Software Design

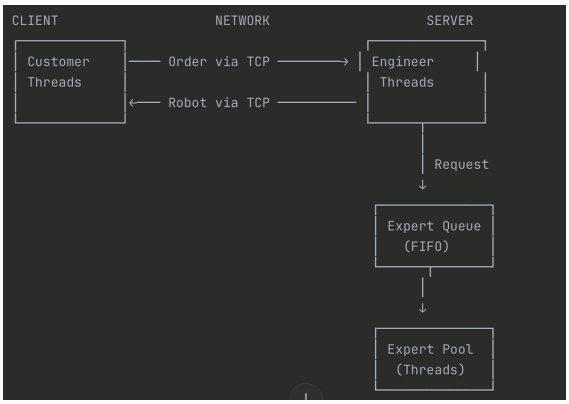
Summary –

This system simulates a robot factory where **Customers** (client threads) place orders and wait for robots, **Engineers** (server threads) assemble robots for customers, **Experts** (thread pool) add special modules to special robots, **Performance** is measured in real-time (latency, throughput).

Design Patterns used –

Remote Procedure Call: where the ClientStub/ServerStub hide network complexity, Order() function looks like local function but executes remotely
Producer-Consumer Design: Engineers produce expert requests, Experts consume from shared queue
Thread Pool: Fixed number of expert threads, Avoids overhead of creating/destroying threads
Promise/Future: Async communication between engineer and expert, Engineer doesn't need to know which expert processes request
Mutexes released automatically with lock_guard
Stub Pattern: Stubs are proxies for remote objects, Hide marshalling/unmarshalling details

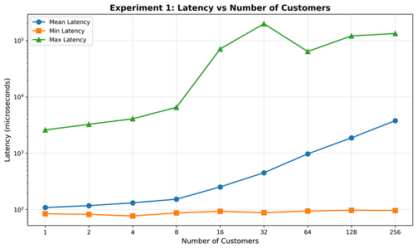
System Architecture –

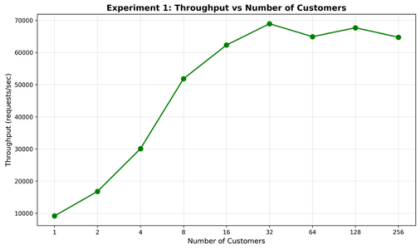


Component Details –

Data Class, Structs Orders and Robot, (Common.h): These classes represent the data structures for orders and robots. They are marshalled and unmarshalled by the stub classes and transferred across the network.
Socket Classes: (SocketComm.h) - TCP socket wrapper for reliable data transmission, Handles partial sends/receives, automatic cleanup. (**ServerSocket)** - used for server-side listening socket management returning a new SocketComm object for each accepted client connection.
Stub Classes: (ClientStub) - High-level client API hiding socket details, Marshals Order to bytes, sends via socket, receives and unmarshals RobotInfo. **(ServerStub) -** High-level server API hiding socket details. Inverse of ClientStub - unmarshals orders, marshals robots
Expert Queue System: (ExpertQueue.h) - Thread-safe FIFO queue for expert engineer thread pool. Synchronization: Mutex + condition variable (blocks when empty, no busy-waiting) (**ExpertRequest) -** Wraps expert work request with async result mechanism
Main Programs (ClientMain, ServerMain): The ServerMain program launches a multi-threaded server with a pool of expert engineers and shuts down on SIGINT (Ctrl+C). The ClientMain program spawns multiple customer threads that connect concurrently to place orders and measure performance.

