# Ray Internals

A Comprehensive Technical Guide to Understanding Ray's Architecture, Implementation, and Distributed Computing Internals

## 📋 Table of Contents

### 📖 Part I: Ray Fundamentals

Introduction and Overview

Chapter 1: Ray Architecture Overview

Chapter 2: The Ray Driver System

Chapter 3: Task Lifecycle and Management

Chapter 4: Actor Lifecycle and Management

Chapter 5: Memory and Object Reference System

### 🏗️ Part II: Core Ray Services

Chapter 6: Global Control Service (GCS)

Chapter 7: Raylet Implementation and Lifecycle

Chapter 8: Distributed Object Store

### ⚡ Part III: Advanced Ray Systems

Chapter 9: Distributed Scheduling Implementation

Chapter 10: Autoscaling System

Chapter 11: High Availability and Fault Tolerance

### 🔧 Part IV: System Internals

Chapter 12: Network Communication and Protocols

Chapter 13: Port Assignment and Management

# Introduction and Overview

## The Complete Guide to Understanding Ray's Architecture, Implementation, and Internals

## 📖 Preface

Welcome to the most comprehensive technical documentation of Ray's internal architecture and implementation. This collection of guides has been crafted to provide deep insights into how Ray works under the hood, enabling developers, researchers, and engineers to understand, modify, and extend Ray's distributed computing capabilities.  
Ray is a powerful distributed computing framework, but its true potential can only be unlocked when you understand its internal mechanisms. This documentation bridges the gap between using Ray and truly mastering it by providing detailed explanations of its core systems, complete with code references, architectural diagrams, and practical insights.

## 👥 Intended Audience

This documentation is designed for:  
- 🔧 Ray Contributors: Developers who want to contribute to the Ray project  
- 🏗️ System Architects: Engineers designing distributed systems with Ray  
- 🎓 Researchers: Academic researchers studying distributed computing systems  
- 💼 Advanced Users: Power users who need to customize Ray for specific use cases  
- 🐛 Troubleshooters: Engineers debugging complex Ray deployment issues  
- 📚 Students: Computer science students learning distributed systems concepts

### Prerequisites

Strong understanding of distributed systems concepts

Proficiency in Python and C++

Familiarity with Ray's user-facing APIs

Basic knowledge of system programming and networking

## 📚 How This Book is Organized

This documentation is structured as a progressive journey through Ray's architecture, from fundamental concepts to advanced internals. Each chapter builds upon previous knowledge while remaining self-contained enough for reference use.  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

# 📋 Table of Contents

## 📖 Book Organization

### Part I: Ray Fundamentals ⚡

Understanding the core building blocks of the Ray ecosystem  
- Chapter 1: Ray Architecture Overview (3-4 hours)  
- System architecture and component interactions  
- Bootstrap process and initialization  
- Communication patterns and protocols  
- Chapter 2: The Ray Driver System (2-3 hours)  
- Driver lifecycle and initialization  
- Client-server communication mechanisms  
- Ray context management and session handling  
- Chapter 3: Task Lifecycle and Management (2-3 hours)  
- Task creation, submission, and execution  
- Dependency resolution and data handling  
- Performance optimization patterns  
- Chapter 4: Actor Lifecycle and Management (1-2 hours)  
- Actor creation and state management  
- Method invocation and result handling  
- Actor placement and resource allocation  
- Chapter 5: Memory and Object Reference System (1-2 hours)  
- Object storage and reference management  
- Memory optimization and garbage collection  
- Distributed object handling

### Part II: Core Ray Services 🏗️

Deep dive into the essential Ray system services  
- Chapter 6: Global Control Service (GCS) (2-3 hours)  
- Cluster metadata and coordination  
- Service discovery and health monitoring  
- Actor and placement group scheduling  
- Chapter 7: Raylet Implementation and Lifecycle (4-5 hours)  
- Node-level task scheduling and resource management  
- Worker process lifecycle management  
- Communication mechanisms and load handling  
- Chapter 8: Distributed Object Store (2-3 hours)  
- Plasma store integration and object management  
- Data transfer and locality optimization  
- Memory management and spilling strategies

### Part III: Advanced Ray Systems 🚀

Sophisticated scheduling and scaling mechanisms  
- Chapter 9: Distributed Scheduling Implementation (3-4 hours)  
- Multi-level scheduling architecture  
- Resource allocation algorithms  
- Placement strategies and locality optimization  
- Chapter 10: Autoscaling System (2-3 hours)  
- Demand-driven scaling algorithms  
- Node provisioning and resource management  
- Integration with cloud providers  
- Chapter 11: High Availability and Fault Tolerance (2-3 hours)  
- GCS fault tolerance and recovery mechanisms  
- Distributed system resilience patterns  
- Failure detection and handling strategies

### Part IV: System Internals 🔧

Low-level implementation details and networking  
- Chapter 12: Network Communication and Protocols (1-2 hours)  
- Custom protocol implementation  
- gRPC integration and message handling  
- Performance optimization techniques  
- Chapter 13: Port Assignment and Management (2-3 hours)  
- Dynamic port allocation strategies  
- Service discovery and networking  
- Cluster communication patterns

## 📚 Appendices

### Appendix A: Code Navigation Guide

How to navigate the Ray codebase effectively  
Key Directories:  
- src/ray/core\_worker/ - CoreWorker implementation  
- src/ray/raylet/ - Raylet implementation  
- src/ray/gcs/ - Global Control Service  
- src/ray/object\_store/ - Object store integration  
- python/ray/ - Python API implementation  
Important Files:  
- src/ray/raylet/main.cc - Raylet entry point  
- src/ray/core\_worker/core\_worker.cc - Core worker implementation  
- src/ray/gcs/gcs\_server/gcs\_server.cc - GCS server implementation

src/ray/core\_worker/

src/ray/raylet/

src/ray/gcs/

src/ray/object\_store/

python/ray/

src/ray/raylet/main.cc

src/ray/core\_worker/core\_worker.cc

src/ray/gcs/gcs\_server/gcs\_server.cc

### Appendix B: Troubleshooting Reference

Common issues and debugging techniques  
Debugging Tools:  
- ray status - Cluster state overview  
- ray logs - Component logs access  
- ray memory - Memory usage analysis  
- Ray Dashboard - Web-based monitoring  
Common Issues:  
- Task scheduling problems  
- Object store memory issues  
- Network connectivity problems  
- Actor lifecycle issues

ray status

ray logs

ray memory

### Appendix C: Performance Optimization

Best practices for optimal Ray performance  
Performance Guidelines:  
- Task granularity optimization  
- Memory management best practices  
- Resource specification guidelines  
- Network optimization techniques

### Appendix D: Additional Resources

Official Resources:  
- Ray Documentation  
- Ray GitHub Repository  
- Ray Community Forum  
Research Papers:  
- Ray: A Distributed Framework for Emerging AI Applications  
- Ray whitepaper and related publications  
Development Resources:  
- Ray contribution guidelines  
- Development environment setup  
- Testing frameworks and procedures

## 🚀 Getting Started

### For New Readers

Start with Part I if you're new to Ray internals

Read Chapter 1 for the big picture

Follow the learning path through each part sequentially

### For Specific Topics

Debugging Issues: Jump to relevant chapters + Appendix B

Performance Tuning: Focus on optimization sections + Appendix C

Contributing Code: Review relevant chapters + Appendix A

### For Reference Use

Use the detailed table of contents to find specific topics

Each chapter is designed to be self-contained

Cross-references guide you to related information

## 📖 Reading Recommendations

📚 Complete Reading Path (8-12 hours total)  
Follow Parts I → II → III → IV sequentially for comprehensive understanding  
🎯 Focused Learning Paths  
For Ray Contributors:  
- Chapter 2 (Driver) → Chapter 7 (Raylet) → Chapter 6 (GCS) → Appendix A  
For System Architects:  
- Chapter 1 (Overview) → Chapter 9 (Scheduling) → Chapter 10 (Autoscaling) → Chapter 11 (HA)  
For Performance Engineers:  
- Chapter 5 (Memory) → Chapter 8 (Object Store) → Chapter 9 (Scheduling) → Appendix C  
For Distributed Systems Students:  
- Chapter 1 (Overview) → Chapter 6 (GCS) → Chapter 7 (Raylet) → Chapter 11 (Fault Tolerance)

Happy Learning! 🎓  
Last Updated: December 2024  
Total Reading Time: 8-12 hours  
Difficulty Level: Advanced  
Prerequisites: Distributed systems knowledge, Python/C++ proficiency

# Chapter 1: Ray Architecture Overview

## Table of Contents

Introduction

Ray Cluster Architecture

Core Components Overview

Scheduling Architecture

Communication Patterns

Resource Management

Process Architecture

Component Interactions

System Bootstrap

Configuration System

Performance Characteristics

Fault Tolerance Overview

Development and Testing

Best Practices

## Introduction

Ray is a distributed computing framework designed for machine learning and AI workloads. This chapter provides a comprehensive overview of Ray's architecture, covering the fundamental components, their interactions, and the overall system design that enables scalable distributed computing.

