Assignment 3 Documentation

Alexandre Payumo

# Summary:

The Distributed Supply Management System (DSMS) is a sophisticated distributed application that manages inventory across three geographically distributed stores: Quebec (QC), Ontario (ON), and British Columbia (BC). The system demonstrates advanced distributed systems concepts including JAX-WS Web Services (SOAP/HTTP) for client-server communication, UDP for inter-server communication, comprehensive concurrency control, and atomic two-phase commit protocol for cross-store item exchanges.

# System Architecture:

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Component Architecture**

**Core Components:**

- **StoreServer Interface**: Java interface defining service methods with JAX-WS annotations

- **StoreServerImpl**: Main server implementation with @WebService annotation

- **UDPCommunicator**: Handles marshalled inter-server communication

- **ManagerClient/CustomerClient**: User interface applications using generated web service stubs

- **DSMSLogger**: Comprehensive logging system

- **Generated Client Stubs**: wsimport-generated classes for web service consumption

- **ExchangeTransaction**: Data structure tracking pending exchange operations for two-phase commit

**Data Models:**

- **Item**: Inventory item with ID, name, quantity, price

- **Purchase**: Purchase record with customer, item, date, price

- **UDPMessage/Request/Response**: Marshalled communication objects

- **ExchangeTransaction:** Tracks customer ID, new item ID, old item ID, and timestamp

  for pending exchange operations during two-phase commit protocol

**Communication Protocols**

**Web Services Communication (Client ↔ Server):**

Protocol Stack:

- Application Layer: JAX-WS (Java API for XML Web Services)

- Message Format: SOAP (Simple Object Access Protocol) - XML-based

- Transport: HTTP/HTTPS

- Service Description: WSDL (Web Services Description Language)

Ports:

- QC Server (8080 HTTP), ON Server (8081 HTTP), BC Server: (8082 HTTP)

Endpoints:

- QC: <http://localhost:8080/QCServer>, ON: <http://localhost:8081/ONServer>, BC: <http://localhost:8082/BCServer>

WSDL URLs (for client stub generation):

- QC: <http://localhost:8080/QCServer?wsdl>, ON: <http://localhost:8081/ONServer?wsdl>, BC: <http://localhost:8082/BCServer?wsdl>

Methods:

- Manager: addItem, removeItem, listItemAvailability

- Customer: purchaseItem, findItem, returnItem, exchangeItem

- Helper: addToWaitlist, getStorePrefix

**UDP Communication (Server ↔ Server):**

- Ports: 9001 (QC), 9002 (ON), 9003 (BC)

- Protocol: UDP with marshalled Java objects

- Operations: Cross-store purchases, inventory searches

# Technical Implementation:

**Data Structures**

A computer screen shot of a program

AI-generated content may be incorrect.

**Design Benefits:**

- **Scalability**: ConcurrentHashMap allows multiple concurrent readers

- **Consistency**: ReadWriteLocks ensure data integrity during updates

- **Performance**: Minimal lock contention with per-item locking strategy

# Test Scenarios:

**Test Case 1: Manager Operations**

**Description:** Testing manager functionality including add, remove, and list operations with security validation through JAX-WS web service interface.

**Test Scenarios:**

- Add new items to inventory

- Update existing item quantities (aggregation)

- Remove partial quantities from items

- Remove all quantity (set to 0, trigger waitlist)

- List inventory with proper formatting

- Invalid manager ID rejection

**Expected Output:** Successful item management with proper security enforcement through web service endpoint validation.

**Actual Results: ALL TESTS PASSED (6/6)**

**Analysis:** Perfect security implementation with role-based access control working correctly through JAX-WS. The annotated service implementation properly validates manager credentials before executing operations via SOAP requests.

**Additional Tests Verified:**

- Waitlist auto-assignment when manager adds quantity

- Complete item removal from inventory

- Remove non-existent item error handling

- Cross-store manager operation prevention

**Test Case 2: Customer Purchase Operations**

**Description:** Testing customer purchase functionality including local/cross-store purchases with quantity support, item searches, budget management, and waitlist handling via web services implementation.

**Test Scenarios:**

- Local purchase with quantity specification

- Remote purchase from another store (UDP coordination)

- Multi-store item search across all three stores

- Insufficient budget rejection

- Insufficient quantity handling

- Out of stock triggers waitlist prompt

- Add customer to waitlist

- Invalid quantity rejection (0 or negative)

**Expected Output:** Successful customer operations with proper business rule enforcement through web service calls.

**Actual Results: ALL TESTS PASSED (8/8)**

**Analysis:** Excellent business logic implementation with proper UDP cross-store communication. JAX-WS web services seamlessly handle client-server communication via SOAP/HTTP while UDP manages inter-server coordination.

