

# Software Design Document (SDD)

---

## System Overview

This software implements a distributed file system that stores data across IoT-based storage nodes, enabling automatic replication for fault tolerance and data availability. The system employs a master-minion architecture where files are distributed and mirrored across IoT devices using RAID01 topology, ensuring data remains accessible even when primary storage nodes fail or become corrupted.

File storage operations are managed through an asynchronous task queue processed by a multi-threaded execution engine. This design enables concurrent I/O operations while maintaining system responsiveness and efficient resource utilization. Integration with Network Block Device (NBD) protocol allows the distributed storage to present as a standard Linux block device, enabling users to interact with the system through native Linux file management interfaces.

The software also supports dynamic plugin architecture with runtime loading of user-defined plugins, enabling customization of read and write operations, as well as other extended functionality, without system downtime.

## 2. Requirements

### Functional Requirements (FRs):

- Distributed Storage System: Develop an IoT-based distributed storage system with RAID 0+1 redundancy across network-connected storage nodes
- Linux Integration: Provide standard Linux directory interface through NBD integration for seamless user interaction
- IoT Storage Node Support: Support distributed IoT devices as storage endpoints
- Plugin Architecture: Enable runtime loading and registration of custom plugins to extend or override default read/write operations
- Reliability & Error Handling: Implement error detection, reporting, and recovery mechanisms for IoT node communication failures

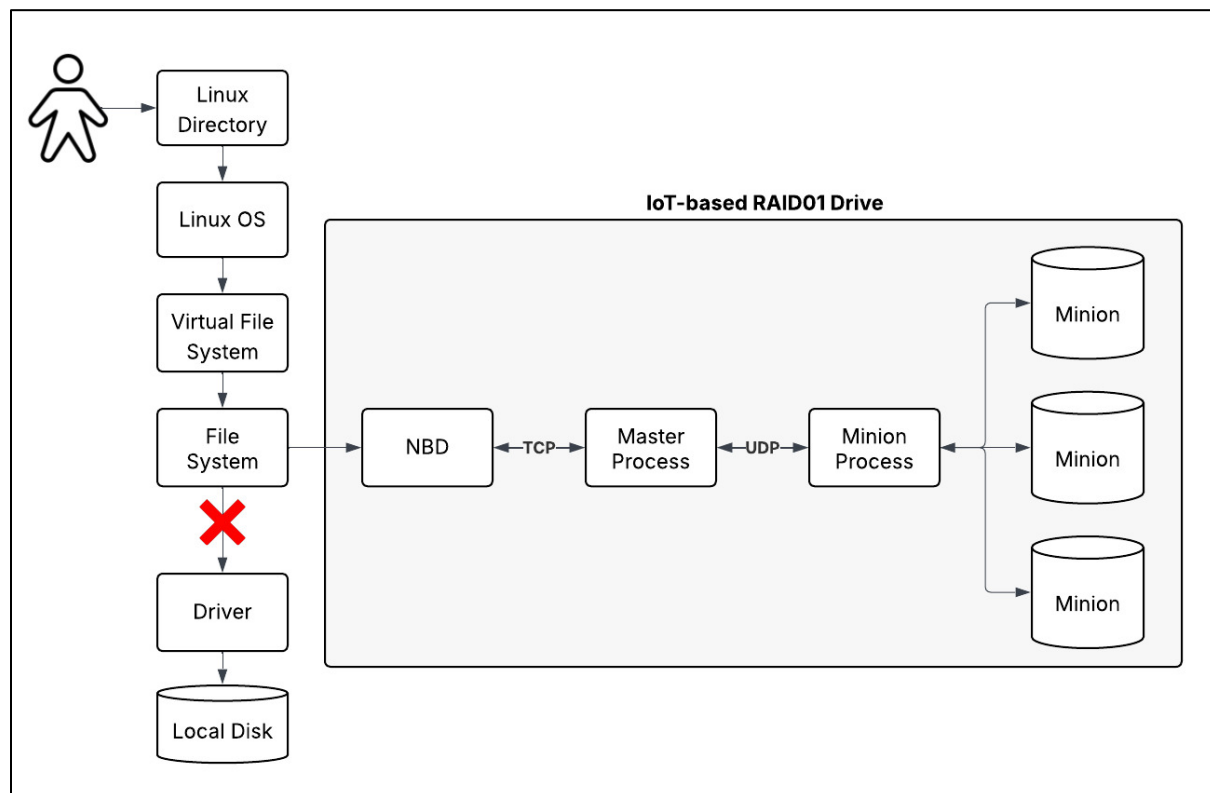
## Non-Functional Requirements (NFRs):

Performance, reliability, scalability, security, portability, usability, maintainability, observability, modularity, code reuse, concurrency support, adherence to programming best practices, and modern C++ implementation.

