CMPE 275 Grand Challenge Report

Under the guidance of Professor John D.Gash

Teammates:

017463602 - Yeshvanth Raju Kurapati
017405765 - Sai Deekshith Katukojwala
016990558 - Mahendra Chittupolu
017046939 - Maheedhar

Table of Contents:

Project Overview:

High-Performance Distributed Task Scheduler Using gRPC

This project presents a high-performance distributed task scheduler utilizing gRPC (Google Remote Procedure Call) to facilitate efficient communication across multiple nodes. The system is designed to optimize task distribution and execution in a scalable, fault-tolerant, and dynamic manner, addressing modern distributed computing requirements.

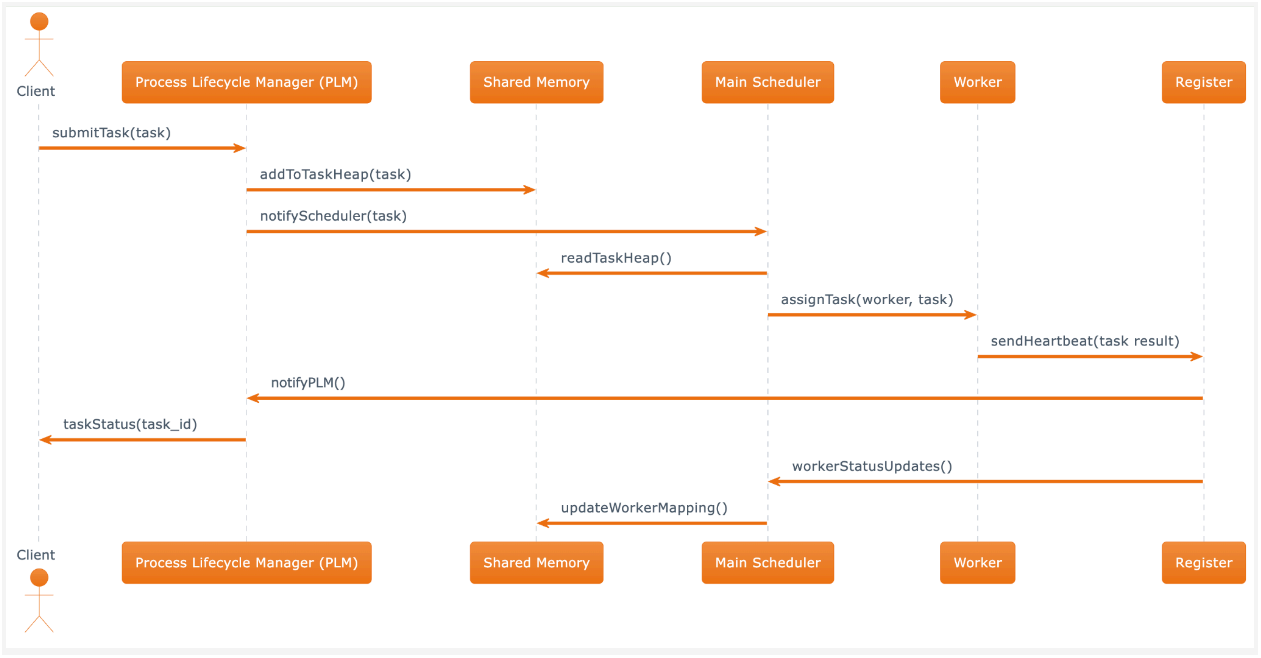
Motivation and Context

Efficient task scheduling and execution across multiple nodes are critical in distributed computing for maximizing performance and resource utilization. Traditional task schedulers often struggle with scalability, fault tolerance, and communication efficiency. Our project addresses these challenges by implementing a distributed task scheduler using gRPC, offering robust, high-performance communication capabilities.

System Architecture

The architecture comprises several key components: the Client, Worker, Process Lifecycle Manager (PLM), Main Scheduler, and Register. Each component plays a crucial role in ensuring seamless task management and execution.

- Client: Submits tasks via gRPC and handles asynchronous task responses. The client interacts with the PLM to initiate tasks and receives results upon completion.
- Worker: Executes tasks assigned by the scheduler and sends periodic heartbeat signals to the Register. This ensures dynamic management of worker nodes and prompt failure detection.
- Process Lifecycle Manager (PLM): Acts as an intermediary between the client and scheduler. It receives tasks from clients, assigns task IDs, manages the task queue, and maintains task-worker mappings.
- Main Scheduler: Core component responsible for task allocation. It reads tasks from the task heap, selects appropriate workers from the worker heap, and assigns tasks based on advanced scheduling algorithms to optimize distribution and load balancing.
- Register: Maintains a registry of all clients and workers, continuously monitoring system health through heartbeat signals. It updates worker status, detects node failures, and informs the PLM and Main Scheduler for seamless task reassignment and fault tolerance.



