Consistency, Reliability, and Failure Handling Strategy

Target Audience: System Architects, DevOps, Senior Backend Engineers
Purpose: Defining how the TMS ensures data integrity and system stability in the face of concurrency, network partitions, and infrastructure failures.

---

1. Consistency Philosophy

In a distributed microservices environment, we accept that **Perfect Consistency** across the entire system at a single point in time is impossible (CAP Theorem). We adopt a hybrid strategy:

- **Transactional Boundaries (Strong Consistency):** Within a single Service Domain (e.g., Ticketing), data MUST be strongly consistent. You cannot sell the same seat twice.
- **Cross-Domain Propagation (Eventual Consistency):** Between services (e.g., Payment -> Notification), we accept a delay of milliseconds to seconds.

---

2. Concurrency Control Strategies

2.1 The "Seat Sale" Race Condition

- **The Problem:** Two users (A and B) try to buy Seat 5 on Trip 101 at the exact same millisecond.
- **The Strategy:** Pessimistic Locking via Redis (Redlock).
    1. Incoming Request -> Attempt to acquire Lock Key `lock:schedule:{id}:seat:{no}`.
    2. **Winner:** Successfully sets the key. Proceed to DB transaction.
    3. **Loser:** Receives "Lock Busy" error. System returns `HTTP 409 Conflict` ("Seat selected by another user").
- **Why Redis?** Faster than Database Row Locking for high-traffic filtering.
- **Fallback:** Unique Constraint on the Database Table `(schedule_id, seat_number)` serves as the final hard barrier.

2.2 Inventory Updates

- **The Problem:** Updating the "Available Seats" count on a bus.

- **The Strategy:** Atomic Increments/Decrements.
  - `UPDATE schedules SET available_seats = available_seats - 1 WHERE id = X AND available_seats > 0`
  - This ensures we never dip below zero, regardless of concurrency.

---

3. Failure Handling & Resilience

3.1 Partial System Failure

- **Scenario:** The `Notification Service` crashes.
- **Impact:** Users buy tickets but don't get SMS confirmations.
- **Mitigation (Decoupling):**
  - The `Ticketing Service` publishes a `TicketSold` event to RabbitMQ and considers its job done.
  - RabbitMQ holds the message in a Durable Queue.
  - When `Notification Service` restarts (even hours later), it consumes the backlog and sends the SMS.
  - **Result:** Revenue is not lost; User experience is degraded but recoverable.

3.2 Cascading Failures

- **Scenario:** The Database slows down, causing `Ticketing Service` to hang, backing up the `Gateway`.
- **Mitigation: Circuit Breakers**.
  - If a service detects high latency or 50% Error Rate from a dependency (e.g., DB or external Payment API), it "Trips the Breaker".
  - Subsequent requests fail fast (`HTTP 503 Service Unavailable`) instantly, rather than waiting for timeouts.
  - This allows the stressed resource time to recover.

3.3 Network Timeouts (The "Unknown State")

- **Scenario:** We send a request to Mobile Money API. The connection times out. Did the user pay or not?
- **Strategy: Reconciliation Polling**.
  - We mark the local transaction as `PENDING_VERIFICATION`.
  - A background Cron Job polls the Payment Provider: "What is the status of Transaction X?"
  - We update our state based on the provider's definitive answer.

---

4. Idempotency Strategy

Idempotency is the property where performing an operation multiple times has the same result as performing it once. This is **critical** for payments.

- **Implementation:** All critical mutation APIs accept an `Idempotency-Key` header (UUID).
- **Process:**
  1. Client generates UUID `A` and sends `POST /charge`.
  2. Server stores `KEY: A` -> `Response: 200 OK`.
  3. Client crashes, doesn't receive response.
  4. Client retries `POST /charge` with `Idempotency-Key: A`.
  5. Server sees `KEY: A` exists. **does NOT charge again**.
  6. Server returns the stored `Response: 200 OK`.

---

5. Data Recovery & Durability

5.1 The "Offline POS" Conflict

- **Scenario:** Internet cuts out. Agent sells Seat 5 offline. Meanwhile, Online User buys Seat 5. Internet returns. Sync occurs.
- **Conflict:** Two Sales, One Seat.
- **Policy: Server Authority.**
  - The Online sale is honored (it reached the "Source of Truth" first).
  - The Offline sale is rejected during sync.
  - Operational Protocol: Agent sees "Sync Error: Seat Taken" and must refund/re-seat the passenger manually.

5.2 Disaster Recovery (DR)

- **Database:** Point-in-Time Recovery (PITR) enabled via WAL logs (PostgreSQL). We can restore the DB to the state it was in at `14:03:00` precisely.
- **Queues:** RabbitMQ configured with `delivery_mode=2` (Persistent). Messages are written to disk, surviving a broker restart.

---

6. Summary of Guarantees

Component	Guarantee Level	Mechanism
Payments	Exactly Once (Effectively)	Idempotency Keys + Polling
Inventory	Strong Consistency	Redis Lock + DB Unique Constraint
Notifications	At Least Once	RabbitMQ Durable Queues
Fleet Tracking	Best Effort	UDP/Volatile Redis Keys (Loss acceptable)