Project : SwiftTrade

Aahan Piplani

January 19, 2025

1 Data Structure Selection

In optimizing the `SimpleWebSocketServer`, the following data structures and techniques were employed to ensure performance, scalability, and maintainability:

- **Unordered Sets (`unordered_set`):** Used for caching valid instruments to achieve $\mathcal{O}(1)$ average time complexity for lookups. This significantly reduces the validation overhead during high-frequency operations.
- **Unordered Maps (`unordered_map`):** Applied to store mappings such as subscriptions and client states, providing $\mathcal{O}(1)$ average time complexity for retrieval and updates. This is crucial for handling dynamic WebSocket clients and their associated subscriptions efficiently.
- **Hashing and Caching:** By caching valid instrument names in an `unordered_set`, redundant network calls are minimized. Instruments are only validated against the API once and cached thereafter.
- **Vectors (`vector`):** Used to manage sequential storage such as message queues in the WebSocket sessions. Vectors are efficient for data that needs frequent traversal with minimal insertions or deletions.
- **String Views (`string_view`):** Applied to avoid unnecessary string copying, reducing memory usage and increasing processing speed during message parsing.

These optimizations not only improve the responsiveness of the server under heavy load but also reduce network latency and memory overhead. The choice of data structures aligns with the real-time requirements of a WebSocket-based trading platform, where low latency and high throughput are essential.

1.1 Future Optimizations

To further enhance the performance and efficiency of the `SimpleWebSocketServer`, the following data structure optimizations can be considered:

- **Custom Allocators:**
  - Implement custom memory allocators for frequently used data structures like `unordered_set` and `unordered_map`.
  - This reduces the overhead associated with standard memory allocation and deallocation, especially under high-load conditions.
- **Lock-Free Data Structures:**
  - Replace standard containers with lock-free alternatives, such as `concurrent_unordered_map`, to reduce mutex contention and improve performance in multi-threaded scenarios.
  - This is particularly useful for high-frequency operations like client state management and subscription lookups.
- **Small-String Optimization:**
  - Optimize string handling by leveraging small-string optimizations (SSO), where strings below a certain length are stored directly within the object, avoiding heap allocation.
  - This can significantly improve performance during frequent string manipulations.
- **Sparse Hashing:**
  - Use sparse hash maps or sets (e.g., `google::sparse_hash_map`) for large datasets to minimize memory usage while maintaining efficient lookups.
  - This is especially advantageous when managing subscriptions for a large number of instruments or clients.
- **Batch Processing for Message Queues:**
  - Instead of processing messages individually, implement batch processing for the `vector`-based message queues.
  - This reduces the overhead of frequent queue operations and improves overall throughput.

- **Pre-allocated Buffers:**
  - Use pre-allocated buffers for WebSocket read and write operations to minimize dynamic memory allocations.
  - This approach reduces latency and improves real-time performance for high-frequency data streams.
- **Optimized Hash Functions:**
  - Replace default hash functions with domain-specific, high-performance hash functions optimized for instrument names or client identifiers.
  - This reduces collisions in `unordered_map` and `unordered_set`, enhancing lookup performance.

These future optimizations aim to build on the current implementation, addressing potential bottlenecks and further improving the server's ability to handle high loads and scale efficiently.

2 Thread Management

2.1 Thread Management Optimization

Thread management optimization in the system is implemented using the `std::thread` library to handle concurrent operations. Key features include the following:

- **Concurrency with Threads:** Threads are used to perform multiple tasks concurrently, such as establishing a WebSocket connection, reading messages, and processing market data updates. This ensures that the main thread remains unblocked and responsive.
  - *WebSocket Server and Data Streams:* Threads are created to manage WebSocket connections for each client, as well as handle real-time data streaming for each instrument that a client subscribes to.
  - *Cleaning and Maintaining Resources:* A dedicated cleanup thread (`cleanupThread_`) is used to periodically clean up expired subscriptions and stale sessions, ensuring the efficient use of system resources.

