIC frames without blocking for a long time. I amended the specification to recommend using a zero-second sleep rather than relying on a nonexistent API.

Certificate generation also required an update. I assumed that running a basic openssl command would produce a certificate the client could trust. However, modern QUIC clients reject certificates without a Subject Alternative Name (SAN) entry. To fix this, I added instructions in the specification to create a temporary OpenSSL configuration file that explicitly includes SAN entries for “localhost” and “127.0.0.1.” This change ensures the generated certificate works without TLS errors.

Finally, while implementing pause and resume, I discovered that the streaming loop must check the current state before reading each chunk. If the server is paused, the loop must sleep briefly and wait. Otherwise, it can read and send chunks too quickly or close the stream too early. I revised the specification to detail how the server’s streaming loop should pause, sleep for a short time, and then resume when instructed.

In summary, coding the protocol highlighted areas where my initial specification needed more detail. By updating the design to include buffer management, zero-second sleeps, SAN certificate instructions, and precise pause logic, the specification now matches the real implementation. This feedback-driven process improved both my learning and the final protocol document.