Communication Protocol and Data Flow

The system leverages gRPC for efficient, low-latency communication between components. The communication protocol and data flow are as follows:

1. Task Submission: The client submits tasks to the PLM using the submitTask gRPC method. The PLM assigns a task ID and adds the task to the task heap.
2. Task Assignment: The Main Scheduler reads tasks from the task heap, selects available workers from the worker heap, updates the task-worker mapping, and sends task details to the selected worker using gRPC.
3. Task Execution and Result Reporting: Workers execute tasks and send results back to the scheduler within heartbeat messages. The Register reads heartbeats, updates task status, and communicates completion to the PLM.
4. Result Delivery: The PLM retrieves task details from the task-worker mapping and sends the result to the client using gRPC, updating the mapping to reflect completion.
5. Failure Handling: If a worker node fails (detected by missing heartbeats), the Register marks the worker as dead, updates the worker heap, and informs the PLM. The PLM reassigns tasks to available workers, ensuring continuous processing.

Performance and Scalability

Using gRPC and shared memory enhances the system's performance by enabling efficient communication and data sharing between processes. Shared memory stores task and worker heaps, allowing quick access and updates by the Main Scheduler and PLM, reducing inter-process communication overhead and ensuring low-latency task management. The system is designed for horizontal scaling by adding more worker nodes and dynamically distributing the load. The Main Scheduler's advanced algorithms ensure optimal load balancing, while the Register's real-time monitoring capabilities maintain system health and performance.

Background Work:

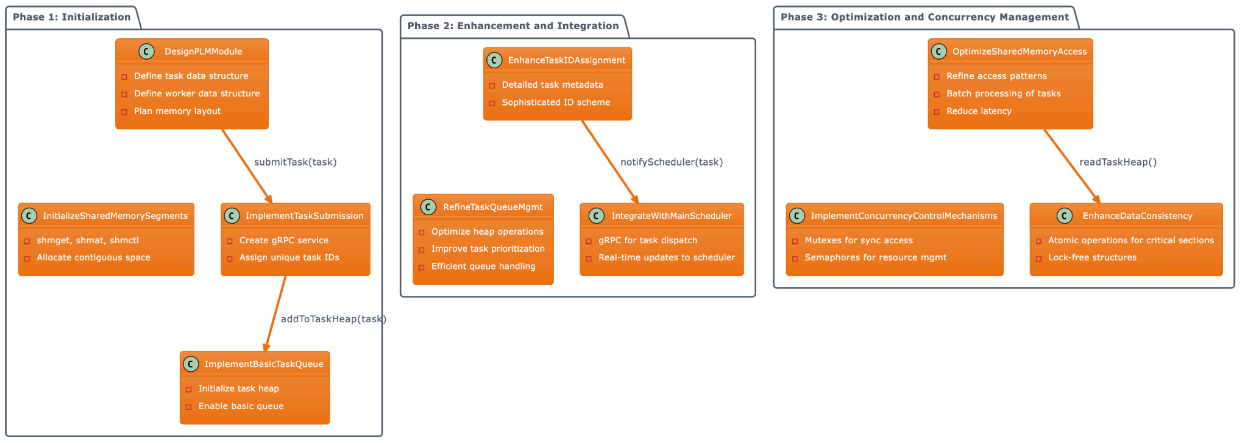
Research and Preliminary Studies

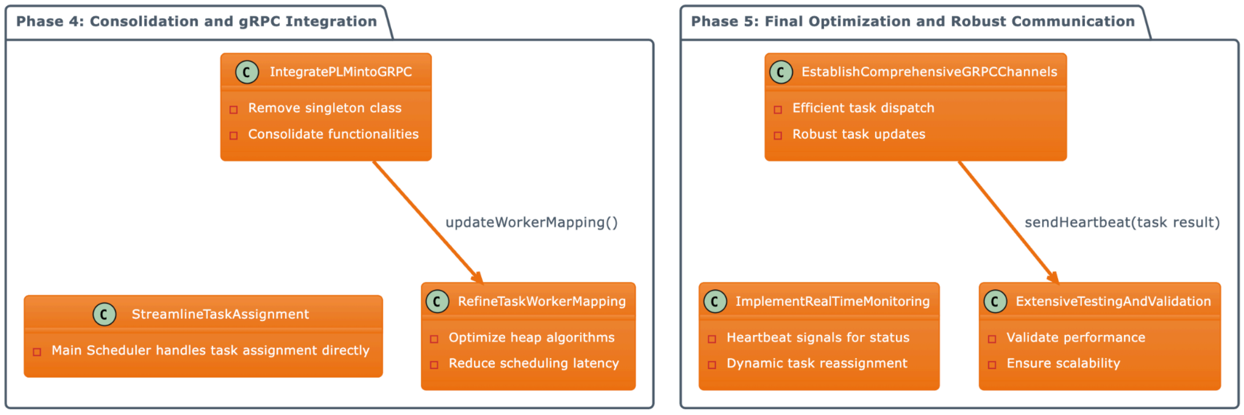
The foundation of our project is built upon extensive research into distributed systems, task scheduling algorithms, and high-performance communication protocols. To design and implement an effective distributed task scheduler using gRPC, we conducted a thorough review of existing literature and technologies, focusing on the following areas:

1. Distributed Systems and Task Scheduling:
    - We studied the principles of distributed computing, including the challenges of scalability, fault tolerance, and efficient resource utilization. Task scheduling in distributed systems involves the allocation of tasks to available resources in a manner that maximizes system performance while minimizing latency and ensuring fault tolerance. Key insights were drawn from "Distributed Systems: Principles and Paradigms" by Andrew S. Tanenbaum and Maarten Van Steen, which provided a comprehensive overview of distributed system architectures and task scheduling strategies.
2. High-Performance Communication Protocols:
    - The selection of gRPC as the communication protocol was influenced by its ability to provide high-throughput, low-latency communication, which is crucial for the performance of distributed systems. We analyzed the advantages of gRPC over other communication protocols such as REST and SOAP, particularly in terms of performance and ease of integration with various programming languages. Our understanding was deepened by reviewing technical specifications and performance benchmarks of gRPC, as outlined in the paper "gRPC: A High-Performance, Open-Source Universal RPC Framework" by Google, which provided detailed insights into its architecture, performance characteristics, and use cases.