- **Mutexes for Synchronization:** To protect shared resources from concurrent access, `std::mutex` is used. For example, `dataMutex` and `tokenMutex` ensure that market data and authentication tokens are accessed in a thread-safe manner, preventing race conditions.
- **Atomic Flags for Control:** The use of `atomic<bool>` flags, such as `shuttingDown` and `running`, allows safe control over the execution flow of threads. These flags enable threads to shut down gracefully when required, ensuring clean termination of threads without data inconsistency.
- **Thread Pooling and Resource Management:**
  - The `SimpleWebSocketServer` class handles multiple client connections using weak pointers and message queues. This allows for efficient management of client subscriptions, with each client session running in a separate thread.
  - The server can scale to support multiple client connections without blocking, thanks to the use of threads for each session.
- **Asynchronous Operations:** The system leverages `boost::asio` for asynchronous I/O operations. The `doAccept` method, for example, uses `async_accept` to handle incoming client connections asynchronously, allowing the main thread to continue with other tasks while waiting for new connections.

2.2 Future Improvements

While the current implementation using threads and `boost::asio` is highly efficient, the following future optimizations can further enhance performance:

- **Thread Pooling:**
  - Instead of spawning a new thread for each client, a thread pool can be implemented to manage threads more effectively.
  - This approach reduces the overhead associated with thread creation and destruction, improves resource utilization, and ensures scalability for high client loads.
- **Reducing Mutex Contention:**
  - To minimize contention, the scope of mutex locks can be narrowed.

  - Lock-free data structures or read-write locks (e.g., `std::shared_mutex`) can be employed to optimize access to shared resources, especially in high-frequency operations like message processing.

- **Task Prioritization:**
  - Implement priority queues for tasks, allowing critical operations like market data updates to take precedence over less time-sensitive operations.
  - This ensures responsiveness under high workload conditions.

- **Load Balancing:**
  - Distribute WebSocket connections across multiple threads or server instances using a load balancer to avoid bottlenecks on a single CPU core.

- **Profiling and Monitoring:**
  - Use profiling tools to monitor thread behavior and identify bottlenecks, enabling targeted optimizations for thread management.

- **Event-Driven Architecture:**
  - Transition parts of the application to a fully event-driven model to further reduce the dependency on threads and improve CPU efficiency.

3 Memory Improvements

3.1 Robust Memory Management

The following changes were implemented to enhance memory management and ensure efficient resource utilization:

- **RAII for CURL Handles:** A wrapper class `CurlHandle` was introduced to manage the lifecycle of CURL handles. This ensures that CURL handles are automatically cleaned up when they go out of scope, preventing memory leaks. Each CURL handle is now encapsulated within an RAII object, ensuring cleanup occurs even if exceptions are thrown during execution.

- **Static Initialization for CURL:** The `curl_global_init()` and `curl_global_cleanup()` calls were replaced with a static guard class `CurlGlobalInit`. This ensures that global initialization and cleanup are performed only once, even if multiple instances of `DeribitApi` are created. The static guard guarantees that CURL's global resources are properly managed without redundant initialization.

- **Atomic Token Updates:** Tokens such as `accessToken_` and `refreshToken_` are now updated atomically using temporary variables. This prevents invalid states during token refresh operations, ensuring that existing tokens remain valid until the new tokens are successfully fetched and validated. This approach minimizes downtime and reduces the risk of failed API requests due to invalid tokens.

- **Thread-Safe Operations:** A mutex named `tokenMutex_` was introduced to guard sensitive member variables, such as access tokens and refresh tokens, ensuring thread-safe access and updates in multi-threaded environments. All operations that modify shared resources are now synchronized to prevent race conditions and data corruption.