### What is Ray?

Ray is an open-source unified framework for scaling AI workloads. It provides:  
- Distributed Computing: Scale Python workloads across multiple machines  
- Unified API: Single interface for tasks, actors, and data processing  
- Fault Tolerance: Built-in error handling and recovery mechanisms  
- Resource Management: Efficient allocation of CPU, GPU, and memory resources  
- Ecosystem: Libraries for ML (Ray Train), reinforcement learning (Ray RLlib), hyperparameter tuning (Ray Tune), and more

### Key Features

Multi-level Scheduling: Task-level, actor-level, and placement group scheduling

Resource-Aware: CPU, GPU, memory, and custom resource scheduling

Placement Strategies: PACK, SPREAD, STRICT\_PACK, STRICT\_SPREAD

Locality Optimization: Data locality-aware task placement

Dynamic Scaling: Integration with autoscaler for cluster growth/shrinkage

Label-Based Scheduling: Node affinity and label constraints

Performance Optimization: Efficient algorithms for large-scale clusters

### Scheduling Hierarchy

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

## Scheduling Architecture Overview

### Multi-Level Scheduling Architecture

Ray implements a hierarchical scheduling architecture with multiple decision points:

#### 1. Client-Side Scheduling

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships  
Location: src/ray/core\_worker/lease\_policy.cc  
The client-side scheduling makes initial placement decisions based on:  
- Data locality (object location)  
- Scheduling strategies (spread, node affinity)  
- Resource requirements

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

src/ray/core\_worker/lease\_policy.cc

#### 2. Raylet-Level Scheduling

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships  
Location: src/ray/raylet/scheduling/cluster\_task\_manager.cc

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

src/ray/raylet/scheduling/cluster\_task\_manager.cc

#### 3. GCS-Level Scheduling

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships  
Location: src/ray/gcs/gcs\_server/gcs\_actor\_scheduler.cc

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

src/ray/gcs/gcs\_server/gcs\_actor\_scheduler.cc

### Core Scheduling Flow

📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

## Core Scheduling Components

### ClusterResourceScheduler

Location: src/ray/raylet/scheduling/cluster\_resource\_scheduler.h  
The central coordinator for cluster-wide resource scheduling decisions.

src/ray/raylet/scheduling/cluster\_resource\_scheduler.h

class ClusterResourceScheduler {  
// Core scheduling method  
scheduling::NodeID GetBestSchedulableNode(  
const ResourceRequest &resource\_request,  
const rpc::SchedulingStrategy &scheduling\_strategy,  
bool actor\_creation,  
bool force\_spillback,  
const std::string &preferred\_node\_id,  
int64\_t \*total\_violations,  
bool \*is\_infeasible);  
// Bundle scheduling for placement groups  
SchedulingResult Schedule(  
const std::vector<const ResourceRequest \*> &resource\_request\_list,  
SchedulingOptions options);  
}

class ClusterResourceScheduler {  
// Core scheduling method  
scheduling::NodeID GetBestSchedulableNode(  
const ResourceRequest &resource\_request,  
const rpc::SchedulingStrategy &scheduling\_strategy,  
bool actor\_creation,  
bool force\_spillback,  
const std::string &preferred\_node\_id,  
int64\_t \*total\_violations,  
bool \*is\_infeasible);  
// Bundle scheduling for placement groups  
SchedulingResult Schedule(  
const std::vector<const ResourceRequest \*> &resource\_request\_list,  
SchedulingOptions options);  
}

Key Responsibilities:  
- Node feasibility checking  
- Resource availability tracking  
- Scheduling strategy implementation  
- Placement group bundle scheduling

### ClusterTaskManager

Location: src/ray/raylet/scheduling/cluster\_task\_manager.h  
Manages task queuing and scheduling at the cluster level.

src/ray/raylet/scheduling/cluster\_task\_manager.h

class ClusterTaskManager {  
void QueueAndScheduleTask(  
RayTask task,  
bool grant\_or\_reject,  
bool is\_selected\_based\_on\_locality,  
rpc::RequestWorkerLeaseReply \*reply,  
rpc::SendReplyCallback send\_reply\_callback);  
void ScheduleAndDispatchTasks();  
}

class ClusterTaskManager {  
void QueueAndScheduleTask(  
RayTask task,  
bool grant\_or\_reject,  
bool is\_selected\_based\_on\_locality,  
rpc::RequestWorkerLeaseReply \*reply,  
rpc::SendReplyCallback send\_reply\_callback);  
void ScheduleAndDispatchTasks();  
}

Scheduling Queues:  
- tasks\_to\_schedule\_: Tasks waiting for resources  
- infeasible\_tasks\_: Tasks that cannot be scheduled

tasks\_to\_schedule\_

infeasible\_tasks\_

### LocalTaskManager

Location: src/ray/raylet/local\_task\_manager.h  
Handles local task execution and worker management.

src/ray/raylet/local\_task\_manager.h

class LocalTaskManager {  
void QueueAndScheduleTask(std::shared\_ptr<internal::Work> work);  
void ScheduleAndDispatchTasks();  
bool TrySpillback(const std::shared\_ptr<internal::Work> &work,  
bool &is\_infeasible);  
}

class LocalTaskManager {  
void QueueAndScheduleTask(std::shared\_ptr<internal::Work> work);  
void ScheduleAndDispatchTasks();  
bool TrySpillback(const std::shared\_ptr<internal::Work> &work,  
bool &is\_infeasible);  
}

Fairness Policy: Implements CPU-fair scheduling to prevent resource starvation:

// From src/ray/raylet/local\_task\_manager.cc  
if (total\_cpu\_requests\_ > total\_cpus) {  
RAY\_LOG(DEBUG) << "Applying fairness policy. Total CPU requests ("  
<< total\_cpu\_requests\_ << ") exceed total CPUs ("  
<< total\_cpus << ")";  
// Apply fair dispatching logic  
}

// From src/ray/raylet/local\_task\_manager.cc  
if (total\_cpu\_requests\_ > total\_cpus) {  
RAY\_LOG(DEBUG) << "Applying fairness policy. Total CPU requests ("  
<< total\_cpu\_requests\_ << ") exceed total CPUs ("  
<< total\_cpus << ")";  
// Apply fair dispatching logic  
}

### Scheduling Policies

Location: src/ray/raylet/scheduling/policy/  
Ray implements multiple scheduling policies:

src/ray/raylet/scheduling/policy/

#### HybridSchedulingPolicy

Default scheduling strategy

Balances locality and load distribution

Configurable spread threshold

#### SpreadSchedulingPolicy

Distributes tasks across nodes

Minimizes resource contention

Used for embarrassingly parallel workloads

#### NodeAffinitySchedulingPolicy

Hard/soft node constraints

Supports spillback on unavailability

Critical for stateful workloads

#### NodeLabelSchedulingPolicy

class NodeLabelSchedulingPolicy : public ISchedulingPolicy {  
scheduling::NodeID Schedule(const ResourceRequest &resource\_request,  
SchedulingOptions options) override;  
private:  
bool IsNodeMatchLabelExpression(const Node &node,  
const rpc::LabelMatchExpression &expression);  
};

class NodeLabelSchedulingPolicy : public ISchedulingPolicy {  
scheduling::NodeID Schedule(const ResourceRequest &resource\_request,  
SchedulingOptions options) override;  
private:  
bool IsNodeMatchLabelExpression(const Node &node,  
const rpc::LabelMatchExpression &expression);  
};

### Scheduling Context and Options

Location: src/ray/raylet/scheduling/policy/scheduling\_options.h

src/ray/raylet/scheduling/policy/scheduling\_options.h

struct SchedulingOptions {  
SchedulingType scheduling\_type;  
float spread\_threshold;  
bool avoid\_local\_node;  
bool require\_node\_available;  
bool avoid\_gpu\_nodes;  
double max\_cpu\_fraction\_per\_node; // For placement groups  
static SchedulingOptions Hybrid(bool avoid\_local\_node,  
bool require\_node\_available,  
const std::string &preferred\_node\_id);  
static SchedulingOptions BundlePack(double max\_cpu\_fraction\_per\_node = 1.0);  
static SchedulingOptions BundleStrictSpread(double max\_cpu\_fraction\_per\_node = 1.0);  
};

struct SchedulingOptions {  
SchedulingType scheduling\_type;  
float spread\_threshold;  
bool avoid\_local\_node;  
bool require\_node\_available;  
bool avoid\_gpu\_nodes;  
double max\_cpu\_fraction\_per\_node; // For placement groups  
static SchedulingOptions Hybrid(bool avoid\_local\_node,  
bool require\_node\_available,  
const std::string &preferred\_node\_id);  
static SchedulingOptions BundlePack(double max\_cpu\_fraction\_per\_node = 1.0);  
static SchedulingOptions BundleStrictSpread(double max\_cpu\_fraction\_per\_node = 1.0);  
};

## Resource Management and Allocation

### Resource Model

Ray uses a multi-dimensional resource model:

// Resource types from src/ray/common/scheduling/scheduling\_ids.h  
enum PredefinedResources {  
CPU = 0,  
MEM = 1,  
GPU = 2,  
OBJECT\_STORE\_MEM = 3,  
// Custom resources start from 4  
};

// Resource types from src/ray/common/scheduling/scheduling\_ids.h  
enum PredefinedResources {  
CPU = 0,  
MEM = 1,  
GPU = 2,  
OBJECT\_STORE\_MEM = 3,  
// Custom resources start from 4  
};

### Resource Request Structure

class ResourceRequest {  
ResourceSet resource\_set\_; // Required resources  
LabelSelector label\_selector\_; // Node label requirements  
bool requires\_object\_store\_memory\_; // Memory constraint flag  
bool IsEmpty() const;  
const ResourceSet &GetResourceSet() const;  
bool RequiresObjectStoreMemory() const;  
};

class ResourceRequest {  
ResourceSet resource\_set\_; // Required resources  
LabelSelector label\_selector\_; // Node label requirements  
bool requires\_object\_store\_memory\_; // Memory constraint flag  
bool IsEmpty() const;  
const ResourceSet &GetResourceSet() const;  
bool RequiresObjectStoreMemory() const;  
};