PLM Team Project Description:

From the PLM team's perspective, the project involved designing and implementing the Process Lifecycle Manager (PLM) to efficiently handle task management within a distributed task scheduler. Starting with the initialization of shared memory and task ID assignment, the team focused on optimizing task queues and integrating advanced scheduling algorithms. They refined concurrency control mechanisms to ensure data consistency and streamlined the system by integrating PLM functionalities into gRPC. The final phase saw the establishment of robust gRPC communication channels, real-time monitoring, and comprehensive validation, culminating in a highly efficient and scalable distributed task scheduler.


Phase 1: Initialization
C DesignPLMModule
Define task data structure
Define worker data structure
Plan memory layout
submitTask(task)
C InitializeSharedMemorySegments
shmget, shmat, shmctl
Allocate contiguous space
C ImplementTaskSubmission
Create gRPC service
Assign unique task IDs
addToTaskHeap(task)
C ImplementBasicTaskQueue
Initialize task heap
Enable basic queue
Phase 2: Enhancement and Integration
C EnhanceTaskIDAssignment
Detailed task metadata
Sophisticated ID scheme
notifyScheduler(task)
C RefineTaskQueueMgmt
Optimize heap operations
Improve task prioritization
Efficient queue handling
C IntegrateWithMainScheduler
gRPC for task dispatch
Real-time updates to scheduler
Phase 3: Optimization and Concurrency Management
C OptimizeSharedMemoryAccess
Refine access patterns
Batch processing of tasks
Reduce latency
readTaskHeap()
C ImplementConcurrencyControlMechanisms
Mutexes for sync access
Semaphores for resource mgmt
C EnhanceDataConsistency
Atomic operations for critical sections
Lock-free structures