- **Enhanced CURL Request Cleanup:** Headers used in CURL requests (e.g., authentication headers) are explicitly freed using `curl_slist_free_all()` after each request to ensure no memory is leaked. Each request's resources are cleaned up immediately after use, preventing accumulation of unused memory.

- **Memory Leak Checks with Valgrind:** Memory leak checks were introduced into the testing workflow using Valgrind. The application is now run with `valgrind --leak-check=full` to identify and resolve any memory leaks. This ensures that all allocated resources are properly released and no leaks occur during execution.

- **Graceful Signal Handling:** A global signal handler was implemented to manage cleanup upon receiving `SIGINT` signals.All the sessions are freed , The handler ensures that WebSocket servers and active streams are properly stopped, and memory is released, enabling graceful termination of the application and preventing memory leakage which is validated by valgrind. sample output is shown below :

These changes ensure robust memory management, prevent resource leaks, and improve the stability and reliability of the `SwiftTrade` app in both single-threaded and multi-threaded environments.

3.2 Efficient Memory Management

To enhance memory efficiency and optimize resource usage in the `DeribitMarketData` class, the following improvements were implemented:

- **Buffer Pool Implementation:** A memory pool was introduced for `Beast::flat_buffer` objects. This pool pre-allocates buffers and reuses them across multiple operations, significantly reducing allocation overhead and fragmentation.
- **Efficient String Handling:** The use of `std::string` was replaced with `std::string_view` wherever possible for read-only operations. This minimizes unnecessary memory allocation and copying, improving performance.
- **Thread Safety with RAII:** Thread management was enhanced using RAII principles. The WebSocket thread is now properly managed with an `std::atomic<bool>` flag to ensure safe termination.
- **Optimized Object Lifecycle:** Placement new was used in conjunction with the memory pool to construct objects efficiently and deallocate them when no longer needed.
- **Reduced Dynamic Memory Usage:** By reusing pre-allocated resources (e.g., buffers), dynamic memory usage was minimized, leading to better predictability and lower latency during runtime.

These memory improvements ensure the `DeribitMarketData` class operates efficiently with reduced memory overhead and improved resource reuse.

3.3 Future Improvements

To further improve the memory management of the system, including leveraging advanced techniques like memory pooling, the following approaches can be explored:

- **Enhanced Memory Pool Implementation:** The current implementation uses a basic memory pool for managing objects like WebSocket sessions. Future work could include:
  - Dynamic resizing of the memory pool to handle varying workloads.
  - Monitoring pool usage to identify underutilization or overflows.

  - Incorporating thread-safe operations to allow concurrent access to the pool in multi-threaded environments.

- **Custom Allocators:** Develop custom allocators integrated with STL containers (e.g., `std::vector`, `std::map`) to optimize memory allocation and reduce fragmentation for frequently used data structures.

- **Efficient Buffer Management:** Implement a buffer pool to manage frequently allocated buffers (e.g., for WebSocket message handling) to minimize allocation overhead and reuse buffers efficiently.

- **Smart Pointer Integration:** Utilize `std::shared_ptr` or `std::unique_ptr` with custom deleters for automatic resource cleanup, ensuring no memory is leaked even in the presence of exceptions.

- **Garbage Collection for Idle Resources:** Introduce a garbage collection mechanism to periodically clean up unused or idle objects from the memory pool and other resource containers.

- **Shared Resource Optimization:** For shared resources like CURL handles or SSL contexts, implement pooling mechanisms to manage limited resources efficiently without repeated allocation and deallocation.

- **Adaptive Memory Strategies:** Implement adaptive memory management strategies to optimize allocation based on real-time workload patterns. This could include adjusting the pool size or buffer sizes dynamically during runtime.

- **Region-Based Memory Allocation:** Explore region-based memory allocation where a single memory block is allocated for a group of related objects and freed all at once, reducing allocation overhead.

These enhancements aim to reduce runtime overhead, improve scalability, and ensure robustness in memory management, making the system more efficient and reliable under various workloads.

