**5. Channel Allocation Strategies**

Let’s consider a scenario where we use messaging systems: CQRS (Command Query Responsibility Segregation) applications.

Since the command path never returns any data, the service can execute them asynchronously. In a typical implementation, we have an HTTP POST endpoint that internally builds a message and sends it to a queue for later processing.

**Now, for a service that must handle dozens or even hundreds of concurrent requests, opening connections and channels every time is not a realistic option**. Instead, a better approach is to use a channel pool.

Of course, this leads to the next problem: should we create a single connection and create channels from it or use multiple connections?

**5.1. Single Connection/Multiple Channels**

In this strategy, we’ll use a single connection and just create a channel pool with a capacity equal to the maximum number of concurrent connections the service can manage. For a traditional thread-per-request model, this should be set to the same size as the request handler thread pool.

The downside of this strategy is that, under heavier loads, the fact that we must send commands one at a time through the associated channel implies that we must use a synchronization mechanism. This, in turn, adds extra latency in the command path, which we want to minimize.

**5.2. Connection-per-Thread Strategy**

Another option is to go to the other extreme and use a *Connection* pool, so there’s never contention for a channel. For each *Connection*, we’ll create a single *Channel*that a handler thread will use to issue commands to the server.

However, the fact that we remove synchronization from the client side comes with a cost. The broker must allocate additional resources for each connection, such as socket descriptors and state information. Moreover, the server must split the available throughput between clients.