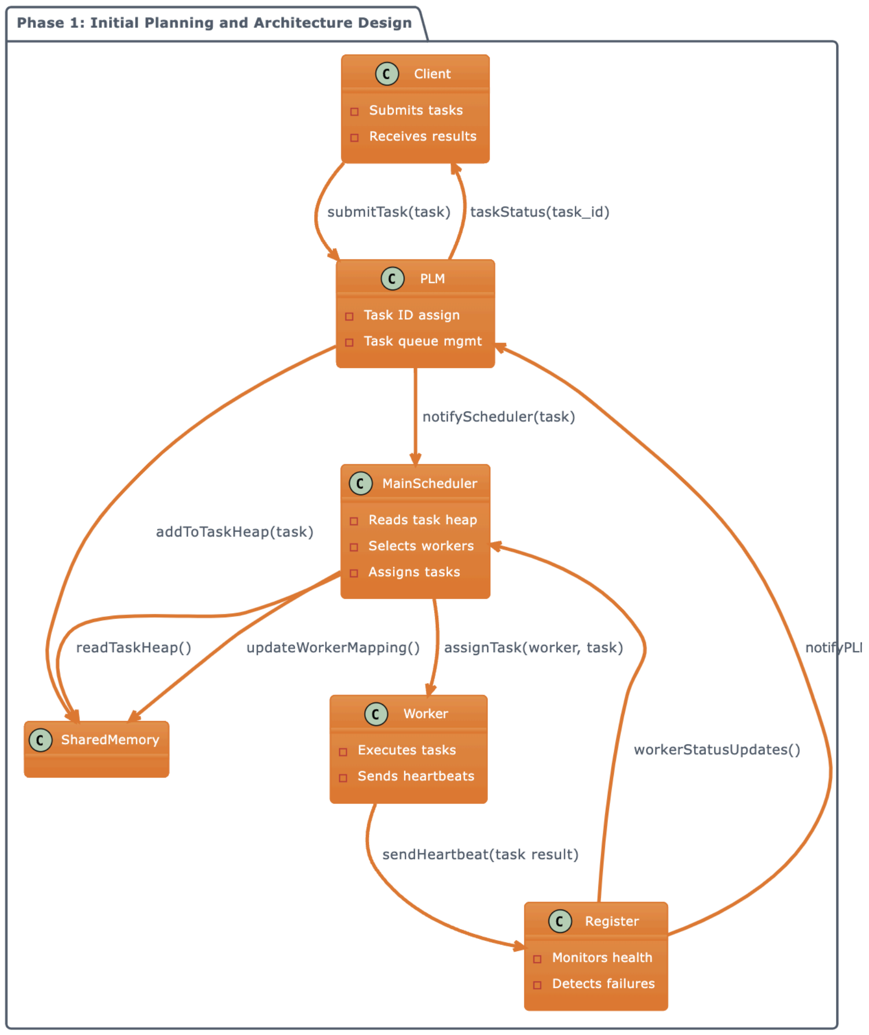

Phase 4: Consolidation and gRPC Integration
C IntegratePLMintoGRPC
Remove singleton class
Consolidate functionalities
updateWorkerMapping()
C StreamlineTaskAssignment
Main Scheduler handles task assignment directly
C RefineTaskWorkerMapping
Optimize heap algorithms
Reduce scheduling latency
Phase 5: Final Optimization and Robust Communication
C EstablishComprehensiveGRPCChannels
Efficient task dispatch
Robust task updates
sendHeartbeat(task result)
C ImplementRealTimeMonitoring
Heartbeat signals for status
Dynamic task reassignment
C ExtensiveTestingAndValidation
Validate performance
Ensure scalability

Project Phases and Key Technical Improvements

Phase	**Key Technical Improvements/Changes**
Phase 1	Initial Planning and Architecture Design
	- Designed initial architecture with Client, Worker, PLM, Main Scheduler, and Register.
	- Implemented basic task submission and worker assignment using gRPC.
	- Established shared memory for task and worker heaps.
Phase 2	Refinement of PLM and Scheduler Integration
	- Enhanced PLM with task ID assignment and detailed task queue management.
	- Integrated advanced scheduling algorithms in Main Scheduler.
	- Optimized shared memory access patterns for reduced latency.
Phase 3	Optimization and Concurrency Management
	- Removed MPI, using continuous shared memory reading for updates.
	- Introduced singleton class for centralized task and worker management.
	- Enhanced shared memory locking mechanisms for data consistency.
Phase 4	Consolidation and Integration of PLM into gRPC
	- Integrated PLM functionalities into gRPC, eliminating singleton classes.
	- Moved task assignment to Main Scheduler, improving data sharing via shared memory.
Phase 5	Final Optimization and Full Communication Establishment
	- Established robust gRPC communication across all components.
	- Implemented heartbeat signals from workers to Register with results piggybacked on heartbeats.
	- Developed sophisticated task reassignment strategies for dynamic handling of worker failures.

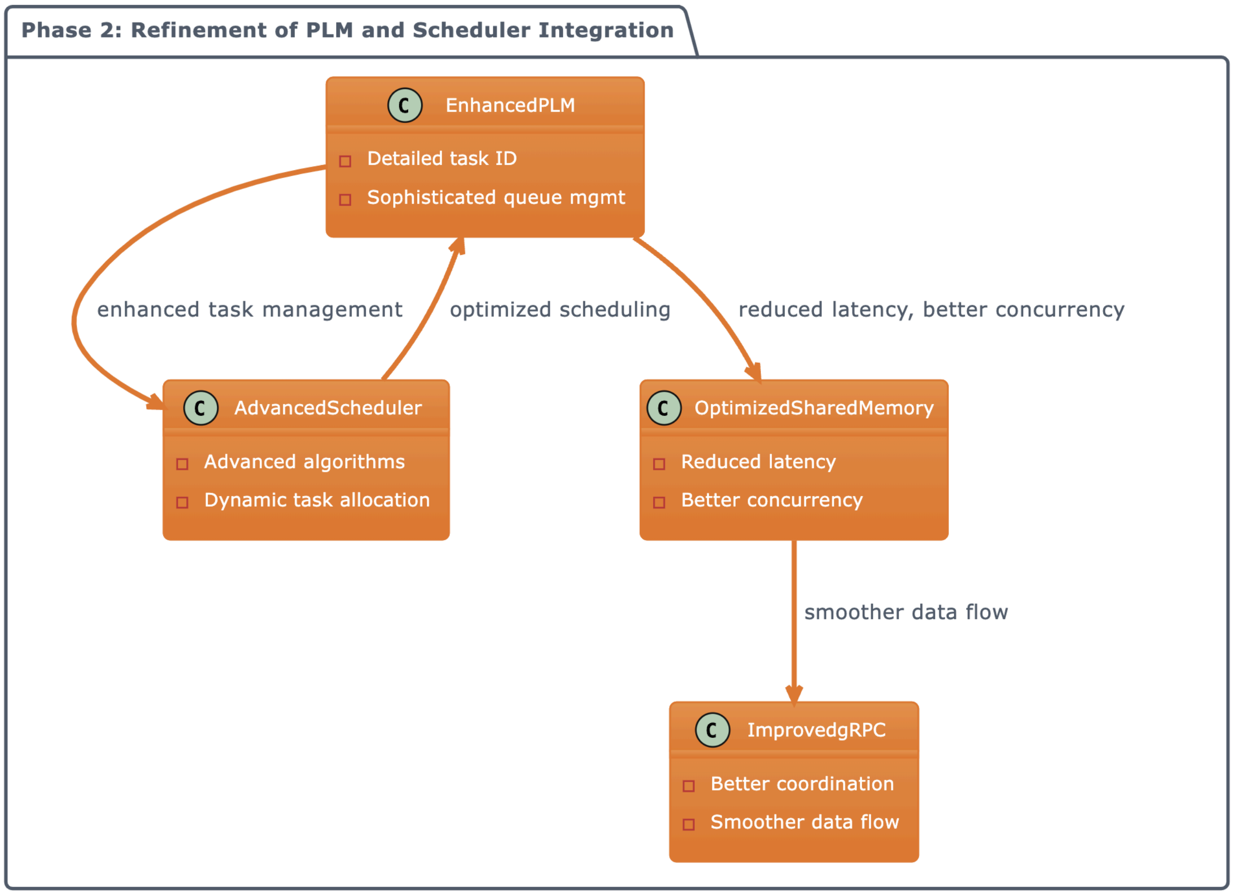
Phase 1: Initial Planning and Architecture Design