## Constraints & Assumptions:

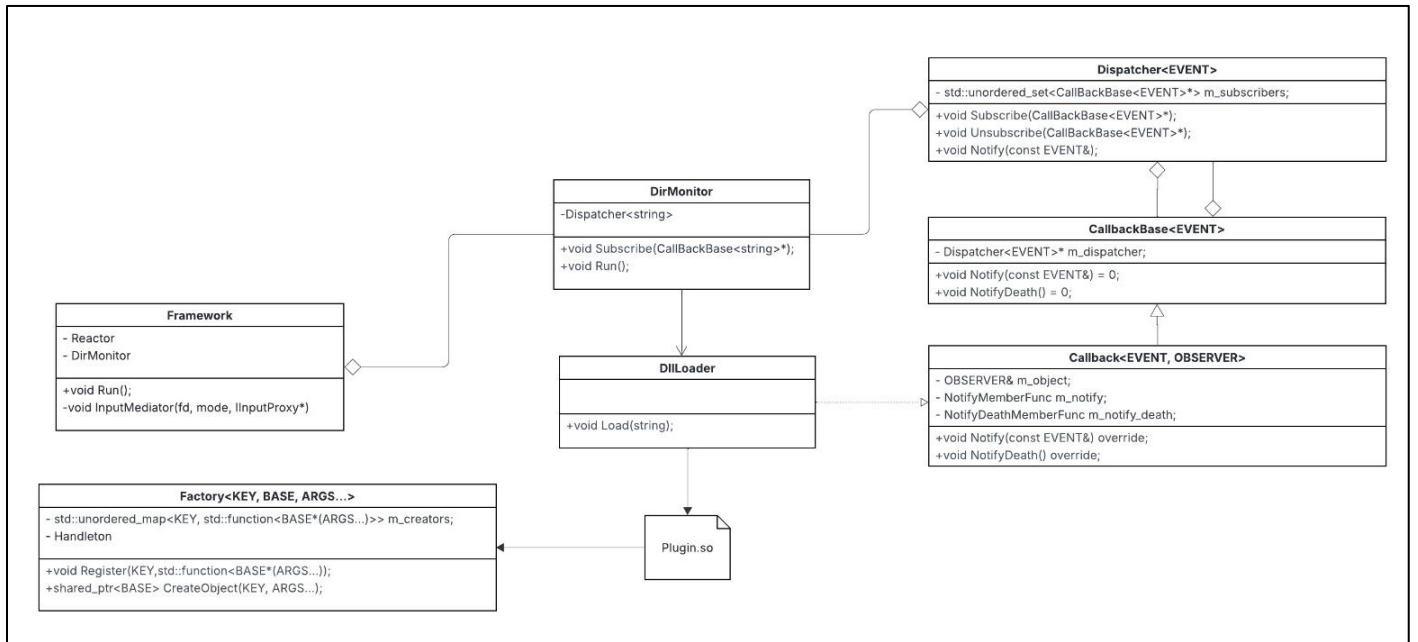
- **Platform/OS:** Linux-based systems
- **Languages:** Modern C++ (C++20 or later)
- **Libraries:** Standard C++ libraries
- **Standards:** Linux directory standards, NBD protocol compliance
- **Deployment:** IoT devices supporting Unix-like operating systems with network connectivity