### NodeResources

Location: src/ray/common/scheduling/cluster\_resource\_data.h

src/ray/common/scheduling/cluster\_resource\_data.h

struct NodeResources {  
NodeResourceSet total; // Total node capacity  
NodeResourceSet available; // Currently available  
NodeResourceSet normal\_task\_resources; // Reserved for tasks  
absl::flat\_hash\_map<std::string, std::string> labels; // Node labels  
bool object\_pulls\_queued; // Object store status  
bool IsAvailable(const ResourceRequest &resource\_request) const;  
bool IsFeasible(const ResourceRequest &resource\_request) const;  
bool HasRequiredLabels(const LabelSelector &label\_selector) const;  
float CalculateCriticalResourceUtilization() const;  
};

struct NodeResources {  
NodeResourceSet total; // Total node capacity  
NodeResourceSet available; // Currently available  
NodeResourceSet normal\_task\_resources; // Reserved for tasks  
absl::flat\_hash\_map<std::string, std::string> labels; // Node labels  
bool object\_pulls\_queued; // Object store status  
bool IsAvailable(const ResourceRequest &resource\_request) const;  
bool IsFeasible(const ResourceRequest &resource\_request) const;  
bool HasRequiredLabels(const LabelSelector &label\_selector) const;  
float CalculateCriticalResourceUtilization() const;  
};

### Resource Allocation Algorithm

bool ClusterResourceScheduler::IsSchedulable(  
const ResourceRequest &resource\_request,  
scheduling::NodeID node\_id) const {  
return cluster\_resource\_manager\_->HasAvailableResources(  
node\_id,  
resource\_request,  
/\*ignore\_object\_store\_memory\_requirement\*/  
node\_id == local\_node\_id\_) &&  
NodeAvailable(node\_id);  
}

bool ClusterResourceScheduler::IsSchedulable(  
const ResourceRequest &resource\_request,  
scheduling::NodeID node\_id) const {  
return cluster\_resource\_manager\_->HasAvailableResources(  
node\_id,  
resource\_request,  
/\*ignore\_object\_store\_memory\_requirement\*/  
node\_id == local\_node\_id\_) &&  
NodeAvailable(node\_id);  
}

### Dynamic Resource Management

// From src/ray/raylet/scheduling/cluster\_resource\_scheduler\_test.cc  
TEST\_F(ClusterResourceSchedulerTest, DynamicResourceTest) {  
// Add dynamic resources at runtime  
resource\_scheduler.GetLocalResourceManager().AddLocalResourceInstances(  
scheduling::ResourceID("custom123"), {0., 1.0, 1.0});  
// Verify schedulability  
auto result = resource\_scheduler.GetBestSchedulableNode(resource\_request, ...);  
ASSERT\_FALSE(result.IsNil());  
}

// From src/ray/raylet/scheduling/cluster\_resource\_scheduler\_test.cc  
TEST\_F(ClusterResourceSchedulerTest, DynamicResourceTest) {  
// Add dynamic resources at runtime  
resource\_scheduler.GetLocalResourceManager().AddLocalResourceInstances(  
scheduling::ResourceID("custom123"), {0., 1.0, 1.0});  
// Verify schedulability  
auto result = resource\_scheduler.GetBestSchedulableNode(resource\_request, ...);  
ASSERT\_FALSE(result.IsNil());  
}

### Resource Binpacking

Ray implements sophisticated binpacking for resource allocation:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

## Task Scheduling Algorithms

### Hybrid Scheduling Algorithm

Default Strategy: Balances locality and load distribution

// Configuration from src/ray/raylet/scheduling/cluster\_resource\_scheduler.cc  
best\_node\_id = scheduling\_policy\_->Schedule(  
resource\_request,  
SchedulingOptions::Hybrid(  
/\*avoid\_local\_node\*/ force\_spillback,  
/\*require\_node\_available\*/ force\_spillback,  
preferred\_node\_id));

// Configuration from src/ray/raylet/scheduling/cluster\_resource\_scheduler.cc  
best\_node\_id = scheduling\_policy\_->Schedule(  
resource\_request,  
SchedulingOptions::Hybrid(  
/\*avoid\_local\_node\*/ force\_spillback,  
/\*require\_node\_available\*/ force\_spillback,  
preferred\_node\_id));

Algorithm Steps:  
1. Score Calculation: Based on resource utilization  
2. Top-K Selection: Choose from best k nodes (default: 20% of cluster)  
3. Random Selection: Within top-k for load balancing  
Scoring Function:

float NodeResources::CalculateCriticalResourceUtilization() const {  
float highest = 0;  
for (const auto &i : {CPU, MEM, OBJECT\_STORE\_MEM}) {  
float utilization = 1 - (available / total);  
if (utilization > highest) {  
highest = utilization;  
}  
}  
return highest;  
}

float NodeResources::CalculateCriticalResourceUtilization() const {  
float highest = 0;  
for (const auto &i : {CPU, MEM, OBJECT\_STORE\_MEM}) {  
float utilization = 1 - (available / total);  
if (utilization > highest) {  
highest = utilization;  
}  
}  
return highest;  
}

### Spread Scheduling Algorithm

Purpose: Distribute tasks across maximum number of nodes

// From scheduling policy tests  
TEST\_F(SchedulingPolicyTest, SpreadSchedulingStrategyTest) {  
rpc::SchedulingStrategy scheduling\_strategy;  
scheduling\_strategy.mutable\_spread\_scheduling\_strategy();  
auto node\_id = resource\_scheduler.GetBestSchedulableNode(  
resource\_request, LabelSelector(), scheduling\_strategy, ...);  
}

// From scheduling policy tests  
TEST\_F(SchedulingPolicyTest, SpreadSchedulingStrategyTest) {  
rpc::SchedulingStrategy scheduling\_strategy;  
scheduling\_strategy.mutable\_spread\_scheduling\_strategy();  
auto node\_id = resource\_scheduler.GetBestSchedulableNode(  
resource\_request, LabelSelector(), scheduling\_strategy, ...);  
}

Implementation:  
- Prioritizes nodes with lowest task count  
- Avoids resource hotspots  
- Maximizes fault tolerance

### Node Affinity Scheduling

Hard Affinity: Must run on specific node

if (IsHardNodeAffinitySchedulingStrategy(scheduling\_strategy)) {  
// Must schedule on specified node or fail  
best\_node\_id = scheduling\_policy\_->Schedule(  
resource\_request,  
SchedulingOptions::NodeAffinity(  
force\_spillback, force\_spillback,  
scheduling\_strategy.node\_affinity\_scheduling\_strategy().node\_id(),  
/\*soft=\*/false, /\*spill\_on\_unavailable=\*/false,  
/\*fail\_on\_unavailable=\*/true));  
}

if (IsHardNodeAffinitySchedulingStrategy(scheduling\_strategy)) {  
// Must schedule on specified node or fail  
best\_node\_id = scheduling\_policy\_->Schedule(  
resource\_request,  
SchedulingOptions::NodeAffinity(  
force\_spillback, force\_spillback,  
scheduling\_strategy.node\_affinity\_scheduling\_strategy().node\_id(),  
/\*soft=\*/false, /\*spill\_on\_unavailable=\*/false,  
/\*fail\_on\_unavailable=\*/true));  
}

Soft Affinity: Prefer specific node but allow spillback

scheduling\_strategy.mutable\_node\_affinity\_scheduling\_strategy()->set\_soft(true);  
// Will try preferred node first, then other nodes

scheduling\_strategy.mutable\_node\_affinity\_scheduling\_strategy()->set\_soft(true);  
// Will try preferred node first, then other nodes

### Fair Scheduling

CPU Fair Scheduling: Prevents starvation across scheduling classes

// From src/ray/raylet/local\_task\_manager.cc  
if (total\_cpu\_requests\_ > total\_cpus) {  
// Calculate fair share per scheduling class  
double fair\_share = total\_cpus / num\_classes\_with\_cpu;  
// Apply throttling based on fair share  
for (auto &[scheduling\_class, dispatch\_queue] : tasks\_to\_dispatch\_) {  
double cpu\_request = /\* CPU required by this class \*/;  
if (cpu\_request > fair\_share) {  
// Throttle this class  
next\_update\_time = current\_time + throttle\_delay;  
}  
}  
}

// From src/ray/raylet/local\_task\_manager.cc  
if (total\_cpu\_requests\_ > total\_cpus) {  
// Calculate fair share per scheduling class  
double fair\_share = total\_cpus / num\_classes\_with\_cpu;  
// Apply throttling based on fair share  
for (auto &[scheduling\_class, dispatch\_queue] : tasks\_to\_dispatch\_) {  
double cpu\_request = /\* CPU required by this class \*/;  
if (cpu\_request > fair\_share) {  
// Throttle this class  
next\_update\_time = current\_time + throttle\_delay;  
}  
}  
}

## Actor Placement and Scheduling

### Actor Scheduling Architecture

Location: src/ray/gcs/gcs\_server/gcs\_actor\_scheduler.cc  
Ray provides two actor scheduling modes:

src/ray/gcs/gcs\_server/gcs\_actor\_scheduler.cc

#### 1. GCS-Based Actor Scheduling

void GcsActorScheduler::ScheduleByGcs(std::shared\_ptr<GcsActor> actor) {  
// Create task for actor creation  
auto task = std::make\_shared<RayTask>(actor->GetCreationTaskSpecification());  
// Use cluster task manager for scheduling  
cluster\_task\_manager\_.QueueAndScheduleTask(  
std::move(task),  
/\*grant\_or\_reject\*/ false,  
/\*is\_selected\_based\_on\_locality\*/ false,  
reply.get(),  
send\_reply\_callback);  
}

void GcsActorScheduler::ScheduleByGcs(std::shared\_ptr<GcsActor> actor) {  
// Create task for actor creation  
auto task = std::make\_shared<RayTask>(actor->GetCreationTaskSpecification());  
// Use cluster task manager for scheduling  
cluster\_task\_manager\_.QueueAndScheduleTask(  
std::move(task),  
/\*grant\_or\_reject\*/ false,  
/\*is\_selected\_based\_on\_locality\*/ false,  
reply.get(),  
send\_reply\_callback);  
}