https://lucid.app/lucidchart/ca5e9169-b038-400b-8eb4-76e449eccb24/edit?viewport_loc=-1047%2C138%2C4419%2C2392%2C0_0&invitationId=inv_58301903-06af-4a2a-83be-d0cdd904016b



We established the core architecture with essential components and implemented task submission, ID assignment, and queue management using gRPC. A basic scheduling algorithm was developed for task distribution, enabling task execution and worker heartbeat monitoring. Challenges included the need for advanced scheduling features, performance bottlenecks with shared memory access, limited fault tolerance, and scalability issues. We demonstrated a functional prototype with initial task and worker management systems and implemented basic fault detection. Remaining issues include the need for improved scheduling algorithms, optimized shared memory access, enhanced fault tolerance, and architectural refinements for better scalability.

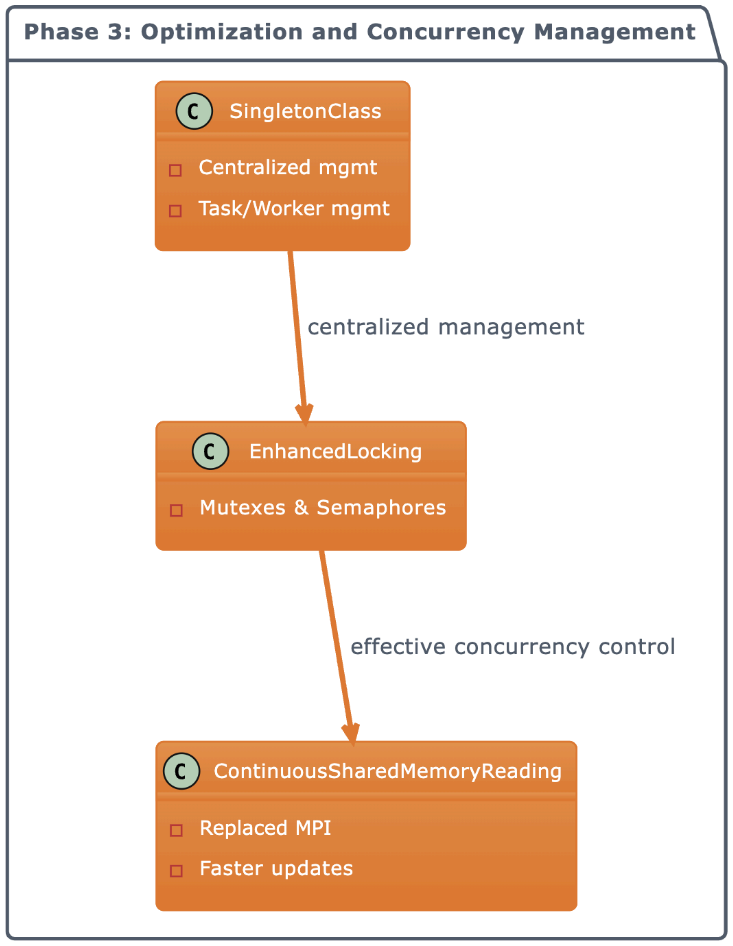
Phase 2: Refinement of PLM and Scheduler Integration



In Phase 2, we refined the PLM by enhancing task ID assignment and queue management, improving task tracking and handling. The Main Scheduler was integrated with advanced algorithms for dynamic task allocation. Shared memory access patterns were optimized, reducing latency and improving concurrency. These enhancements boosted system efficiency, ensuring faster task processing and better resource utilization.

New challenges included increased complexity from advanced scheduling algorithms, synchronization issues with concurrent memory access, and the need for robust inter-process communication. We overcame these by integrating and testing advanced algorithms, implementing efficient locking mechanisms for shared memory, and improving gRPC communication for better coordination and data flow.

