Distributed Supply Management System (DSMS) – Design Overview

Architecture Summary

The application is implemented as a Java RMI system where every store (QC, ON, BC) runs an instance of StoreServerImpl. A shared RMI registry (running on the port in the same JVM) is used to bind the three store instances. Clients (managers and customers) locate their home store by inspecting the first two characters of their ID and resolve the corresponding RMI stub through StoreServerRegistry.

Each server maintains its state only in-memory as required by the assignment statement. Inventory is stored in a ConcurrentHashMap. For every item the store also keeps a wait-list implemented as a FIFO Deque containing customer IDs. Additional per-customer data such as remaining budget, purchases and return history are managed through CustomerAccountManager, which creates one CustomerAccount instance per customer. Because all three servers run inside one JVM in this reference implementation, the manager acts as a lightweight shared repository accessed through synchronized methods.

Inter-server communication uses RMI invocations (requestRemotePurchase, requestRemoteItemLookup, requestRemoteReturn). These methods encapsulate the logic that, in a distributed deployment, would be triggered through UDP calls. Servers log every meaningful event to logs/_server.log. Clients log their actions to logs/clients/.log.

Main Components

Server Layer

StoreServerImpl – Implements the RMI interface, contains the inventory, waitlist handling and orchestrates automatic waitlist fulfilment. Uses fine-grained locks (ReentrantLock) in ItemRecord for concurrency during quantity updates.
CustomerAccount / CustomerAccountManager – Tracks a user's remaining CAD 1000 budget, purchases per store and per item, and manages refunds or policy enforcement (remote store purchase limit of 1 item).
StoreServerRegistry – Utility that ensures an RMI registry is available, binds the three servers and offers lookup helpers for clients and other servers.

Client Layer

ManagerClient – Command-line program for store managers (add/remove/list operations). Maintains per-manager log.
CustomerClient – Command-line program for customers (purchase/find/return operations) and logging.

Common

StoreServer – RMI interface exposing manager and customer operations alongside internal inter-server methods.
PurchaseResult – Serializable DTO carrying success flag, message and charged price.

Data Structures

ConcurrentHashMap – Inventory per store.
Deque – Waitlist per item, ensures FIFO order.
CustomerAccount – Maintains Map purchases per store (enforces cross-store limit) and Map> purchase history per item.
PurchaseRecord – Java class containing item ID, store code, purchase date and price.

Concurrency Strategy

ItemRecord holds a ReentrantLock used to guard quantity changes and waitlist assignments for the specific item.
CustomerAccount methods are synchronized to keep budget checks and purchase recordings atomic, avoiding race conditions when multiple servers operate on the same customer simultaneously.
CustomerAccountManager stores accounts in a ConcurrentHashMap to permit concurrent access.

Automatic Waitlist Fulfilment

Whenever inventory increases (through addItem or a successful return) the server calls processWaitList. The queue is processed while stock remains. Purchases triggered from the waitlist reuse the same validation logic (performLocalPurchase) ensuring budget and policy rules are enforced. Failures (e.g., customer budget exhausted) are logged and the next customer is considered.

Logging

Server logs: logs/_server.log
Client logs: logs/clients/.log
Both use Java Util Logging with file handlers and simple formatting.

Testing Strategy

JUnit 5 tests (StoreServerImplTest) cover the happy path for adding inventory, purchasing, waitlisting and returning items, and verify that item lookup returns non-empty results. Manual testing is performed through the CLI clients.

Notable Challenges

Coordinating cross-store purchases and budget enforcement required a shared customer account manager. In a multi-process deployment this component would need to be distributed (e.g., replicated via UDP messages or persistent storage). For the scope of this assignment it remains in-process. Automatic waitlist fulfilment must avoid duplicate entries and handle failures gracefully while maintaining concurrency safety.