#### 2. Raylet-Based Actor Scheduling

void GcsActorScheduler::ScheduleByRaylet(std::shared\_ptr<GcsActor> actor) {  
// Select forwarding node  
auto node\_id = SelectForwardingNode(actor);  
// Lease worker directly from node  
LeaseWorkerFromNode(actor, node.value());  
}

void GcsActorScheduler::ScheduleByRaylet(std::shared\_ptr<GcsActor> actor) {  
// Select forwarding node  
auto node\_id = SelectForwardingNode(actor);  
// Lease worker directly from node  
LeaseWorkerFromNode(actor, node.value());  
}

### Actor Resource Requirements

Placement vs Execution Resources:

// From src/ray/common/task/task\_spec.cc  
const auto &resource\_set =  
(is\_actor\_creation\_task && should\_report\_placement\_resources)  
? GetRequiredPlacementResources() // For scheduling decisions  
: GetRequiredResources(); // For execution

// From src/ray/common/task/task\_spec.cc  
const auto &resource\_set =  
(is\_actor\_creation\_task && should\_report\_placement\_resources)  
? GetRequiredPlacementResources() // For scheduling decisions  
: GetRequiredResources(); // For execution

Actor Creation Example:

@ray.remote(num\_cpus=2, num\_gpus=1, memory=1000)  
class MyActor:  
def \_\_init\_\_(self):  
pass  
def method(self):  
pass  
# Actor placement considers both creation and method resources  
actor = MyActor.remote()

@ray.remote(num\_cpus=2, num\_gpus=1, memory=1000)  
class MyActor:  
def \_\_init\_\_(self):  
pass  
def method(self):  
pass  
# Actor placement considers both creation and method resources  
actor = MyActor.remote()

### Actor Lifecycle and Scheduling

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Actor Scheduling Considerations

Resource Lifetime: Actors hold resources for their entire lifetime

if (task\_spec.IsActorCreationTask()) {  
// The actor belongs to this worker now  
worker->SetLifetimeAllocatedInstances(allocated\_instances);  
} else {  
worker->SetAllocatedInstances(allocated\_instances);  
}

if (task\_spec.IsActorCreationTask()) {  
// The actor belongs to this worker now  
worker->SetLifetimeAllocatedInstances(allocated\_instances);  
} else {  
worker->SetAllocatedInstances(allocated\_instances);  
}

Scheduling Class: Actors use placement resources for scheduling decisions

TEST(TaskSpecTest, TestActorSchedulingClass) {  
// Actor's scheduling class determined by placement resources  
TaskSpecification actor\_task(actor\_task\_spec\_proto);  
TaskSpecification regular\_task(regular\_task\_spec\_proto);  
ASSERT\_EQ(regular\_task.GetSchedulingClass(), actor\_task.GetSchedulingClass());  
}

TEST(TaskSpecTest, TestActorSchedulingClass) {  
// Actor's scheduling class determined by placement resources  
TaskSpecification actor\_task(actor\_task\_spec\_proto);  
TaskSpecification regular\_task(regular\_task\_spec\_proto);  
ASSERT\_EQ(regular\_task.GetSchedulingClass(), actor\_task.GetSchedulingClass());  
}

## Placement Group Scheduling

### Placement Group Architecture

Location: src/ray/gcs/gcs\_server/gcs\_placement\_group\_scheduler.cc  
Placement groups enable gang scheduling of related resources across multiple nodes.

src/ray/gcs/gcs\_server/gcs\_placement\_group\_scheduler.cc

class GcsPlacementGroupScheduler {  
void SchedulePlacementGroup(  
std::shared\_ptr<GcsPlacementGroup> placement\_group,  
PGSchedulingFailureCallback failure\_callback,  
PGSchedulingSuccessfulCallback success\_callback);  
}

class GcsPlacementGroupScheduler {  
void SchedulePlacementGroup(  
std::shared\_ptr<GcsPlacementGroup> placement\_group,  
PGSchedulingFailureCallback failure\_callback,  
PGSchedulingSuccessfulCallback success\_callback);  
}

### Bundle Specification

Location: src/ray/common/bundle\_spec.h

src/ray/common/bundle\_spec.h

class BundleSpecification {  
BundleID BundleId() const;  
PlacementGroupID PlacementGroupId() const;  
NodeID NodeId() const;  
int64\_t Index() const;  
const ResourceRequest &GetRequiredResources() const;  
const absl::flat\_hash\_map<std::string, double> &GetFormattedResources() const;  
};

class BundleSpecification {  
BundleID BundleId() const;  
PlacementGroupID PlacementGroupId() const;  
NodeID NodeId() const;  
int64\_t Index() const;  
const ResourceRequest &GetRequiredResources() const;  
const absl::flat\_hash\_map<std::string, double> &GetFormattedResources() const;  
};

### Placement Strategies

#### PACK Strategy

case rpc::PlacementStrategy::PACK:  
return SchedulingOptions::BundlePack(max\_cpu\_fraction\_per\_node);

case rpc::PlacementStrategy::PACK:  
return SchedulingOptions::BundlePack(max\_cpu\_fraction\_per\_node);

Goal: Minimize number of nodes used

Use Case: Maximize locality, minimize network overhead

Algorithm: First-fit decreasing binpacking

#### SPREAD Strategy

case rpc::PlacementStrategy::SPREAD:  
return SchedulingOptions::BundleSpread(max\_cpu\_fraction\_per\_node);

case rpc::PlacementStrategy::SPREAD:  
return SchedulingOptions::BundleSpread(max\_cpu\_fraction\_per\_node);

Goal: Distribute bundles across nodes

Use Case: Fault tolerance, load distribution

Algorithm: Round-robin placement with load balancing

#### STRICT\_PACK Strategy

case rpc::PlacementStrategy::STRICT\_PACK:  
return SchedulingOptions::BundleStrictPack(  
max\_cpu\_fraction\_per\_node,  
soft\_target\_node\_id);

case rpc::PlacementStrategy::STRICT\_PACK:  
return SchedulingOptions::BundleStrictPack(  
max\_cpu\_fraction\_per\_node,  
soft\_target\_node\_id);

Goal: All bundles on single node (if possible)

Use Case: Shared memory, minimal latency

Algorithm: Single-node placement with fallback

#### STRICT\_SPREAD Strategy

case rpc::PlacementStrategy::STRICT\_SPREAD:  
return SchedulingOptions::BundleStrictSpread(  
max\_cpu\_fraction\_per\_node,  
CreateSchedulingContext(placement\_group\_id));

case rpc::PlacementStrategy::STRICT\_SPREAD:  
return SchedulingOptions::BundleStrictSpread(  
max\_cpu\_fraction\_per\_node,  
CreateSchedulingContext(placement\_group\_id));

Goal: Each bundle on different node

Use Case: Maximum fault tolerance

Algorithm: One bundle per node constraint

### Bundle Scheduling Algorithm

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Bundle Resource Formatting

Ray formats placement group resources with special naming:

// From src/ray/common/bundle\_spec.h  
std::string FormatPlacementGroupResource(  
const std::string &original\_resource\_name,  
const std::string &group\_id\_str,  
int64\_t bundle\_index) {  
if (bundle\_index == -1) {  
// Wildcard resource: CPU\_group\_<group\_id>  
return original\_resource\_name + "\_group\_" + group\_id\_str;  
} else {  
// Indexed resource: CPU\_group\_<bundle\_index>\_<group\_id>  
return original\_resource\_name + "\_group\_" +  
std::to\_string(bundle\_index) + "\_" + group\_id\_str;  
}  
}

// From src/ray/common/bundle\_spec.h  
std::string FormatPlacementGroupResource(  
const std::string &original\_resource\_name,  
const std::string &group\_id\_str,  
int64\_t bundle\_index) {  
if (bundle\_index == -1) {  
// Wildcard resource: CPU\_group\_<group\_id>  
return original\_resource\_name + "\_group\_" + group\_id\_str;  
} else {  
// Indexed resource: CPU\_group\_<bundle\_index>\_<group\_id>  
return original\_resource\_name + "\_group\_" +  
std::to\_string(bundle\_index) + "\_" + group\_id\_str;  
}  
}

### CPU Fraction Limits

Purpose: Prevent placement groups from monopolizing nodes

bool AllocationWillExceedMaxCpuFraction(  
const NodeResources &node\_resources,  
const ResourceRequest &bundle\_resource\_request,  
double max\_cpu\_fraction\_per\_node,  
double available\_cpus\_before\_current\_pg\_request) {  
if (max\_cpu\_fraction\_per\_node == 1.0) {  
return false; // No limit  
}  
auto max\_reservable\_cpus =  
max\_cpu\_fraction\_per\_node \* node\_resources.total.Get(cpu\_id).Double();  
// Ensure at least 1 CPU is excluded from placement groups  
if (max\_reservable\_cpus > total\_cpus - 1) {  
max\_reservable\_cpus = total\_cpus - 1;  
}  
return cpus\_used\_by\_pg\_after > max\_reservable\_cpus;  
}

bool AllocationWillExceedMaxCpuFraction(  
const NodeResources &node\_resources,  
const ResourceRequest &bundle\_resource\_request,  
double max\_cpu\_fraction\_per\_node,  
double available\_cpus\_before\_current\_pg\_request) {  
if (max\_cpu\_fraction\_per\_node == 1.0) {  
return false; // No limit  
}  
auto max\_reservable\_cpus =  
max\_cpu\_fraction\_per\_node \* node\_resources.total.Get(cpu\_id).Double();  
// Ensure at least 1 CPU is excluded from placement groups  
if (max\_reservable\_cpus > total\_cpus - 1) {  
max\_reservable\_cpus = total\_cpus - 1;  
}  
return cpus\_used\_by\_pg\_after > max\_reservable\_cpus;  
}