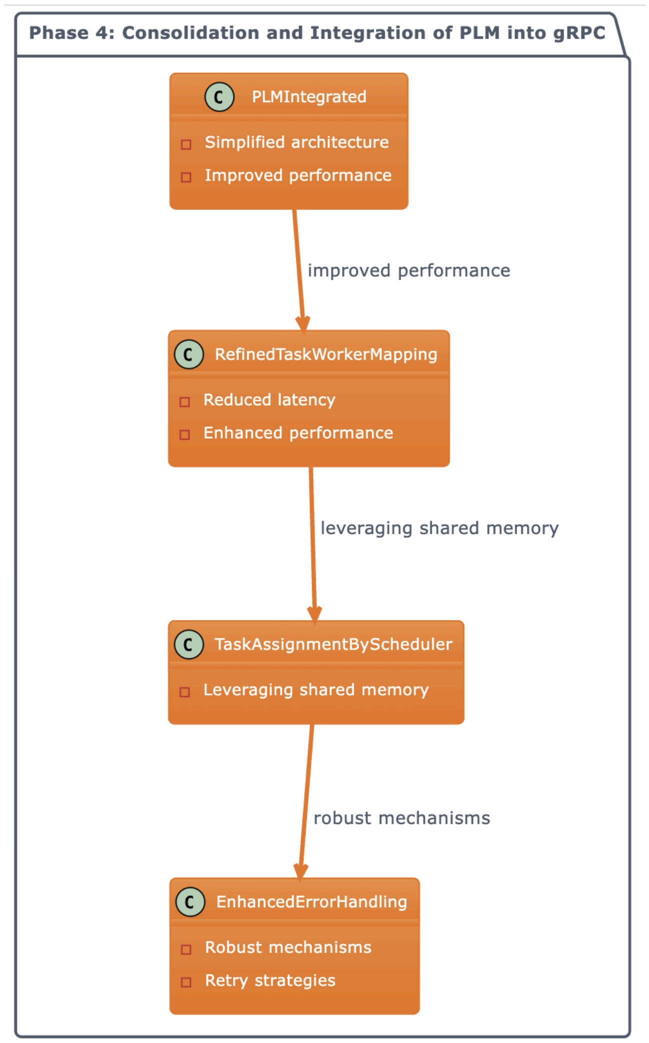
Phase 3: Optimization and Concurrency Management



In Phase 3, we optimized the system and enhanced concurrency management. We replaced MPI with continuous shared memory reading for faster task and worker status updates. A singleton class was introduced for centralized task and worker management, improving coordination between the PLM and Main Scheduler. Enhanced locking mechanisms, including mutexes and semaphores, ensured effective concurrency control, preventing race conditions and ensuring data consistency in a multi-threaded environment. These improvements resulted in more efficient task handling and reduced latency.

New challenges included managing concurrent access to shared memory, preventing data corruption, and handling the complexity of centralized management. We overcame these by implementing advanced locking mechanisms, utilizing the singleton class for streamlined management, and applying effective concurrency control techniques, such as lock-free data structures and atomic operations.

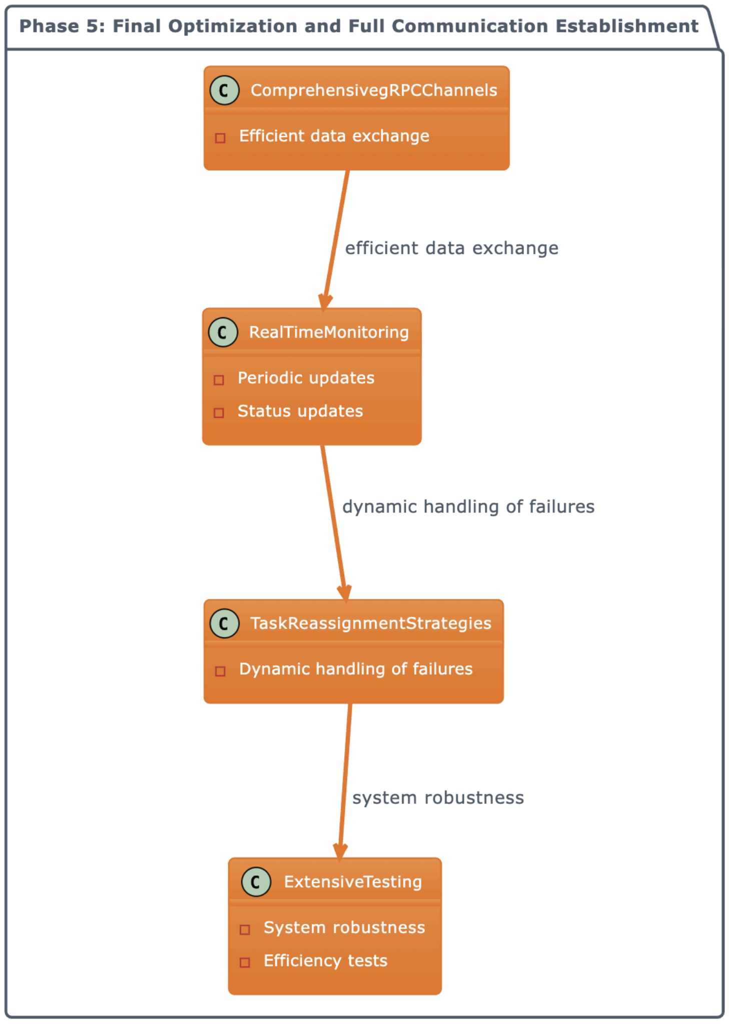
Phase 4: Consolidation and Integration of PLM into gRPC



In Phase 4, we consolidated the system architecture by integrating the Process Lifecycle Manager (PLM) directly into the gRPC framework. This eliminated the need for separate singleton classes, streamlining the overall design. Task assignment was moved from the PLM to the Main Scheduler, leveraging shared memory for efficient data sharing. We refined task-worker mapping and heap management algorithms to reduce latency and enhance performance. Robust error handling and retry mechanisms were implemented to ensure reliable task assignment in case of worker node failures.