## 3. System Architecture

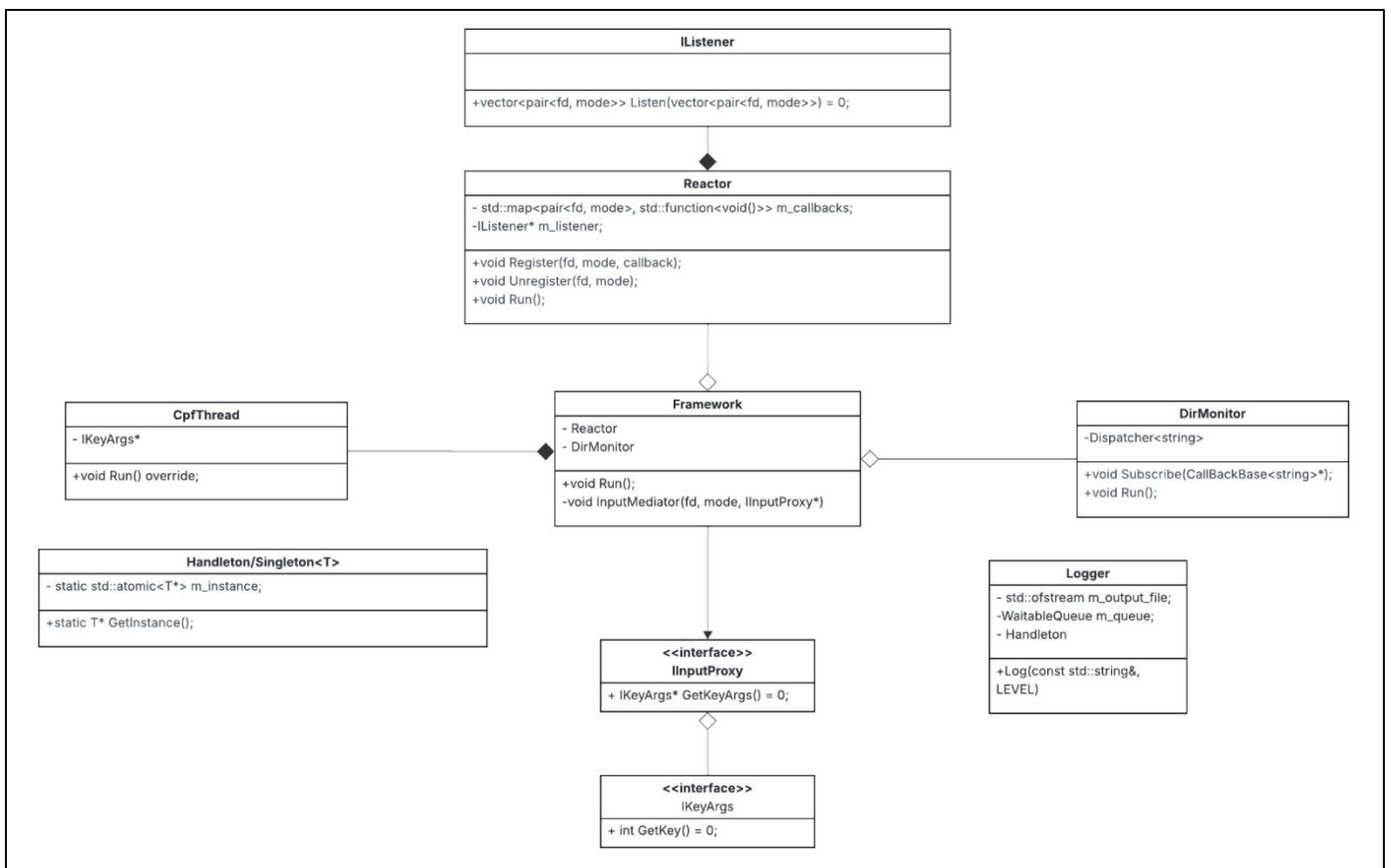


## 4. Software Design

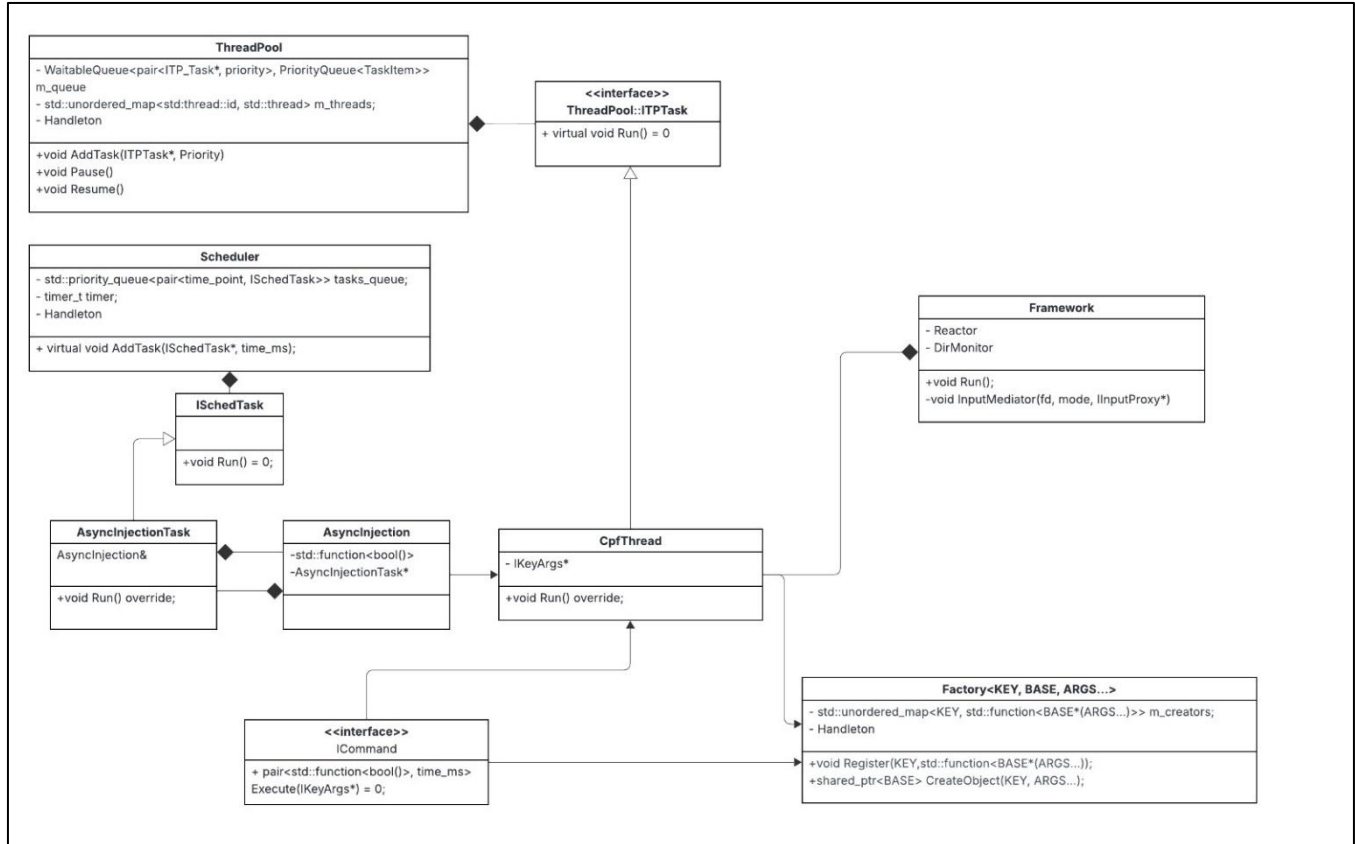