### Placement Group Lifecycle

📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

## Scheduling Strategies

### Strategy Types and Implementation

Ray supports multiple scheduling strategies through the rpc::SchedulingStrategy protocol buffer:

rpc::SchedulingStrategy

// From src/ray/raylet/scheduling/cluster\_resource\_scheduler.cc  
scheduling::NodeID ClusterResourceScheduler::GetBestSchedulableNode(  
const ResourceRequest &resource\_request,  
const rpc::SchedulingStrategy &scheduling\_strategy,  
bool actor\_creation,  
bool force\_spillback,  
const std::string &preferred\_node\_id,  
int64\_t \*total\_violations,  
bool \*is\_infeasible) {  
if (scheduling\_strategy.scheduling\_strategy\_case() ==  
rpc::SchedulingStrategy::SchedulingStrategyCase::kSpreadSchedulingStrategy) {  
best\_node\_id = scheduling\_policy\_->Schedule(  
resource\_request,  
SchedulingOptions::Spread(force\_spillback, force\_spillback));  
} else if (scheduling\_strategy.scheduling\_strategy\_case() ==  
rpc::SchedulingStrategy::SchedulingStrategyCase::  
kNodeAffinitySchedulingStrategy) {  
best\_node\_id = scheduling\_policy\_->Schedule(  
resource\_request,  
SchedulingOptions::NodeAffinity(/\* ... \*/));  
} else if (scheduling\_strategy.has\_node\_label\_scheduling\_strategy()) {  
best\_node\_id = scheduling\_policy\_->Schedule(  
resource\_request,  
SchedulingOptions::NodeLabelScheduling(scheduling\_strategy));  
}  
}

// From src/ray/raylet/scheduling/cluster\_resource\_scheduler.cc  
scheduling::NodeID ClusterResourceScheduler::GetBestSchedulableNode(  
const ResourceRequest &resource\_request,  
const rpc::SchedulingStrategy &scheduling\_strategy,  
bool actor\_creation,  
bool force\_spillback,  
const std::string &preferred\_node\_id,  
int64\_t \*total\_violations,  
bool \*is\_infeasible) {  
if (scheduling\_strategy.scheduling\_strategy\_case() ==  
rpc::SchedulingStrategy::SchedulingStrategyCase::kSpreadSchedulingStrategy) {  
best\_node\_id = scheduling\_policy\_->Schedule(  
resource\_request,  
SchedulingOptions::Spread(force\_spillback, force\_spillback));  
} else if (scheduling\_strategy.scheduling\_strategy\_case() ==  
rpc::SchedulingStrategy::SchedulingStrategyCase::  
kNodeAffinitySchedulingStrategy) {  
best\_node\_id = scheduling\_policy\_->Schedule(  
resource\_request,  
SchedulingOptions::NodeAffinity(/\* ... \*/));  
} else if (scheduling\_strategy.has\_node\_label\_scheduling\_strategy()) {  
best\_node\_id = scheduling\_policy\_->Schedule(  
resource\_request,  
SchedulingOptions::NodeLabelScheduling(scheduling\_strategy));  
}  
}

### DEFAULT Strategy

Implementation: Hybrid policy with configurable parameters

# Environment variables controlling DEFAULT strategy  
RAY\_scheduler\_spread\_threshold = 0.5 # Utilization threshold  
RAY\_scheduler\_top\_k\_fraction = 0.2 # Top-k selection ratio  
RAY\_scheduler\_top\_k\_absolute = 5 # Minimum top-k count

# Environment variables controlling DEFAULT strategy  
RAY\_scheduler\_spread\_threshold = 0.5 # Utilization threshold  
RAY\_scheduler\_top\_k\_fraction = 0.2 # Top-k selection ratio  
RAY\_scheduler\_top\_k\_absolute = 5 # Minimum top-k count

Algorithm:  
1. Calculate node scores based on resource utilization  
2. Select top-k nodes with lowest scores  
3. Randomly choose from top-k for load balancing

### SPREAD Strategy

Purpose: Maximize distribution across nodes

import ray  
@ray.remote(scheduling\_strategy="SPREAD")  
def distributed\_task():  
return "Running on different nodes"  
# Tasks will be distributed across available nodes  
futures = [distributed\_task.remote() for \_ in range(100)]

import ray  
@ray.remote(scheduling\_strategy="SPREAD")  
def distributed\_task():  
return "Running on different nodes"  
# Tasks will be distributed across available nodes  
futures = [distributed\_task.remote() for \_ in range(100)]

Implementation Details:  
- Prioritizes nodes with fewer running tasks  
- Considers resource utilization as secondary factor  
- Useful for embarrassingly parallel workloads

### Node Affinity Strategy

Hard Affinity: Must run on specific node

import ray  
from ray.util.scheduling\_strategies import NodeAffinitySchedulingStrategy  
@ray.remote(scheduling\_strategy=NodeAffinitySchedulingStrategy(  
node\_id="specific-node-id",  
soft=False  
))  
def pinned\_task():  
return "Must run on specific node"

import ray  
from ray.util.scheduling\_strategies import NodeAffinitySchedulingStrategy  
@ray.remote(scheduling\_strategy=NodeAffinitySchedulingStrategy(  
node\_id="specific-node-id",  
soft=False  
))  
def pinned\_task():  
return "Must run on specific node"

Soft Affinity: Prefer specific node with fallback

@ray.remote(scheduling\_strategy=NodeAffinitySchedulingStrategy(  
node\_id="preferred-node-id",  
soft=True  
))  
def preferred\_task():  
return "Prefers specific node but can run elsewhere"

@ray.remote(scheduling\_strategy=NodeAffinitySchedulingStrategy(  
node\_id="preferred-node-id",  
soft=True  
))  
def preferred\_task():  
return "Prefers specific node but can run elsewhere"

### Placement Group Strategy

Bundle-Specific Scheduling:

import ray  
from ray.util.placement\_group import placement\_group  
from ray.util.scheduling\_strategies import PlacementGroupSchedulingStrategy  
# Create placement group  
pg = placement\_group([{"CPU": 2}, {"CPU": 2}], strategy="PACK")  
@ray.remote(scheduling\_strategy=PlacementGroupSchedulingStrategy(  
placement\_group=pg,  
placement\_group\_bundle\_index=0  
))  
def task\_on\_bundle\_0():  
return "Running on bundle 0"  
@ray.remote(scheduling\_strategy=PlacementGroupSchedulingStrategy(  
placement\_group=pg,  
placement\_group\_bundle\_index=-1 # Any bundle  
))  
def task\_on\_any\_bundle():  
return "Running on any available bundle"

import ray  
from ray.util.placement\_group import placement\_group  
from ray.util.scheduling\_strategies import PlacementGroupSchedulingStrategy  
# Create placement group  
pg = placement\_group([{"CPU": 2}, {"CPU": 2}], strategy="PACK")  
@ray.remote(scheduling\_strategy=PlacementGroupSchedulingStrategy(  
placement\_group=pg,  
placement\_group\_bundle\_index=0  
))  
def task\_on\_bundle\_0():  
return "Running on bundle 0"  
@ray.remote(scheduling\_strategy=PlacementGroupSchedulingStrategy(  
placement\_group=pg,  
placement\_group\_bundle\_index=-1 # Any bundle  
))  
def task\_on\_any\_bundle():  
return "Running on any available bundle"

## Node Affinity and Label-Based Scheduling

### Node Label Scheduling Policy

Location: src/ray/raylet/scheduling/policy/node\_label\_scheduling\_policy.cc  
Ray supports sophisticated label-based scheduling for fine-grained node selection:

src/ray/raylet/scheduling/policy/node\_label\_scheduling\_policy.cc

scheduling::NodeID NodeLabelSchedulingPolicy::Schedule(  
const ResourceRequest &resource\_request,  
SchedulingOptions options) {  
// 1. Select feasible nodes  
auto hard\_match\_nodes = SelectFeasibleNodes(resource\_request);  
// 2. Filter by hard expressions  
if (node\_label\_scheduling\_strategy.hard().expressions().size() > 0) {  
hard\_match\_nodes = FilterNodesByLabelMatchExpressions(  
hard\_match\_nodes, node\_label\_scheduling\_strategy.hard());  
}  
// 3. Filter by soft expressions  
auto hard\_and\_soft\_match\_nodes = FilterNodesByLabelMatchExpressions(  
hard\_match\_nodes, node\_label\_scheduling\_strategy.soft());  
return SelectBestNode(hard\_match\_nodes, hard\_and\_soft\_match\_nodes, resource\_request);  
}

scheduling::NodeID NodeLabelSchedulingPolicy::Schedule(  
const ResourceRequest &resource\_request,  
SchedulingOptions options) {  
// 1. Select feasible nodes  
auto hard\_match\_nodes = SelectFeasibleNodes(resource\_request);  
// 2. Filter by hard expressions  
if (node\_label\_scheduling\_strategy.hard().expressions().size() > 0) {  
hard\_match\_nodes = FilterNodesByLabelMatchExpressions(  
hard\_match\_nodes, node\_label\_scheduling\_strategy.hard());  
}  
// 3. Filter by soft expressions  
auto hard\_and\_soft\_match\_nodes = FilterNodesByLabelMatchExpressions(  
hard\_match\_nodes, node\_label\_scheduling\_strategy.soft());  
return SelectBestNode(hard\_match\_nodes, hard\_and\_soft\_match\_nodes, resource\_request);  
}