4 Improving Network Latency

To optimize network latency in the `DeribitApi` class, the following changes were implemented:

- **Enabled HTTP/2:** By setting the option `CURL_HTTP_VERSION_2TLS`, the communication protocol now uses HTTP/2, which provides multiplexing and reduced overhead compared to HTTP/1.1.
- **Connection Reuse:** Options `CURL_FORBID_REUSE` and `CURL_FRESH_CONNECT` were adjusted to allow persistent connections, reducing the need to establish a new connection for every request.
- **SSL Session Caching:** Enabled SSL session caching using `CURL_SSL_SESSIONID_CACHE`, which minimizes the time spent on SSL handshakes for repeated connections.
- **Timeout Settings:** A connection timeout of 10 seconds was set using `CURL_TIMEOUT` to prevent hanging connections and ensure timely retries under slow network conditions.
- **Persistent Headers:** HTTP headers such as `Content-Type` were made persistent across requests by initializing them once and reusing them. This avoids the repeated allocation and deallocation overhead.
- **Static JSON Parsing:** Static objects were used for JSON response parsing to reduce the memory allocation overhead and improve performance for high-frequency requests.
- **Text Frames for Efficiency:** WebSocket communication was optimized by enabling text frames using `ws_.binary(false)`, which reduces overhead for JSON-based messages.
- **Centralized Subscription Logic:** The subscription and authentication processes were encapsulated in dedicated functions, streamlining code and enabling easier reuse.
- **Latency Monitoring:** Detailed latency measurement was added to track the performance of critical operations such as WebSocket reads, writes, and connection closures.
- **Connection Reuse for WebSocket:** Moved resolver and context initialization to the constructor to reuse resources across connections, improving efficiency.
- **Instrument Validation Caching:** Implemented a caching mechanism using a static `unordered_set` to store validated instruments. This avoids redundant network requests for frequently validated instruments.

- **Efficient WebSocket Handling:** Retained WebSocket connections for validation and reused SSL/TLS layers to avoid establishing new connections for every request.
- **Optimized Network Validation Logic:** Simplified and encapsulated network validation logic to reduce overhead. Only instruments not found in the cache trigger a server query for validation.
- **Thread Safety in Caching:** Ensured thread safety by limiting write operations to the cache, thus enabling concurrent validations without contention.
- **Latency Reduction with Local Data:** By using a local cache for instruments, significantly reduced network latency when handling repeated subscription requests.
- **Graceful Error Handling:** Enhanced error handling during WebSocket operations to ensure minimal disruption during transient failures, logging errors without compromising performance.

These changes collectively enhance the network communication efficiency, ensuring faster and more reliable API interactions with the Deribit server.

4.1 Future Improvements

While significant optimizations have been implemented to improve network latency, there are additional measures that can be considered for future enhancements:

- **Load Balancing:**
  - Distribute WebSocket and API requests across multiple servers using a load balancer to prevent overloading a single server and reduce response times.
- **Zero-Copy Networking:**
  - Implement zero-copy mechanisms to minimize memory copying during data transfer, especially for high-frequency WebSocket messages.
- **HTTP/3 Support:**

  - Upgrade the communication protocol to HTTP/3 to leverage its lower latency, improved congestion control, and connection resilience features.

- **Connection Pooling:**
  - Maintain a pool of persistent WebSocket connections to handle multiple client requests efficiently, reducing the overhead of establishing new connections.

- **Distributed Caching:**
  - Employ distributed caching solutions, such as Redis or Memcached, to synchronize cached instrument validation data across multiple server instances.

- **Protocol Buffers for Message Serialization:**
  - Replace JSON with Protocol Buffers for serializing and deserializing messages to reduce message size and processing time.

- **Dynamic Timeout Adjustment:**
  - Implement adaptive timeout mechanisms that adjust based on network conditions to ensure optimal responsiveness without unnecessary delays.