### 4.1.1: Framework Architecture - Plugin Interaction



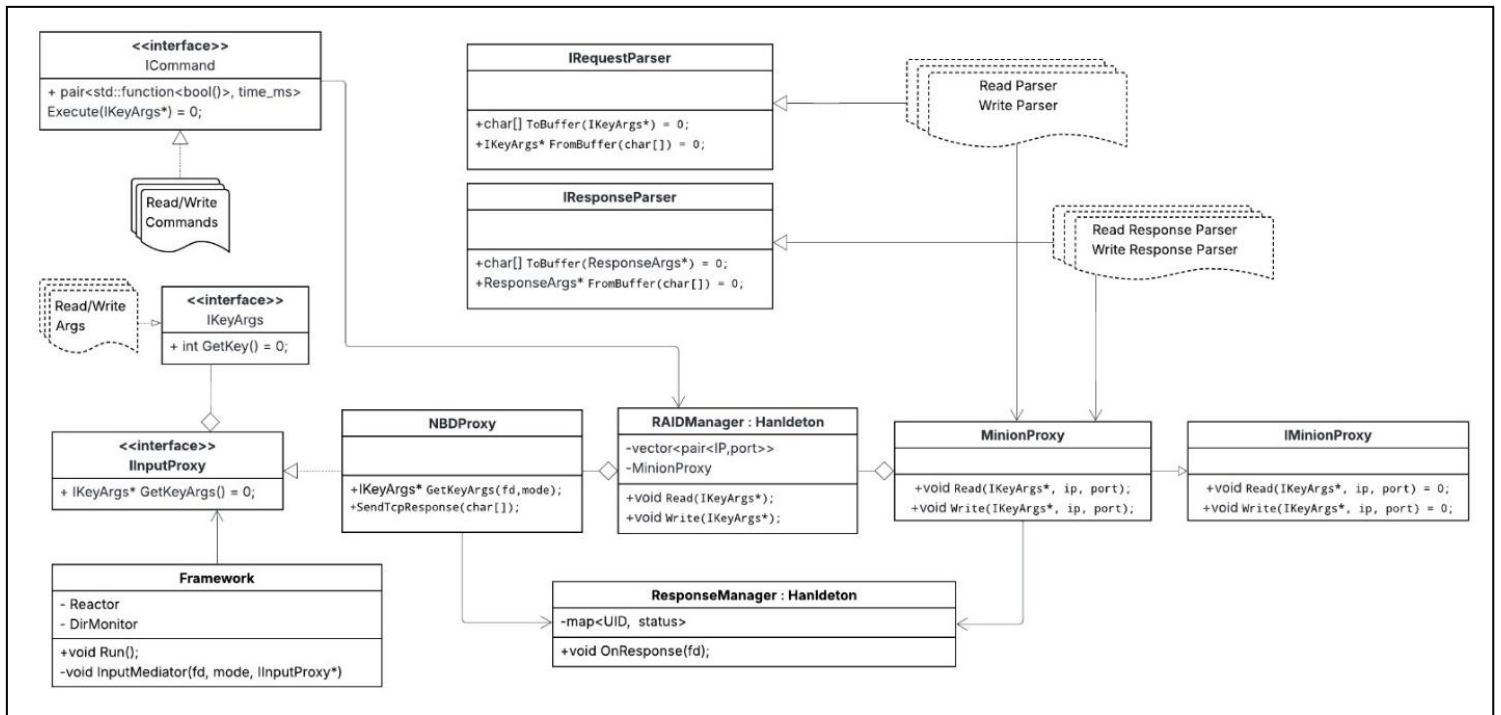
### 4.1.2: Framework Architecture - Reactor