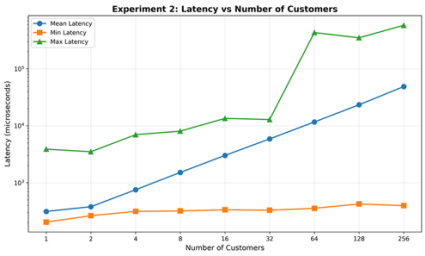
Experiment 1: Regular Robot Type (No Experts)

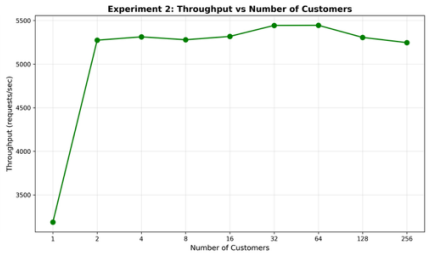




Mean latency increases gradually, Throughput rises quickly until ~32, Max latency grows steeply
The system scales **well up to 32–64 customers**, after which **CPU saturation and context switching** slow down performance.
Latency growth is due to **increased scheduling and socket handling overhead**, not expert contention.
This represents **ideal parallel performance** - every customer has a dedicated engineer, and no shared bottleneck exists.

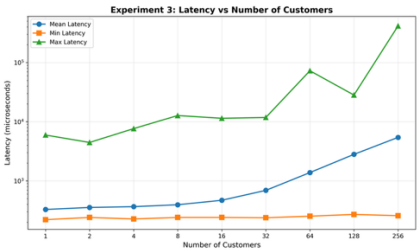
Experiment 2: Special Robot Type (1 Expert Engineer)

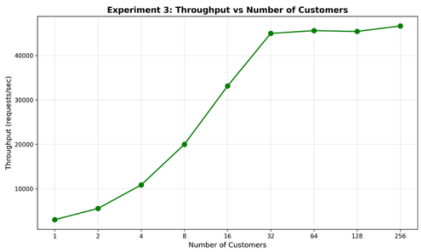




Mean latency skyrockets, Throughput remains nearly flat, max latency spikes indicate queue buildup.
The single expert thread becomes a central bottleneck - all engineers queue requests, waiting for one thread to complete them sequentially.
System becomes serialization-bound, regardless of how many customers are added.
Throughput stagnation demonstrates that the non-parallelizable portion (expert section) dominates.

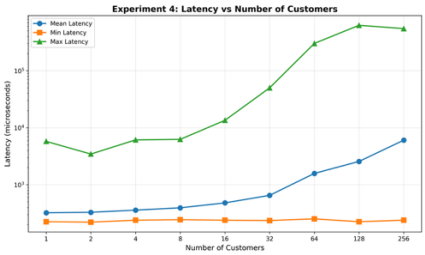
Experiment 3: Special Robot Type (16 Expert Engineers)

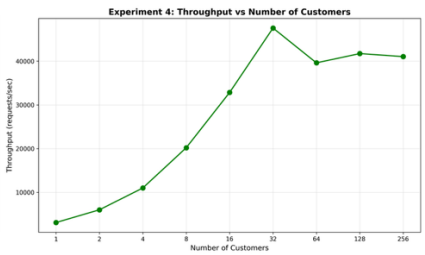




Mean latency stays low and stable up to 16–32 customers. Throughput increases from 3K → 46K.
Latency curve roughly doubles every time concurrency doubles beyond 64 customers.
The thread pool successfully absorbs concurrent requests until **expert saturation (~16–32)**.
Beyond that, requests queue up, leading to a latency increase but smoother than the 1-expert case.
Throughput plateaus after saturation, limited by expert pool processing rate.

Experiment 4: Special Robot Type (Number of Experts = Number of Customers)





Mean latency remains consistently low until 32–64 customers. Throughput rises up to 47K requests/sec and remains stable afterward. Behavior similar to 3, but slightly better at lower customer counts. When experts scale proportionally with customers, the system achieves maximum throughput and lowest queue delay.
Latency increases at very high concurrency (128–256) due to network and CPU saturation, not thread contention.
The setup achieves optimal concurrency utilization before hardware saturation limits appear.

1 is Excellent parallel scalability. Beyond that, resource limits (CPU/network) cause latency inflation.

2 is Poor scalability due to single expert contention. Latency rises exponentially; throughput remains constant. Validates bottleneck effects and shows worst-case performance scenario

3 is a significant improvement in scalability and throughput over single expert. Demonstrates benefit of parallel expert processing but still bounded by fixed pool size. Represents a balanced multi-threaded system

4 is Best performing configuration overall. Achieves ideal parallelism - no expert queue buildup until extreme concurrency. Demonstrates upper bound of system performance