**Additional Tests Verified:**

- Waitlist user choice handling ("Yes"/"No")

- Automatic purchase from waitlist when item becomes available

- Multiple customers in same waitlist (position tracking)

- Cross-store waitlist functionality

- Multiple same-name items across stores

**Test Case 3: Customer Return Operations**

**Description:** Testing item return functionality with 30-day policy enforcement and purchase validation.

**Test Scenarios:**

- Valid return within 30 days

- Reject expired return (>30 days)

- Reject return of non-purchased item

**Expected Output:** Successful returns within policy window with proper refund processing.

**Actual Results: ALL TESTS PASSED (3/3)**

**Analysis:** Robust return policy implementation with accurate date calculation and purchase history validation. Web services implementation properly handles return requests with atomic budget refunds.

**Additional Tests Verified:**

- Return wrong customer (customer can't return others' items)

- Cross-store return (return item purchased from different store)

- Budget correctly updated after return

**Test Case 4: Customer Exchange Operations**

**Description:** Testing the new exchangeItem operation with atomicity guarantees, including local and cross-store exchanges with 30-day policy enforcement.

**Test Scenarios:**

- Local exchange (same store)

- Cross-store exchange (different stores)

- Reject exchange of expired item (>30 days)

- Reject exchange of non-owned item

**Expected Output:** Atomic exchange operations where both return and purchase succeed or both fail, maintaining data consistency across stores.

**Actual Results: ALL TESTS PASSED (4/4)**

**Analysis:** Excellent implementation of atomic two-phase exchange protocol. Cross-store exchanges properly coordinate via UDP with PREPARE→COMMIT/ROLLBACK phases. Budget adjustments (price differences) handled correctly. No data corruption observed during failure scenarios.

**Additional Tests Verified:**

- Exchange with price increase (customer pays difference)

- Exchange with price decrease (customer receives refund)

- Rollback on remote store failure

- Cross-store purchase limit enforcement during exchange

- Exchange eligibility validation before execution

**Test Case 5: Edge Cases & Business Rules**

**Description:** Testing system constraints, security boundaries, and business rule enforcement.

**Test Scenarios:**

- Enforce remote store purchase limit (1 item per remote store)

- Invalid customer ID rejection

- Prevent customer from manager operations

- Prevent manager from customer operations

- Track customer budget correctly across operations

**Expected Output:** Proper enforcement of all business rules and security constraints through web services validation.

**Actual Results: ALL TESTS PASSED (5/5)**

**Analysis:** Comprehensive business logic validation. Purchase limits enforced across stores via centralized purchase history tracking.

**Test Case 6: Concurrency & Synchronization**

**Description:** Testing thread safety with multiple concurrent operations on shared resources.

**Test Scenarios:**

- Handle concurrent purchases of same item

- Handle concurrent add/remove operations

**Expected Output:** No race conditions, data corruption, or deadlocks. Proper synchronization ensures data integrity.

**Actual Results: ALL TESTS PASSED (2/2)**

**Analysis:** Outstanding concurrency control using ReentrantReadWriteLock for item-level locking. Multiple threads can safely operate on different items concurrently while preventing conflicts on same items. JAX-WS's built-in thread-safe HTTP server properly handles concurrent SOAP requests without blocking.

**Additional Tests Verified:**

- Concurrent exchanges on different items

- Simultaneous purchases and returns

- Multiple managers modifying inventory simultaneously

- Waitlist processing during concurrent add operations

- No deadlocks when acquiring multiple locks (ordered locking prevents deadlock)

# Most Important/Difficult aspects:

**Additional Challenge in Assignment 3: Web Services Migration**

**Problem**: Migrating from CORBA to JAX-WS While Maintaining Functionality

**Complexity**:

- Converting CORBA-specific code to web service annotations

- Understanding JAX-WS marshalling/unmarshalling mechanisms

- Generating and integrating client stubs from WSDL

- Ensuring SOAP XML serialization works correctly for all data types

- Fixing cross-store search bug (foundItems serialization in UDPResponse)

- Coordinating HTTP ports (8080/8081/8082) with UDP ports (9001/9002/9003)

- Adapting build process for wsimport stub generation

- Testing web service endpoints and WSDL accessibility

**Solution - JAX-WS Implementation**:

**Server-Side**:

1. Removed all CORBA imports and dependencies (org.omg.\*)

2. Added JAX-WS annotations to interface and implementation:

@WebService

@WebMethod

@WebParam

3. Changed server startup from ORB.init() to Endpoint.publish()

4. Maintained all business logic unchanged

**Client-Side**:

1. Removed CORBA naming service lookups

2. Implemented wsimport-based stub generation in build process

3. Updated clients to use generated service stubs:

StoreServerService service = new StoreServerService(wsdlURL);

StoreServer server = service.getStoreServerImplPort();

4. Maintained same method signatures for seamless transition

**Achievements:**

- Seamless migration with zero functional regressions

- Improved firewall compatibility (HTTP vs IIOP)

- Platform-independent service descriptions (WSDL)

- Industry-standard protocols (SOAP/HTTP)

- Better debugging capabilities (readable XML messages)

- Fixed cross-store search serialization bug from Assignment 2