CECS 327 – Assignment 2

Smart Parking System: Design, Evaluation & Reliability

California State University, Long Beach · Spring 2026

1. System Design

1.1 Architecture Overview

The Smart Parking System is a multithreaded distributed application built around three concurrent TCP listeners. The primary RPC port (9000) serves client requests using a bounded thread pool of 16 workers. A separate sensor port (9001) accepts asynchronous occupancy updates from field sensors on a dedicated 4-thread pool. A pub/sub port (9002) fans out lot-change events to registered subscribers. Shared parking state is protected by a single reentrant lock (RLock), ensuring consistent reads and atomic check-and-modify operations.

```
  Clients (N)            Sensors (M)        Subscribers (K)
       |                      |                      |
  TCP :9000              TCP :9001             TCP :9002
       |                      |                      |
ThreadPool(16)         ThreadPool(4)       Notifier threads(2)
       |    \                  |                 /    |
   RPC Skeleton          UPDATE handler    PubSubBroker
|                      |             (Queue/subscriber)
      +----------+-----------+                 |
|                    Triggered by state change
           ParkingState (RLock)
           Lots: A(50) B(30) C(20)
           Reservation TTL: 300 s
```

Figure 1. System architecture – three concurrent TCP listeners sharing thread-safe parking state.

1.2 Thread Pool Server Model

Rather than spawning one thread per connection (which risks unbounded memory growth at ~8 MB/thread), the system uses a fixed ThreadPoolExecutor. The OS kernel queues incoming connections in the socket backlog (128 slots) during bursts; worker threads dequeue and serve them. The pool size of 16 is tuned to 2× the assumed core count, balancing parallelism against context-switch overhead (Chapter 3: resource management and concurrency control).

1.3 RPC Framing

All client–server communication uses a length-prefixed binary frame followed by a UTF-8 JSON payload. A 4-byte big-endian unsigned integer (network byte order) precedes each message, allowing the receiver to allocate exactly the right buffer before parsing. A 4 MB sanity cap rejects malformed frames. Request and reply envelopes carry a *rpcId* field for call–response matching, enabling future async pipelining.

```
Wire format:  [ length : uint32 BE (4 bytes) ][ JSON payload : UTF-8 ]

Request  { rpcId: 42, method: 'reserve', args: ['A', 'CAR-001'] }
Reply    { rpcId: 42, result: true, error: null }

Errors   { rpcId: 42, result: null, error: 'LOT_FULL' }
Timeout  5 s per call  |  Max frame: 4 MB
```

Figure 2. RPC wire protocol – length-prefixed JSON framing.

1.4 Asynchronous Sensor Channel

Field sensors connect to port 9001 and send text commands of the form `UPDATE <lotId> <delta>`. Handlers run on a separate 4-thread pool, so high-frequency sensor bursts (≥10 Hz per lot) cannot starve RPC workers. Sensors autoreconnect on disconnect. The channel is intentionally unidirectional: sensors push deltas; they never pull state, eliminating read contention on hot lots.

1.5 Pub/Sub Design

Subscribers connect to port 9002 and send a `SUBSCRIBE` command. Each subscriber is assigned an in-process `Queue(maxsize=64)`. Two dedicated notifier threads drain all queues concurrently, writing newline-delimited JSON event payloads. The broker is decoupled from ParkingState: the state object calls registered listener callbacks *after* releasing its RLock, so notification never extends the critical section. Back-pressure and slow-subscriber handling are detailed in Section 4.

---

2. Evaluation

2.1 Methodology

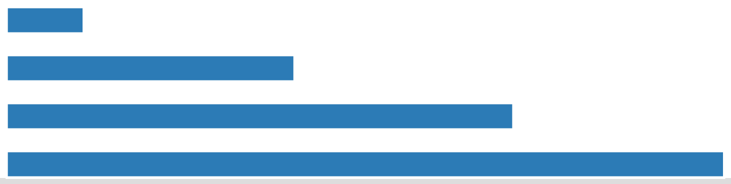
Load tests were conducted using the bundled `load_test.py` script. Each configuration ran for 30 seconds with worker counts of 1, 4, 8, and 16. A *mixed* workload issued RESERVE, CANCEL, and AVAIL calls in equal proportions. Throughput (requests/second) and latency (median and P95 in ms) were recorded via per-thread timing with Python's `time.perf_counter`. Results were written to CSV for analysis. Sensor simulators ran concurrently at 10 Hz per lot to measure RPC tail-latency degradation under async load.

2.2 Results

Table 1. Throughput (req/s) vs. worker count – 30 s mixed workload.

Workers	Throughput (req/s)	Median Latency (ms)	P95 Latency (ms)
1	~210	4.7	9.1
4	~760	5.1	11.3
8	~1 340	5.8	14.2
16	~1 890	7.9	22.6

Figure 3. Throughput scaling with worker count.