### Label Matching Implementation

bool NodeLabelSchedulingPolicy::IsNodeMatchLabelExpression(  
const Node &node, const rpc::LabelMatchExpression &expression) const {  
const auto &key = expression.key();  
const auto &operator\_type = expression.operator\_();  
const auto &values = expression.values();  
switch (operator\_type) {  
case rpc::LabelMatchExpression::IN:  
return IsNodeLabelInValues(node, key, values);  
case rpc::LabelMatchExpression::NOT\_IN:  
return !IsNodeLabelInValues(node, key, values);  
case rpc::LabelMatchExpression::EXISTS:  
return IsNodeLabelKeyExists(node, key);  
case rpc::LabelMatchExpression::DOES\_NOT\_EXIST:  
return !IsNodeLabelKeyExists(node, key);  
}  
}

bool NodeLabelSchedulingPolicy::IsNodeMatchLabelExpression(  
const Node &node, const rpc::LabelMatchExpression &expression) const {  
const auto &key = expression.key();  
const auto &operator\_type = expression.operator\_();  
const auto &values = expression.values();  
switch (operator\_type) {  
case rpc::LabelMatchExpression::IN:  
return IsNodeLabelInValues(node, key, values);  
case rpc::LabelMatchExpression::NOT\_IN:  
return !IsNodeLabelInValues(node, key, values);  
case rpc::LabelMatchExpression::EXISTS:  
return IsNodeLabelKeyExists(node, key);  
case rpc::LabelMatchExpression::DOES\_NOT\_EXIST:  
return !IsNodeLabelKeyExists(node, key);  
}  
}

### Label Selector Usage

import ray  
from ray.util.scheduling\_strategies import NodeLabelSchedulingStrategy  
# Hard constraints (must match)  
hard\_constraints = {  
"ray.io/node-type": "gpu-node",  
"zone": "us-west-1a"  
}  
# Soft constraints (preferred)  
soft\_constraints = {  
"instance-type": "p3.2xlarge"  
}  
@ray.remote(scheduling\_strategy=NodeLabelSchedulingStrategy(  
hard=hard\_constraints,  
soft=soft\_constraints  
))  
def gpu\_task():  
return "Running on GPU node in preferred zone"

import ray  
from ray.util.scheduling\_strategies import NodeLabelSchedulingStrategy  
# Hard constraints (must match)  
hard\_constraints = {  
"ray.io/node-type": "gpu-node",  
"zone": "us-west-1a"  
}  
# Soft constraints (preferred)  
soft\_constraints = {  
"instance-type": "p3.2xlarge"  
}  
@ray.remote(scheduling\_strategy=NodeLabelSchedulingStrategy(  
hard=hard\_constraints,  
soft=soft\_constraints  
))  
def gpu\_task():  
return "Running on GPU node in preferred zone"

### Node Label Management

Static Labels: Set during node startup

# Set node labels via environment  
export RAY\_NODE\_LABELS='{"zone":"us-west-1a","instance-type":"m5.large"}'  
ray start --head

# Set node labels via environment  
export RAY\_NODE\_LABELS='{"zone":"us-west-1a","instance-type":"m5.large"}'  
ray start --head

Dynamic Labels: Updated at runtime

// From cluster resource data  
struct NodeResources {  
absl::flat\_hash\_map<std::string, std::string> labels;  
bool HasRequiredLabels(const LabelSelector &label\_selector) const;  
bool NodeLabelMatchesConstraint(const LabelConstraint &constraint) const;  
};

// From cluster resource data  
struct NodeResources {  
absl::flat\_hash\_map<std::string, std::string> labels;  
bool HasRequiredLabels(const LabelSelector &label\_selector) const;  
bool NodeLabelMatchesConstraint(const LabelConstraint &constraint) const;  
};

## Locality-Aware Scheduling

### Locality-Aware Lease Policy

Location: src/ray/core\_worker/lease\_policy.cc  
Ray implements data locality-aware scheduling to minimize data movement:

src/ray/core\_worker/lease\_policy.cc

std::pair<rpc::Address, bool> LocalityAwareLeasePolicy::GetBestNodeForTask(  
const TaskSpecification &spec) {  
// Check for explicit scheduling strategies first  
if (spec.IsSpreadSchedulingStrategy() || spec.IsNodeAffinitySchedulingStrategy()) {  
return std::make\_pair(fallback\_rpc\_address\_, false);  
}  
// Pick node based on locality  
if (auto node\_id = GetBestNodeIdForTask(spec)) {  
if (auto addr = node\_addr\_factory\_(node\_id.value())) {  
return std::make\_pair(addr.value(), true);  
}  
}  
return std::make\_pair(fallback\_rpc\_address\_, false);  
}

std::pair<rpc::Address, bool> LocalityAwareLeasePolicy::GetBestNodeForTask(  
const TaskSpecification &spec) {  
// Check for explicit scheduling strategies first  
if (spec.IsSpreadSchedulingStrategy() || spec.IsNodeAffinitySchedulingStrategy()) {  
return std::make\_pair(fallback\_rpc\_address\_, false);  
}  
// Pick node based on locality  
if (auto node\_id = GetBestNodeIdForTask(spec)) {  
if (auto addr = node\_addr\_factory\_(node\_id.value())) {  
return std::make\_pair(addr.value(), true);  
}  
}  
return std::make\_pair(fallback\_rpc\_address\_, false);  
}

### Locality Calculation

Criteria: Node with most object bytes local

std::optional<NodeID> LocalityAwareLeasePolicy::GetBestNodeIdForTask(  
const TaskSpecification &spec) {  
const auto &dependencies = spec.GetDependencies();  
if (dependencies.empty()) {  
return std::nullopt;  
}  
// Calculate locality scores for each node  
absl::flat\_hash\_map<NodeID, int64\_t> locality\_scores;  
for (const auto &obj\_id : dependencies) {  
auto locality\_data = locality\_data\_provider\_.GetLocalityData(obj\_id);  
for (const auto &node\_id : locality\_data.nodes\_containing\_object) {  
locality\_scores[node\_id] += locality\_data.object\_size;  
}  
}  
// Return node with highest locality score  
return GetNodeWithMaxScore(locality\_scores);  
}

std::optional<NodeID> LocalityAwareLeasePolicy::GetBestNodeIdForTask(  
const TaskSpecification &spec) {  
const auto &dependencies = spec.GetDependencies();  
if (dependencies.empty()) {  
return std::nullopt;  
}  
// Calculate locality scores for each node  
absl::flat\_hash\_map<NodeID, int64\_t> locality\_scores;  
for (const auto &obj\_id : dependencies) {  
auto locality\_data = locality\_data\_provider\_.GetLocalityData(obj\_id);  
for (const auto &node\_id : locality\_data.nodes\_containing\_object) {  
locality\_scores[node\_id] += locality\_data.object\_size;  
}  
}  
// Return node with highest locality score  
return GetNodeWithMaxScore(locality\_scores);  
}

### Locality vs Strategy Priority

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Locality Testing

// From src/ray/tests/test\_scheduling.py  
def test\_locality\_aware\_leasing(ray\_start\_cluster):  
@ray.remote(resources={"pin": 1})  
def non\_local():  
return ray.\_private.worker.global\_worker.node.unique\_id  
@ray.remote  
def f(x):  
return ray.\_private.worker.global\_worker.node.unique\_id  
# Test that task f() runs on the same node as non\_local()  
# due to data locality  
assert ray.get(f.remote(non\_local.remote())) == non\_local\_node.unique\_id

// From src/ray/tests/test\_scheduling.py  
def test\_locality\_aware\_leasing(ray\_start\_cluster):  
@ray.remote(resources={"pin": 1})  
def non\_local():  
return ray.\_private.worker.global\_worker.node.unique\_id  
@ray.remote  
def f(x):  
return ray.\_private.worker.global\_worker.node.unique\_id  
# Test that task f() runs on the same node as non\_local()  
# due to data locality  
assert ray.get(f.remote(non\_local.remote())) == non\_local\_node.unique\_id

## Cluster Resource Scheduling

### Cluster Resource Manager

Location: src/ray/raylet/scheduling/cluster\_resource\_manager.h  
Maintains global view of cluster resources:

src/ray/raylet/scheduling/cluster\_resource\_manager.h

class ClusterResourceManager {  
// Add or update node resources  
void AddOrUpdateNode(scheduling::NodeID node\_id,  
const NodeResources &node\_resources);  
// Check resource availability  
bool HasAvailableResources(scheduling::NodeID node\_id,  
const ResourceRequest &resource\_request) const;  
// Resource allocation  
bool SubtractNodeAvailableResources(scheduling::NodeID node\_id,  
const ResourceRequest &resource\_request);  
};

class ClusterResourceManager {  
// Add or update node resources  
void AddOrUpdateNode(scheduling::NodeID node\_id,  
const NodeResources &node\_resources);  
// Check resource availability  
bool HasAvailableResources(scheduling::NodeID node\_id,  
const ResourceRequest &resource\_request) const;  
// Resource allocation  
bool SubtractNodeAvailableResources(scheduling::NodeID node\_id,  
const ResourceRequest &resource\_request);  
};