New challenges included the complexity of integrating PLM functionalities into gRPC, optimizing data sharing through shared memory, and developing robust error handling. We overcame these by seamlessly integrating PLM into gRPC, optimizing shared memory usage, and implementing comprehensive error handling and retry mechanisms. This phase significantly enhanced system robustness and efficiency, improving reliability and scalability.

Phase 5: Final Optimization and Full Communication Establishment


Phase 5: Final Optimization and Full Communication Establishment
ComprehensivegRPCChannels
Efficient data exchange
efficient data exchange
RealTimeMonitoring
Periodic updates
Status updates
dynamic handling of failures
TaskReassignmentStrategies
Dynamic handling of failures
system robustness
ExtensiveTesting
System robustness
Efficiency tests

Execution and Achievements

In Phase 5, the primary focus was on final optimization and establishing robust communication channels across all system components. We implemented comprehensive gRPC communication channels between the Client, Worker, PLM, Main Scheduler, and Register, ensuring seamless and efficient data exchange. Workers were enhanced to send periodic heartbeat signals to the Register, with task results piggybacked on these heartbeats, enabling real-time monitoring and status updates. The Register's functionalities were expanded to continuously update worker status and notify the PLM of any changes. Sophisticated task reassignment strategies were developed for the PLM and Main Scheduler to dynamically handle worker failures, ensuring uninterrupted task processing. Extensive testing and validation were conducted to guarantee system robustness, efficiency, and scalability.

New Challenges

- Real-Time Monitoring: Implementing efficient real-time monitoring and status updates via heartbeats.
- Dynamic Task Reassignment: Developing and integrating sophisticated strategies for dynamic task reassignment in case of worker failures.

Overcoming Challenges

- Optimized gRPC Channels: Established optimized gRPC communication channels, minimizing overhead and ensuring efficient data exchange.
- Effective Monitoring: Successfully implemented real-time monitoring through periodic heartbeats, providing accurate and timely status updates.
- Advanced Reassignment Strategies: Developed and tested robust task reassignment strategies, ensuring continuous task processing and system resilience.

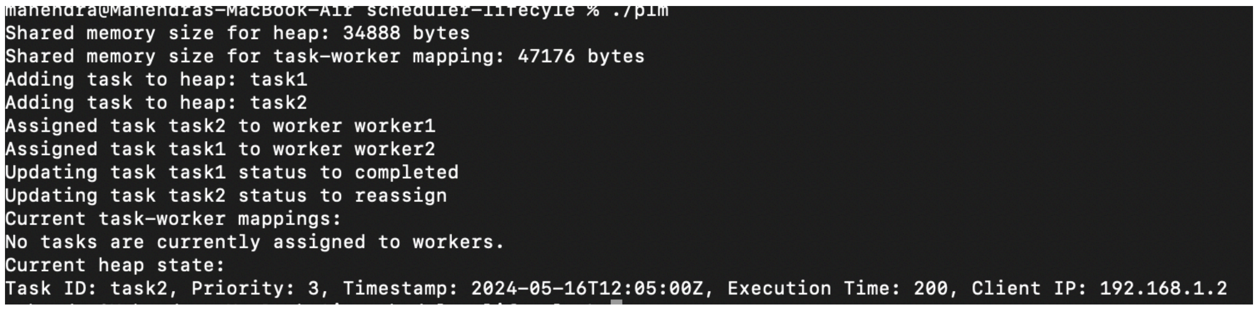
Through these final optimizations and the establishment of robust communication mechanisms, Phase 5 culminated in a highly efficient, reliable, and scalable distributed task scheduler, capable of handling complex and dynamic workloads in modern distributed computing environments.

Execution result:

Command:

```
g++ -o plm plm.cpp heap.cpp -lpthread

./plm
```



```
mahendra@Mahendras-MacBook-Air scheduler-lifecyle % ./plm
Shared memory size for heap: 34888 bytes
Shared memory size for task-worker mapping: 47176 bytes
Adding task to heap: task1
Adding task to heap: task2
Assigned task task2 to worker worker1
Assigned task task1 to worker worker2
Updating task task1 status to completed
Updating task task2 status to reassign
Current task-worker mappings:
No tasks are currently assigned to workers.
Current heap state:
Task ID: task2, Priority: 3, Timestamp: 2024-05-16T12:05:00Z, Execution Time: 200, Client IP: 192.168.1.2
```

Implementation Details: Shared Memory Utilization

Choice and Implementation of Shared Memory

- Shared memory was selected for its low-latency, high-throughput communication capabilities, which are essential for efficient inter-process communication (IPC) in our distributed task scheduler. We initialized shared memory segments using POSIX-compliant system calls (shmget, shmat, shmctl) and carefully planned memory allocation to ensure contiguous space for task and worker heaps. Task heaps were implemented as priority queues for efficient task management based on priority, while worker heaps managed availability and load using a similar priority queue structure. Custom data structures for tasks and workers included metadata such as IDs, priorities, and statuses, with pointers managed to ensure valid references within shared memory, preventing issues like dangling pointers.