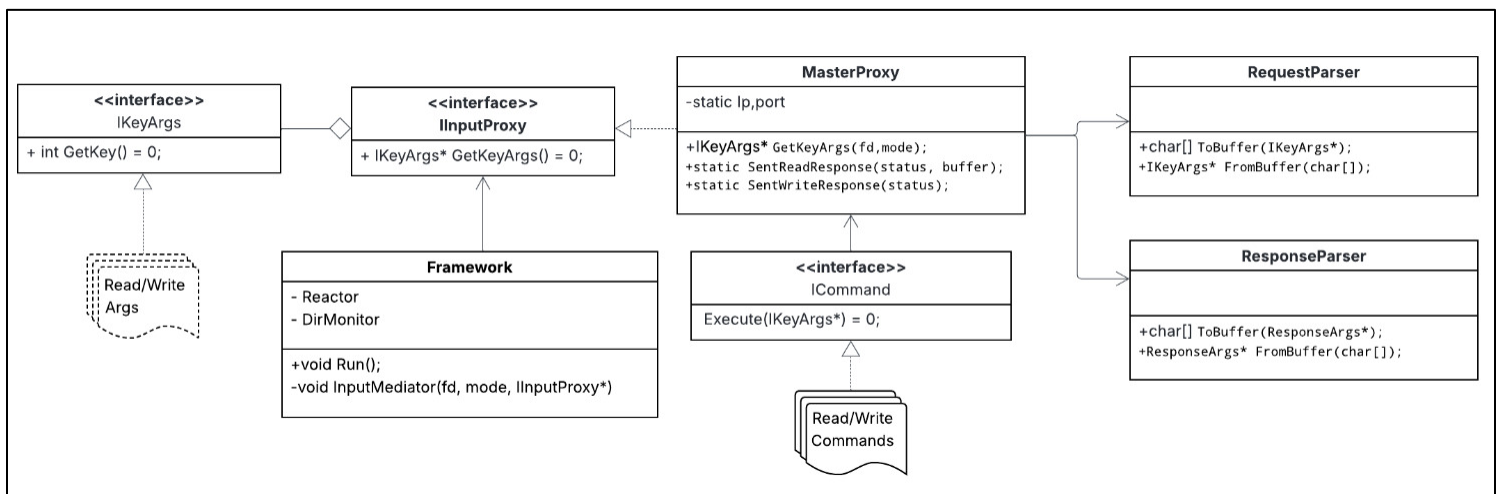
### 4.1.3: Framework Architecture - ThreadPool