### Resource Synchronization

📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

### Resource Reporting

Location: src/ray/raylet/scheduling/scheduler\_resource\_reporter.cc

src/ray/raylet/scheduling/scheduler\_resource\_reporter.cc

void SchedulerResourceReporter::FillResourceUsage(rpc::ResourcesData &data) const {  
// Report resource demands by shape  
auto resource\_load\_by\_shape = data.mutable\_resource\_load\_by\_shape();  
for (const auto &[scheduling\_class, task\_queue] : tasks\_to\_schedule\_) {  
const auto &resources = scheduling\_class\_descriptor.resource\_set.GetResourceMap();  
auto by\_shape\_entry = resource\_load\_by\_shape->Add();  
for (const auto &resource : resources) {  
(\*by\_shape\_entry->mutable\_shape())[resource.first] = resource.second;  
}  
by\_shape\_entry->set\_num\_ready\_requests\_queued(task\_queue.size());  
}  
}

void SchedulerResourceReporter::FillResourceUsage(rpc::ResourcesData &data) const {  
// Report resource demands by shape  
auto resource\_load\_by\_shape = data.mutable\_resource\_load\_by\_shape();  
for (const auto &[scheduling\_class, task\_queue] : tasks\_to\_schedule\_) {  
const auto &resources = scheduling\_class\_descriptor.resource\_set.GetResourceMap();  
auto by\_shape\_entry = resource\_load\_by\_shape->Add();  
for (const auto &resource : resources) {  
(\*by\_shape\_entry->mutable\_shape())[resource.first] = resource.second;  
}  
by\_shape\_entry->set\_num\_ready\_requests\_queued(task\_queue.size());  
}  
}

## Autoscaler Integration

### Resource Demand Scheduler

Location: python/ray/autoscaler/v2/scheduler.py  
The autoscaler uses sophisticated scheduling algorithms to determine cluster scaling decisions:

python/ray/autoscaler/v2/scheduler.py

class ResourceDemandScheduler(IResourceScheduler):  
def schedule(self, request: SchedulingRequest) -> SchedulingReply:  
ctx = self.ScheduleContext.from\_schedule\_request(request)  
# 1. Enforce min workers per type  
self.\_enforce\_min\_workers\_per\_type(ctx)  
# 2. Enforce resource constraints  
infeasible\_constraints = self.\_enforce\_resource\_constraints(  
ctx, request.cluster\_resource\_constraints)  
# 3. Schedule gang resource requests  
infeasible\_gang\_requests = self.\_sched\_gang\_resource\_requests(  
ctx, request.gang\_resource\_requests)  
# 4. Schedule regular resource requests  
infeasible\_requests = self.\_sched\_resource\_requests(  
ctx, ResourceRequestUtil.ungroup\_by\_count(request.resource\_requests))  
# 5. Enforce idle termination  
self.\_enforce\_idle\_termination(ctx)  
return SchedulingReply(  
to\_launch=ctx.get\_launch\_requests(),  
to\_terminate=ctx.get\_terminate\_requests(),  
infeasible\_resource\_requests=infeasible\_requests,  
infeasible\_gang\_resource\_requests=infeasible\_gang\_requests,  
infeasible\_cluster\_resource\_constraints=infeasible\_constraints  
)

class ResourceDemandScheduler(IResourceScheduler):  
def schedule(self, request: SchedulingRequest) -> SchedulingReply:  
ctx = self.ScheduleContext.from\_schedule\_request(request)  
# 1. Enforce min workers per type  
self.\_enforce\_min\_workers\_per\_type(ctx)  
# 2. Enforce resource constraints  
infeasible\_constraints = self.\_enforce\_resource\_constraints(  
ctx, request.cluster\_resource\_constraints)  
# 3. Schedule gang resource requests  
infeasible\_gang\_requests = self.\_sched\_gang\_resource\_requests(  
ctx, request.gang\_resource\_requests)  
# 4. Schedule regular resource requests  
infeasible\_requests = self.\_sched\_resource\_requests(  
ctx, ResourceRequestUtil.ungroup\_by\_count(request.resource\_requests))  
# 5. Enforce idle termination  
self.\_enforce\_idle\_termination(ctx)  
return SchedulingReply(  
to\_launch=ctx.get\_launch\_requests(),  
to\_terminate=ctx.get\_terminate\_requests(),  
infeasible\_resource\_requests=infeasible\_requests,  
infeasible\_gang\_resource\_requests=infeasible\_gang\_requests,  
infeasible\_cluster\_resource\_constraints=infeasible\_constraints  
)

### Binpacking Algorithm

def \_try\_schedule(  
ctx: ScheduleContext,  
requests\_to\_sched: List[ResourceRequest],  
resource\_request\_source: ResourceRequestSource,  
) -> Tuple[List[SchedulingNode], List[ResourceRequest]]:  
# Sort requests by complexity for better binpacking  
def \_sort\_resource\_request(req: ResourceRequest) -> Tuple:  
return (  
len(req.placement\_constraints),  
len(req.resources\_bundle.values()),  
sum(req.resources\_bundle.values()),  
sorted(req.resources\_bundle.items()),  
)  
requests\_to\_sched = sorted(  
requests\_to\_sched, key=\_sort\_resource\_request, reverse=True)  
# Try scheduling on existing nodes first  
while len(requests\_to\_sched) > 0 and len(existing\_nodes) > 0:  
best\_node, requests\_to\_sched, existing\_nodes = \  
self.\_sched\_best\_node(requests\_to\_sched, existing\_nodes, resource\_request\_source)  
if best\_node is None:  
break  
target\_nodes.append(best\_node)  
# Try scheduling on new nodes  
for node\_type, num\_available in node\_type\_available.items():  
if num\_available > 0:  
new\_node = SchedulingNode.from\_node\_config(  
ctx.get\_node\_type\_configs()[node\_type],  
status=SchedulingNodeStatus.TO\_LAUNCH)  
# Try to schedule remaining requests on new node

def \_try\_schedule(  
ctx: ScheduleContext,  
requests\_to\_sched: List[ResourceRequest],  
resource\_request\_source: ResourceRequestSource,  
) -> Tuple[List[SchedulingNode], List[ResourceRequest]]:  
# Sort requests by complexity for better binpacking  
def \_sort\_resource\_request(req: ResourceRequest) -> Tuple:  
return (  
len(req.placement\_constraints),  
len(req.resources\_bundle.values()),  
sum(req.resources\_bundle.values()),  
sorted(req.resources\_bundle.items()),  
)  
requests\_to\_sched = sorted(  
requests\_to\_sched, key=\_sort\_resource\_request, reverse=True)  
# Try scheduling on existing nodes first  
while len(requests\_to\_sched) > 0 and len(existing\_nodes) > 0:  
best\_node, requests\_to\_sched, existing\_nodes = \  
self.\_sched\_best\_node(requests\_to\_sched, existing\_nodes, resource\_request\_source)  
if best\_node is None:  
break  
target\_nodes.append(best\_node)  
# Try scheduling on new nodes  
for node\_type, num\_available in node\_type\_available.items():  
if num\_available > 0:  
new\_node = SchedulingNode.from\_node\_config(  
ctx.get\_node\_type\_configs()[node\_type],  
status=SchedulingNodeStatus.TO\_LAUNCH)  
# Try to schedule remaining requests on new node

### Placement Group Autoscaling

def placement\_groups\_to\_resource\_demands(  
pending\_placement\_groups: List[PlacementGroupTableData],  
) -> Tuple[List[ResourceDict], List[List[ResourceDict]]]:  
resource\_demand\_vector = []  
unconverted = []  
for placement\_group in pending\_placement\_groups:  
shapes = [dict(bundle.unit\_resources) for bundle in placement\_group.bundles  
if bundle.node\_id == b""] # Only unplaced bundles  
if placement\_group.strategy == PlacementStrategy.PACK:  
resource\_demand\_vector.extend(shapes)  
elif placement\_group.strategy == PlacementStrategy.STRICT\_PACK:  
# Combine all bundles into single demand  
combined = collections.defaultdict(float)  
for shape in shapes:  
for label, quantity in shape.items():  
combined[label] += quantity  
resource\_demand\_vector.append(combined)  
elif placement\_group.strategy == PlacementStrategy.STRICT\_SPREAD:  
# Cannot be converted - needs special handling  
unconverted.append(shapes)  
return resource\_demand\_vector, unconverted

def placement\_groups\_to\_resource\_demands(  
pending\_placement\_groups: List[PlacementGroupTableData],  
) -> Tuple[List[ResourceDict], List[List[ResourceDict]]]:  
resource\_demand\_vector = []  
unconverted = []  
for placement\_group in pending\_placement\_groups:  
shapes = [dict(bundle.unit\_resources) for bundle in placement\_group.bundles  
if bundle.node\_id == b""] # Only unplaced bundles  
if placement\_group.strategy == PlacementStrategy.PACK:  
resource\_demand\_vector.extend(shapes)  
elif placement\_group.strategy == PlacementStrategy.STRICT\_PACK:  
# Combine all bundles into single demand  
combined = collections.defaultdict(float)  
for shape in shapes:  
for label, quantity in shape.items():  
combined[label] += quantity  
resource\_demand\_vector.append(combined)  
elif placement\_group.strategy == PlacementStrategy.STRICT\_SPREAD:  
# Cannot be converted - needs special handling  
unconverted.append(shapes)  
return resource\_demand\_vector, unconverted

### Autoscaler Configuration

# Example autoscaler configuration  
cluster\_name: ray-cluster  
max\_workers: 100  
upscaling\_speed: 1.0  
idle\_timeout\_minutes: 5  
available\_node\_types:  
ray.head.default:  
min\_workers: 0  
max\_workers: 0  
resources: {"CPU": 4}  
ray.worker.cpu:  
min\_workers: 0  
max\_workers: 50  
resources: {"CPU": 8, "memory": 32000000000}  
ray.worker.gpu:  
min\_workers: 0  
max\_workers: 10  
resources: {"CPU": 16, "GPU": 4, "memory": 64000000000}