- To ensure data consistency and prevent race conditions, we used mutexes and semaphores for synchronizing access to shared memory and employed atomic operations for critical sections to enable lock-free, thread-safe modifications. Memory barriers were implemented to prevent CPU memory operation reordering, and techniques to maintain cache coherence ensured immediate visibility of changes across processes. Robust error handling mechanisms were integrated to manage memory allocation failures and access violations, with fallback procedures allowing for graceful degradation or reinitialization of shared memory segments. Access patterns were optimized for minimal contention and maximal throughput, with batch processing employed to reduce synchronization points and read/write operations profiled and tuned for efficiency.

Integration with System Components

- PLM and Main Scheduler: Utilized shared memory for managing tasks and workers. The PLM added tasks to the task heap, and the Main Scheduler assigned tasks based on heap data.
- Workers: Updated their status and retrieved tasks via shared memory. Heartbeat signals included updates to their status and task completion.
- Register: Monitored worker statuses and updated shared memory upon detecting failures, informing the PLM and Main Scheduler for reassignment

Advantages of Using Shared Memory in Our Project

- Low Latency: Direct memory access minimizes communication delays, crucial for real-time task scheduling.
- High Throughput: Supports rapid data exchange between PLM, Main Scheduler, and Workers, ensuring efficient task handling.
- Efficient Resource Utilization: Reduces overhead by eliminating data copying, leading to faster task assignments and updates.
- Scalability: Facilitates horizontal scaling by enabling quick adjustments to handle more tasks and workers as the system grows.

Best Optimal Shared Memory Size and How to Choose It

Determining Optimal Size for Our Project

To determine the optimal size for shared memory in our distributed task scheduler, we first estimated the memory requirements for each task and worker. Each task required approximately 128 bytes, including task ID, priority, status, and metadata. Each worker needed about 256 bytes for similar metadata. We then defined the peak number of concurrent tasks and workers, estimating 10,000 tasks and 1,000 workers. Using these estimates, we calculated the base memory requirements by multiplying the memory per task and worker by their respective maximum counts. Additionally, we allocated memory for the data structures managing tasks and workers, estimated at 0.5 MB, and added a buffer for future scalability, which was 1 MB.

Shared Memory Size Calculation

Calculation	**Value**
Memory per Task	128 bytes
Memory per Worker	256 bytes
Maximum Concurrent Tasks	10,000 tasks
Maximum Concurrent Workers	1,000 workers
Base Memory for Tasks	1,280,000 bytes (1.28 MB)
Base Memory for Workers	256,000 bytes (0.256 MB)
Data Structure Overhead	0.5 MB

Scalability Buffer	1 MB
Total Shared Memory Size	3.036 MB
Final Shared Memory Size (rounded up)	4 MB

Final Shared Memory Size

256 MB.

Why Multiple gRPC Services Were Used

In our distributed task scheduler project, multiple gRPC services were implemented to facilitate efficient and reliable communication between different system components. This design choice was driven by the need for component-specific communication, isolation of responsibilities, scalability, performance optimization, and fault tolerance. Each gRPC service was dedicated to a specific communication pathway: task submissions from Client to PLM, task assignment and updates between PLM and Scheduler, task execution commands from Scheduler to Worker, heartbeat signals and task status updates from Worker to Register, and worker status changes from Register to PLM. This isolation simplified the system architecture, making it easier to manage and debug. It also allowed independent scaling of components, enhancing system flexibility and efficiency.

Each service was optimized for its specific workload, such as high throughput for task submissions and low-latency updates for heartbeat signals, improving overall system performance. Additionally, using multiple services increased fault tolerance by isolating potential issues to individual services, preventing system-wide disruptions. These strategies collectively enhanced the modularity, scalability, performance, and robustness of our distributed task scheduler, effectively meeting the complex communication requirements of a distributed environment.

Shared Memory Locks for Data Consistency

To ensure data consistency in our distributed task scheduler, we employed shared memory locks, preventing simultaneous access by multiple processes. These locks were critical in maintaining the integrity of the task and worker heaps stored in shared memory. By using POSIX mutexes and semaphores, we synchronized access to shared memory segments, ensuring that only one process could read or write to a memory location at a time. This approach prevented race conditions and data corruption, which are common issues in multi-threaded environments.

Additionally, we optimized locking mechanisms to minimize contention and latency, using techniques like lock-free data structures and atomic operations for performance-critical sections. Memory barriers were also employed to prevent the CPU from reordering operations, ensuring that changes made by one process were immediately visible to others. These shared memory locks were integral to maintaining the system's reliability and performance, enabling safe and efficient concurrent access across multiple components in our distributed task scheduler.

References:

1. A. S. Tanenbaum and M. Van Steen, *Distributed Systems: Principles and Paradigms*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 2007.
2. The gRPC Authors, "gRPC: A High-Performance, Open-Source Universal RPC Framework," Google, [Online]. Available: https://grpc.io/. [Accessed: 17-May-2024].
3. I. Stevens, *UNIX Network Programming, Volume 2: Interprocess Communications*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1999.
4. M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. San Francisco, CA: No Starch Press, 2010.
5. Google Inc., "gRPC Core Concepts, Architecture and API," [Online]. Available: https://grpc.io/docs/guides/. [Accessed: 17-May-2024].
6. R. Love, Linux System Programming: Talking Directly to the Kernel and C Library, 2nd ed. Sebastopol, CA: O'Reilly Media, 2013.