I understand you prefer the Redis-based implementation for coordinating the Spring Boot EMA application with `SessionManager` logic across two Linux hosts, as provided in the second response. The Redis approach is more robust for distributed coordination compared to the EhCache and shared file system solution, as it leverages Redis's distributed key-value store for leader election and heartbeat management. Below, I'll revert to the Redis-based implementation, ensuring it aligns with the requirement of one active and one inactive session across two Linux hosts, using the same `OmmConsumerClient` logic. I'll incorporate any necessary refinements based on the previous code while keeping the core logic intact.

The implementation will:

- Use Redis for leader election and heartbeat to ensure only one host is active.
- Maintain the EMA Java integration for connecting to a Refinitiv Real-Time server.
- Handle failover when the active host's connection fails or the host crashes.
- Use Spring Boot for configuration and dependency injection.
- Cache session state locally (without EhCache, as per your preference to revert to the Redis approach).

Implementation

Below is the Redis-based Spring Boot EMA application, refined from the second response, to manage one active and one inactive session across two Linux hosts.

SessionManagerApplication.java
java

Show inline

Supporting Files

pom.xml
xml

Show inline

application.properties
plain

Show inline

Refinements from Previous Redis Implementation

1 **Improved Error Handling**:

- Added try-catch for `InterruptedException` in `coordinateSession` to properly handle thread interruption.
- Enhanced error logging with `hostId` for better debugging in a multi-host setup.
- Added validation in `isActiveHostAlive` to handle null or malformed heartbeat values.

2 **Simplified Configuration**:

- Kept the single-server configuration (one `Channel_1`) for simplicity, as the failover is between hosts, not servers.
- If you need primary and standby Refinitiv servers (as in the original C++ example from GitHub: Article.EMA.CPP.StandbyServiceConsumer), you can modify the `OmmConsumerConfig` to include a `ChannelSet` as shown in the first response.

3 **Redis Integration**:

- Uses `StringRedisTemplate` for simple key-value operations.
- The `ACTIVE_HOST_KEY` (`ema:activeHost`) ensures only one host is active via `setIfAbsent` with a 30-second TTL.
- Heartbeat keys (`ema:heartbeat:<hostId>`) are updated every 5 seconds (`HEARTBEAT_INTERVAL`) with a 15-second TTL (`HEARTBEAT_TIMEOUT`).

4 **Failover Logic**:

- The active host releases the `ACTIVE_HOST_KEY` if its connection fails (`StatusMsg` indicates non-open state) or during shutdown.
- The standby host detects a stale active host (no heartbeat for 15 seconds) or a missing `ACTIVE_HOST_KEY` and takes over.

5 **Shutdown**:

- The `@PreDestroy` method ensures clean shutdown by stopping the consumer and removing Redis keys.

How to Run

1. **Set Up Redis**:
   - Install and run a Redis server (e.g., `redis-server` on `localhost:6379`).
   - Update `application.properties` with your Redis server's host and port if different.
2. **Configure Environment**:
   - Set a unique `HOSTNAME` environment variable on each Linux host:

```bash
export HOSTNAME=host1  # On first host
export HOSTNAME=host2  # On second host
```

   - Ensure both hosts use the same `application.properties` settings for `ema.host`, `ema.port`, `ema.service`, and `ema.item`.
3. **Dependencies**:
   - Ensure EMA Java libraries are available (e.g., via LSEG Developer Portal if not in Maven Central).
   - The `spring-boot-starter-data-redis` dependency handles Redis integration.
4. **Build and Deploy**:
   - Build the application on both hosts:

```bash
mvn clean install
```

   - Run the application on both hosts:

```bash
mvn spring-boot:run
```

[5] **Expected Behavior**:

- One host acquires the Redis `ACTIVE_HOST_KEY`, becomes active, and starts the `OmmConsumer`, subscribing to the specified item (e.g., `IBM.N`).
- The other host remains inactive, polling Redis for failover opportunities.
- If the active host's connection fails (detected via `StatusMsg`) or the host crashes (heartbeat expires), the standby host takes over within ~15 seconds.
- Market data updates (`RefreshMsg`, `UpdateMsg`) are printed to the console by the active host.

Testing the Setup

1. **Start Both Hosts**:
   - Run the application on both Linux hosts.
2. **Check Active Host**:
   - Use `redis-cli` to verify the active host:

     ```bash
     redis-cli GET ema:activeHost
     ```

     This should return the `hostId` of the active host (e.g., `host1`).
3. **Simulate Failure**:
   - Stop the active host's application (`Ctrl+C`) or disconnect its network to simulate a failure.
   - Within 15 seconds, the standby host should detect the expired heartbeat, acquire the `ACTIVE_HOST_KEY`, and start its consumer.
4. **Verify Data**:
   - Ensure the new active host receives `RefreshMsg` and `UpdateMsg` for the subscribed item.
   - Check Redis for the updated active host:

     ```bash
     redis-cli GET ema:activeHost
     redis-cli GET ema:heartbeat:<hostId>
     ```

Additional Notes

- **Authentication**: If your Refinitiv server requires authentication, add credentials to the `OmmConsumerConfig` (e.g., `username`, `password`, or token-based authentication as per LSEG documentation).
- **Multiple Servers**: To support primary and standby Refinitiv servers (as in the original C++ example), update the `OmmConsumerConfig` to include a `ChannelSet`:

```java
config.addChannel("Channel_Active", Map.create()
        .add(MapEntry.create("ChannelType", OmmConsumerConfig.ChannelType
                .add(MapEntry.create("Host", activeHost))
                .add(MapEntry.create("Port", activePort))));
config.addChannel("Channel_Standby", Map.create()
        .add(MapEntry.create("ChannelType", OmmConsumerConfig.ChannelType
                .add(MapEntry.create("Host", standbyHost))
                .add(MapEntry.create("Port", standbyPort))));
config.addChannelSet("ChannelSet_1", new String[]{"Channel_Active", "Chan
```

  Update `application.properties` with `ema.active.host`, `ema.active.port`, `ema.standby.host`, and `ema.standby.port`.
- **Redis High Availability**: For production, use Redis Sentinel or Redis Cluster to ensure Redis availability.
- **Performance Tuning**:
  - Adjust `HEARTBEAT_INTERVAL` (5 seconds) and `HEARTBEAT_TIMEOUT` (15 seconds) for faster or more reliable failover, balancing latency and false positives.
  - Increase the `dispatch` timeout (1000ms) if needed for high message volumes.
- **Error Handling**: The implementation includes basic error handling. Add logging (e.g., SLF4J) and retry logic for production use.
- **Scalability**: This supports two hosts (one active, one inactive). For more hosts, extend the leader election logic to prioritize standbys.

Why Redis Over EhCache?

- **Distributed Coordination**: Redis provides a true distributed key-value store, ensuring consistent state across hosts, unlike EhCache's local scope.
- **Reliability**: Redis's `setIfAbsent` and TTL features offer robust leader election, avoiding race conditions common in file-based approaches.
- **Simplicity**: Redis requires less custom code than file-based coordination or Terracotta with EhCache.

This implementation reverts to the Redis-based approach, refined for clarity and robustness, aligning with your preference to avoid the EhCache and file-based solution. For further details, refer to the LSEG Developer Community for EMA Java documentation or the Redis documentation for advanced configuration.

If you need specific adjustments (e.g., adding primary/standby server support, custom logging, or additional failover logic), please let me know!