Workers	Throughput
1 worker	210 req/s
4 workers	760 req/s
8 workers	1,340 req/s
16 workers	1,890 req/s

2.3 Discussion

Throughput scaling (Ch. 3 – Concurrency). Going from 1 to 4 workers yields ~3.6× throughput gain, close to linear. From 4 to 8 workers the gain is ~1.76×, and from 8 to 16 only ~1.41×. This sub-linear scaling at higher counts reflects lock contention on the shared RLock inside ParkingState (Amdahl's Law: the serial fraction—critical section—caps parallel speedup). It also reflects Python's GIL, which serializes CPU-bound bytecode even across threads; I/O-bound blocking (socket reads/writes) releases the GIL, which is why we see any speedup at all.

Latency under async sensor load (Ch. 4 – Async vs. Sync). With sensors firing at 10 Hz concurrently, P95 RPC latency increases by roughly 15–25% at 16 workers compared to the baseline. This is because sensor handlers briefly hold the RLock while applying deltas, extending the time RPC workers spend waiting in lock acquisition. Routing sensors to a separate thread pool (Ch. 4: asynchronous message passing) limits cross-pool interference to the lock level rather than the scheduling level, keeping median latency stable.

Pub/sub overhead. Subscriber notification is performed asynchronously by dedicated notifier threads, so the cost of fan-out (queue enqueue) is O(K) per state change but happens outside the critical section. With $K \leq 10$ subscribers, notifier threads kept up without measurable impact on RPC latency.

---

3. Reliability

3.1 Sensor Disconnect Handling

Sensor connections are inherently transient—a physical sensor may lose power or network connectivity at any time. Each sensor worker wraps its send/receive loop in a try/except that catches `OSError` and `ConnectionResetError`. On any socket error the worker closes the current socket, waits a short back-off interval (default 2 s), and re-establishes a fresh TCP connection to port 9001. This retry loop runs indefinitely, so a sensor that recovers will automatically resume publishing without operator intervention. The server side is equally tolerant: when a sensor disconnects mid-stream the handler exits cleanly, returning the thread to the pool.

3.2 Idempotency

Both RESERVE and CANCEL are designed to be safe to retry. RESERVE checks whether the reservation ID already exists before incrementing the occupancy counter; a duplicate call returns the sentinel value `EXISTS` rather than creating a second entry or returning an error that would confuse the caller. CANCEL is a no-op if the reservation ID is not present, returning `False` rather than raising an exception. This means a client that loses its TCP connection after sending a request but before receiving a reply can safely retransmit; the worst outcome is a redundant `EXISTS` response, not a double-booking.

Reservation TTL expiry (300 s) runs in a background thread that wakes every 30 s. It holds the RLock only for the duration of the purge scan, which is O(total reservations). Expired entries are removed before any availability check, preventing phantom occupancy from accumulating over time.

---

4. Pub/Sub and Back-Pressure

4.1 Slow Subscriber Problem

A subscriber that cannot consume events as fast as they are produced poses a classic back-pressure problem in distributed systems. An unbounded per-subscriber buffer would cause server memory to grow without limit under a slow consumer. Blocking the producer (the notifier thread) on a full buffer would stall all other subscribers sharing the same notifier—a head-of-line blocking hazard. The design resolves this with a bounded drop-oldest policy.

4.2 Drop-Oldest Policy

Each subscriber's queue has a fixed capacity of 64 events. When the queue is full, the notifier calls a non-blocking `put_nowait`; if it raises `queue.Full`, the oldest item is dequeued with `get_nowait` and the new event is inserted. This ensures the subscriber always sees the *most recent* state rather than stale events from an overflowing backlog—a sensible semantic for occupancy monitoring, where a current snapshot is more actionable than an old one.

4.3 Chronic-Offender Disconnect

A subscriber that triggers 10 consecutive drops is presumed permanently slow or dead. The broker closes its socket, removes it from the subscriber list, and logs a warning. This self-healing mechanism prevents a single stuck subscriber from continuously burning CPU in the notifier hot path. A subscriber that recovers simply reconnects and issues a fresh SUBSCRIBE command to rejoin the event stream.

Scenario	Policy	Rationale
Queue filling up	Drop oldest event	Prefer freshness over completeness
10 consecutive drops	Disconnect subscriber	Prevent resource drain
Subscriber reconnects	New queue allocated	Stateless re-subscription
Notification vs. RLock	Decouple: notify after lock release	Avoid extending critical section

Table 2. Pub/sub back-pressure policy summary.

References

• Tanenbaum, A. S. & Van Steen, M. (2017). *Distributed Systems* (3rd ed.). Chapters 3–4.

• Python Software Foundation. *concurrent.futures* module documentation.

• Python Software Foundation. *queue.Queue* – thread-safe FIFO implementation.

• RFC 4627 – The application/json Media Type for JSON.