## 4.2.1: Master Architecture

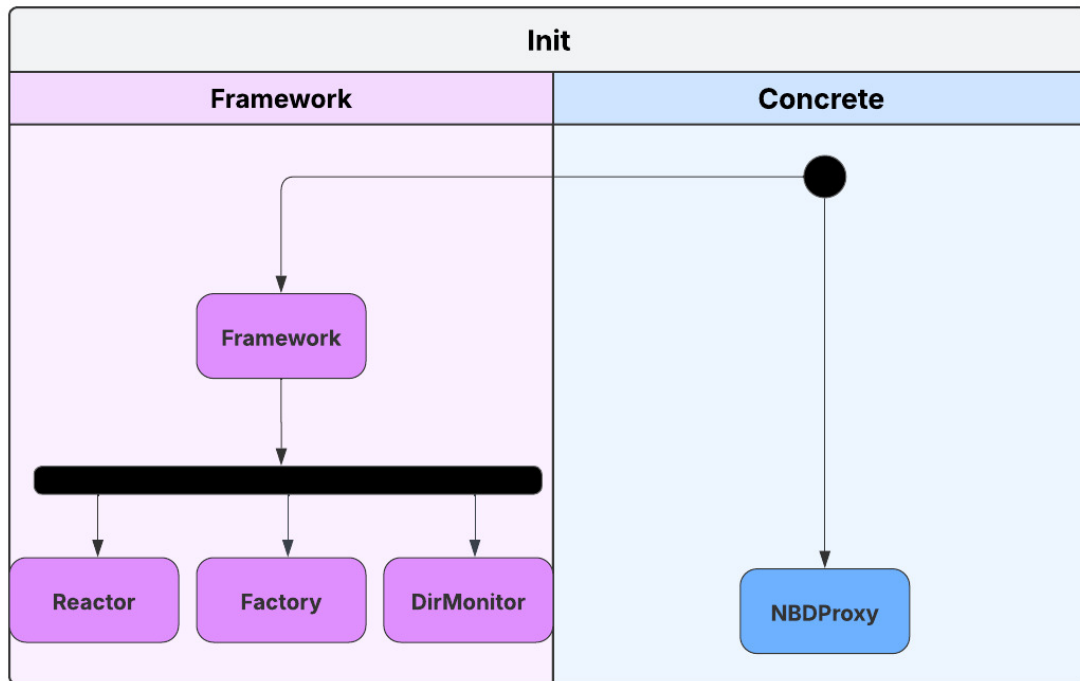


## 4.2.2: Minion Architecture

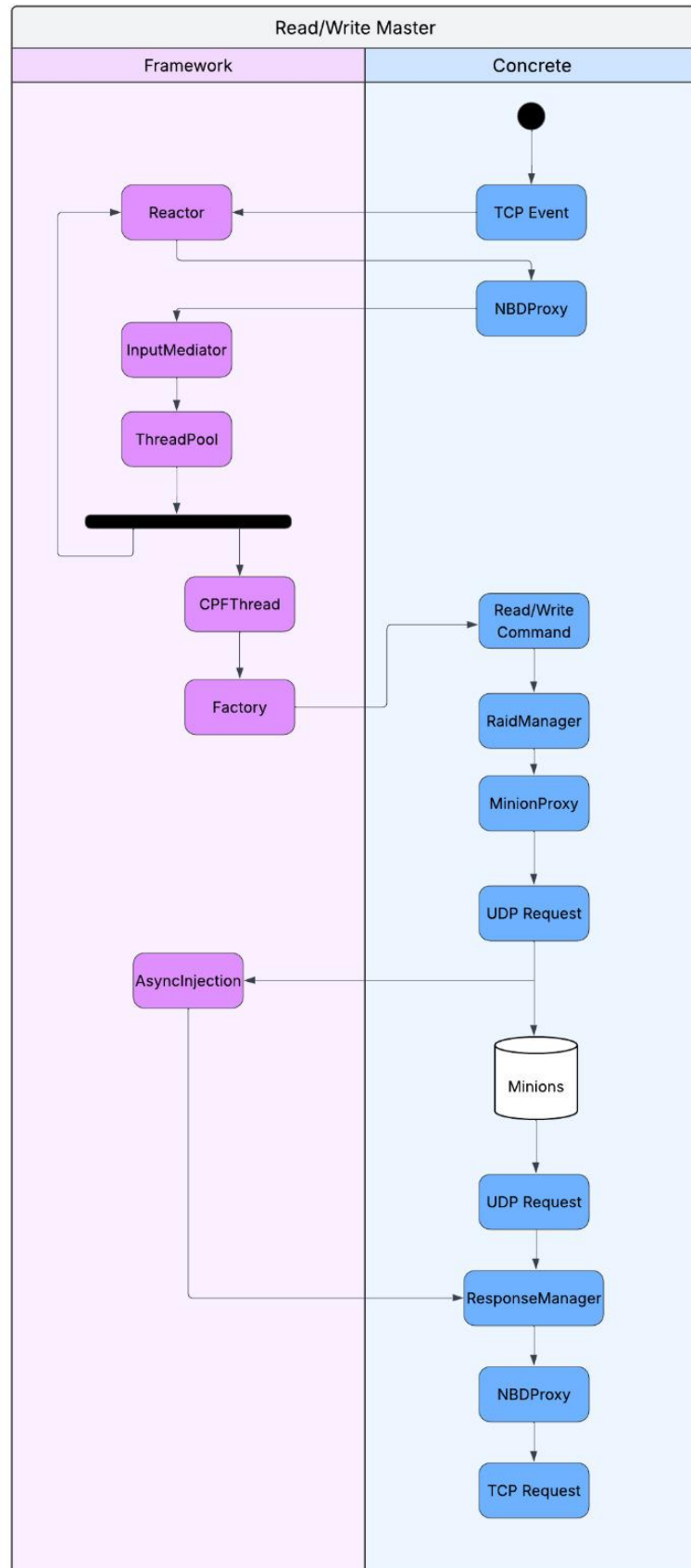


## 5. Activity Diagrams

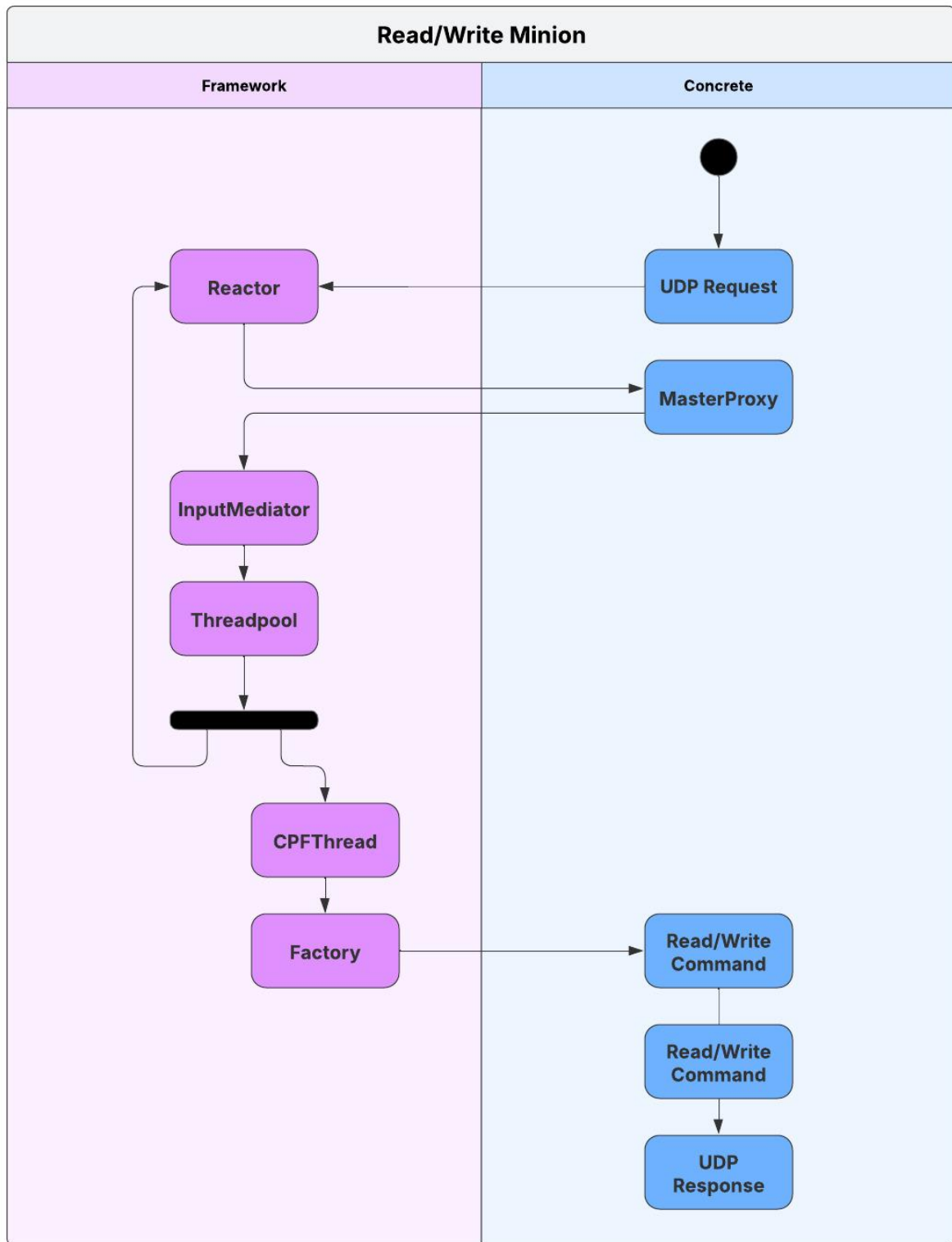
### 5.1: System Initialization



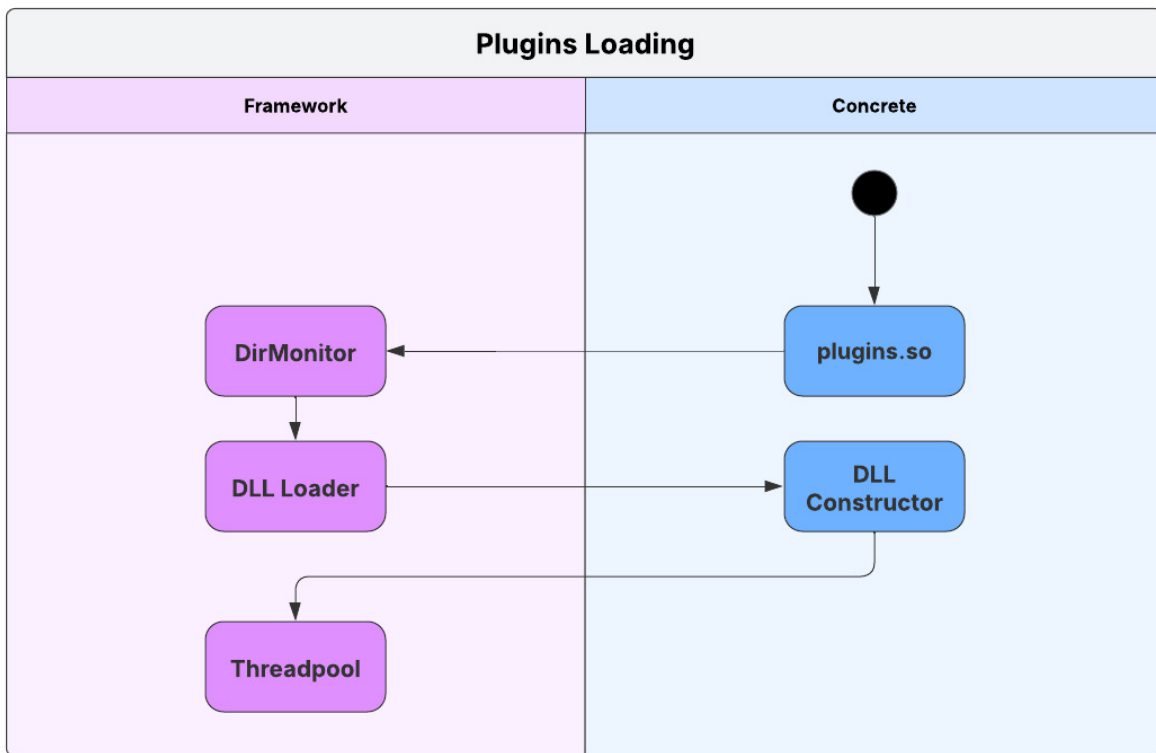
## 5.2: Activity Flow of Read/Write from Master Side



### 5.3: Activity Flow of Read/Write from Minion Side



## 5.4: Plugin Loading Sequence



## 6. Technical Implementation Profile

### Multithreading Architecture:

- (1) *Main Thread* – executes the framework (blocking)
- (2) *Thread Pool* – Dynamically sized worker threads created upon first task arrival to handle concurrent I/O operations
- (3) *Directory Monitor* – Dedicated background thread monitoring the plugins directory for runtime plugin loading
- (4) *Logger* – Spawned on first exception to manage all system logging operations asynchronously
- (5) *Scheduler Thread* – OS-managed thread for periodic task execution when required

## **Multiprocessing:**

The system does not support multi-processing deployment. Running multiple executable instances against the same framework instance will result in undefined behavior due to singleton object conflicts.

## **Inter-Process Communication (IPC):**

*NBD Interfaces* – Receives TCP requests from Network Block Device clients, returning operation status for write command and requested data for read commands.

*Master-Minion Communication* – Maintains UDP connections with minion nodes. Write operations replicate data to two minion nodes (primary + backup) and return aggregated status. Read operations retrieve data from available minion nodes.

## **Performance Characteristics:**

*Time Complexity* – Framework operates at  $O(n + m)$  where  $n$  represents monitored file descriptors and  $m$  represents registered command handlers

*Concurrency* – Supports concurrent operations through configurable thread pool with optimal resource utilization

## **Development Requirements:**

*IDE* – Compatible with modern development environments (tested with Visual Studio Code)

*C++ Version* – Requires C++20 or later (utilizes `std::counting_semaphore`)

*Platform* – Unix-like systems only