- **Multiplexed Subscriptions:**
  - Consolidate multiple subscriptions into a single WebSocket channel for each client, reducing the overhead of maintaining separate channels.

- **Monitoring and Auto-Scaling:**
  - Integrate monitoring tools to track network latency and auto-scale server resources during peak loads to maintain consistent performance.

These future improvements aim to further enhance the scalability, efficiency, and responsiveness of the system, ensuring it can handle larger workloads and adapt to evolving network demands.

5 CPU Optimization

To enhance the performance of the `SimpleWebSocketServer` and ensure efficient CPU utilization, this section outlines optimizations that can be incorporated into the current implementation and future improvements.

5.1 Incorporated Optimizations

The following techniques are implemented to optimize CPU usage:

- **Efficient Data Structures:** `unordered_set` and `unordered_map` are used to minimize lookup and insertion time to $\mathcal{O}(1)$ on average. This reduces the computational overhead in managing subscriptions and client states.
- **Thread Pooling:** Use of thread pools for managing WebSocket sessions instead of creating new threads for each client. This reduces the overhead of frequent thread creation and destruction.
- **Asynchronous I/O:** Boost.Asio's asynchronous operations are utilized for WebSocket communication, reducing CPU blocking time and ensuring efficient context switching between tasks.
- **Lazy Evaluation:** Operations such as subscription validation and network calls are deferred until necessary, avoiding unnecessary computations.
- **Reduced Copying:** Incorporation of `std::string_view` in functions that parse or handle string inputs to minimize memory copying and allocation overhead.
- **Batch Processing:** Messages to clients are batched when possible to reduce the number of write calls, thus lowering the CPU load associated with network I/O.

5.2 Implemented Optimizations

The following CPU optimization techniques have been implemented in the `SimpleWebSocketServer` to enhance performance and reduce computational overhead:

- **Efficient Data Structures:**

  - `unordered_map` and `unordered_set` are used for O(1) average lookup time, improving the performance of managing subscriptions and caching.
  - `vector` is leveraged for sequential access in message queues, benefiting from its cache-friendly memory layout.

- **Thread Management:**
  - `std::thread` is utilized for concurrent operations such as server acceptance and cleanup, ensuring efficient use of CPU resources.

- **Memory Pooling:**
  - A memory pool is implemented for reusable objects, reducing the cost of frequent memory allocations and deallocations.

- **Avoiding Redundant Computations:**
  - Cached valid instruments are used to prevent repeated network calls for validation, reducing unnecessary computations.

- **Asynchronous I/O Operations:**
  - Boost.Asio's asynchronous read and write operations are utilized for WebSocket and REST communications, preventing blocking operations and maximizing CPU utilization.

5.3 Future Optimizations

The following optimizations can be considered for future implementations to further enhance CPU efficiency:

- **Vectorization:** Replace scalar operations in computationally intensive tasks with vectorized instructions using libraries like Intel TBB or SIMD (Single Instruction, Multiple Data).

- **Load Balancing:** Distribute WebSocket connections across multiple server threads or processes to ensure even CPU load distribution.

- **Priority Scheduling:** Implement priority-based task scheduling for high-priority operations like market data updates to ensure real-time responsiveness.

- **Parallel Parsing:** For large JSON responses, use multi-threaded or parallel parsing libraries to split the workload and reduce parsing time.

- **Caching Strategies:** Introduce smarter caching strategies that involve time-to-live (TTL) and least recently used (LRU) eviction policies for instruments and subscriptions.
- **Adaptive Polling:** Replace fixed polling intervals with adaptive intervals based on system load to avoid unnecessary CPU cycles during low activity.
- **Binary Protocols:** Transition from JSON-based messaging to a lightweight binary protocol to reduce the CPU overhead of encoding/decoding operations.

These optimizations aim to achieve a balance between computational efficiency and scalability, ensuring that the server can handle a high number of concurrent connections while maintaining low latency.