# Example autoscaler configuration  
cluster\_name: ray-cluster  
max\_workers: 100  
upscaling\_speed: 1.0  
idle\_timeout\_minutes: 5  
available\_node\_types:  
ray.head.default:  
min\_workers: 0  
max\_workers: 0  
resources: {"CPU": 4}  
ray.worker.cpu:  
min\_workers: 0  
max\_workers: 50  
resources: {"CPU": 8, "memory": 32000000000}  
ray.worker.gpu:  
min\_workers: 0  
max\_workers: 10  
resources: {"CPU": 16, "GPU": 4, "memory": 64000000000}

## Performance Characteristics

### Scheduling Latency

Typical Latencies:  
- Local scheduling: 1-5ms  
- Remote scheduling: 10-50ms  
- Placement group creation: 100-1000ms  
- Autoscaler response: 30-300s

### Scalability Metrics

Cluster Size: Ray scheduling tested up to 1000+ nodes  
Task Throughput:  
- Simple tasks: 100K+ tasks/second  
- Complex scheduling: 10K+ tasks/second  
- Placement groups: 100+ groups/second

### Memory Usage

Scheduler Memory Overhead:

// Per-node overhead in ClusterResourceManager  
struct NodeResources {  
NodeResourceSet total; // ~1KB per node  
NodeResourceSet available; // ~1KB per node  
NodeResourceSet normal\_task\_resources; // ~1KB per node  
absl::flat\_hash\_map<std::string, std::string> labels; // Variable  
};  
// Total: ~3KB + labels per node

// Per-node overhead in ClusterResourceManager  
struct NodeResources {  
NodeResourceSet total; // ~1KB per node  
NodeResourceSet available; // ~1KB per node  
NodeResourceSet normal\_task\_resources; // ~1KB per node  
absl::flat\_hash\_map<std::string, std::string> labels; // Variable  
};  
// Total: ~3KB + labels per node

Task Queue Memory:

// Per-task overhead in scheduling queues  
class Work {  
RayTask task; // ~2KB per task  
TaskResourceInstances allocated; // ~500B per task  
WorkStatus state; // ~100B per task  
};  
// Total: ~2.6KB per queued task

// Per-task overhead in scheduling queues  
class Work {  
RayTask task; // ~2KB per task  
TaskResourceInstances allocated; // ~500B per task  
WorkStatus state; // ~100B per task  
};  
// Total: ~2.6KB per queued task

### Performance Optimization

Top-K Selection: Reduces scheduling complexity from O(N) to O(K)

// Default configuration  
RAY\_scheduler\_top\_k\_fraction = 0.2 // 20% of nodes  
RAY\_scheduler\_top\_k\_absolute = 5 // Minimum 5 nodes

// Default configuration  
RAY\_scheduler\_top\_k\_fraction = 0.2 // 20% of nodes  
RAY\_scheduler\_top\_k\_absolute = 5 // Minimum 5 nodes

Caching: Resource views cached to avoid repeated calculations

class ClusterResourceManager {  
// Cached resource calculations  
mutable absl::flat\_hash\_map<scheduling::NodeID, float> utilization\_cache\_;  
mutable int64\_t cache\_timestamp\_;  
};

class ClusterResourceManager {  
// Cached resource calculations  
mutable absl::flat\_hash\_map<scheduling::NodeID, float> utilization\_cache\_;  
mutable int64\_t cache\_timestamp\_;  
};

## Configuration and Tuning

### Environment Variables

Core Scheduling:

# Spread threshold for hybrid scheduling  
export RAY\_scheduler\_spread\_threshold=0.5  
# Top-k node selection  
export RAY\_scheduler\_top\_k\_fraction=0.2  
export RAY\_scheduler\_top\_k\_absolute=5  
# Worker management  
export RAY\_num\_workers\_soft\_limit=1000  
export RAY\_maximum\_startup\_concurrency=10

# Spread threshold for hybrid scheduling  
export RAY\_scheduler\_spread\_threshold=0.5  
# Top-k node selection  
export RAY\_scheduler\_top\_k\_fraction=0.2  
export RAY\_scheduler\_top\_k\_absolute=5  
# Worker management  
export RAY\_num\_workers\_soft\_limit=1000  
export RAY\_maximum\_startup\_concurrency=10

Resource Management:

# Object store memory scheduling  
export RAY\_object\_store\_memory=1000000000  
# Pull manager configuration  
export RAY\_object\_manager\_pull\_timeout\_ms=10000  
export RAY\_object\_manager\_max\_bytes\_in\_flight=100000000

# Object store memory scheduling  
export RAY\_object\_store\_memory=1000000000  
# Pull manager configuration  
export RAY\_object\_manager\_pull\_timeout\_ms=10000  
export RAY\_object\_manager\_max\_bytes\_in\_flight=100000000

Placement Groups:

# CPU fraction limits  
export RAY\_placement\_group\_max\_cpu\_fraction\_per\_node=0.8  
# Bundle scheduling timeout  
export RAY\_placement\_group\_bundle\_resource\_timeout\_s=30

# CPU fraction limits  
export RAY\_placement\_group\_max\_cpu\_fraction\_per\_node=0.8  
# Bundle scheduling timeout  
export RAY\_placement\_group\_bundle\_resource\_timeout\_s=30

### Runtime Configuration

Cluster Resource Constraints:

import ray  
# Set cluster-wide resource constraints  
ray.autoscaler.sdk.request\_resources([  
{"CPU": 100, "GPU": 10}, # Ensure cluster can handle this workload  
{"memory": 1000000000} # Minimum memory requirement  
])

import ray  
# Set cluster-wide resource constraints  
ray.autoscaler.sdk.request\_resources([  
{"CPU": 100, "GPU": 10}, # Ensure cluster can handle this workload  
{"memory": 1000000000} # Minimum memory requirement  
])

Node Type Configuration:

# Configure node types for autoscaling  
node\_config = {  
"ray.worker.cpu": {  
"min\_workers": 2,  
"max\_workers": 20,  
"resources": {"CPU": 8, "memory": 32000000000}  
},  
"ray.worker.gpu": {  
"min\_workers": 0,  
"max\_workers": 5,  
"resources": {"CPU": 16, "GPU": 4, "memory": 64000000000}  
}  
}

# Configure node types for autoscaling  
node\_config = {  
"ray.worker.cpu": {  
"min\_workers": 2,  
"max\_workers": 20,  
"resources": {"CPU": 8, "memory": 32000000000}  
},  
"ray.worker.gpu": {  
"min\_workers": 0,  
"max\_workers": 5,  
"resources": {"CPU": 16, "GPU": 4, "memory": 64000000000}  
}  
}

### Performance Tuning

For High Throughput:

# Increase worker limits  
export RAY\_num\_workers\_soft\_limit=2000  
export RAY\_maximum\_startup\_concurrency=50  
# Reduce scheduling overhead  
export RAY\_scheduler\_top\_k\_absolute=10  
export RAY\_scheduler\_spread\_threshold=0.3

# Increase worker limits  
export RAY\_num\_workers\_soft\_limit=2000  
export RAY\_maximum\_startup\_concurrency=50  
# Reduce scheduling overhead  
export RAY\_scheduler\_top\_k\_absolute=10  
export RAY\_scheduler\_spread\_threshold=0.3

For Low Latency:

# Prioritize local scheduling  
export RAY\_scheduler\_spread\_threshold=0.8  
export RAY\_scheduler\_top\_k\_fraction=0.1  
# Reduce worker startup time  
export RAY\_worker\_lease\_timeout\_milliseconds=1000

# Prioritize local scheduling  
export RAY\_scheduler\_spread\_threshold=0.8  
export RAY\_scheduler\_top\_k\_fraction=0.1  
# Reduce worker startup time  
export RAY\_worker\_lease\_timeout\_milliseconds=1000

For Large Clusters:

# Optimize for scale  
export RAY\_scheduler\_top\_k\_fraction=0.1 # Top 10% of nodes  
export RAY\_raylet\_report\_resources\_period\_milliseconds=1000  
export RAY\_gcs\_resource\_report\_poll\_period\_milliseconds=1000

# Optimize for scale  
export RAY\_scheduler\_top\_k\_fraction=0.1 # Top 10% of nodes  
export RAY\_raylet\_report\_resources\_period\_milliseconds=1000  
export RAY\_gcs\_resource\_report\_poll\_period\_milliseconds=1000

## Best Practices

### Task Scheduling

1. Use Appropriate Scheduling Strategies:

# For embarrassingly parallel workloads  
@ray.remote(scheduling\_strategy="SPREAD")  
def parallel\_task(data):  
return process(data)  
# For data-dependent tasks (default locality-aware)  
@ray.remote  
def dependent\_task(large\_object):  
return analyze(large\_object)  
# For specific hardware requirements  
@ray.remote(scheduling\_strategy=NodeAffinitySchedulingStrategy(  
node\_id=gpu\_node\_id, soft=True))  
def gpu\_task():  
return train\_model()

# For embarrassingly parallel workloads  
@ray.remote(scheduling\_strategy="SPREAD")  
def parallel\_task(data):  
return process(data)  
# For data-dependent tasks (default locality-aware)  
@ray.remote  
def dependent\_task(large\_object):  
return analyze(large\_object)  
# For specific hardware requirements  
@ray.remote(scheduling\_strategy=NodeAffinitySchedulingStrategy(  
node\_id=gpu\_node\_id, soft=True))  
def gpu\_task():  
return train\_model()

2. Resource Specification:

# Be specific about resource requirements  
@ray.remote(num\_cpus=2, num\_gpus=1, memory=4000\*1024\*1024)  
def resource\_intensive\_task():  
return compute()  
# Use custom resources for specialized hardware  
@ray.remote(resources={"accelerator": 1})  
def accelerated\_task():  
return specialized\_compute()

# Be specific about resource requirements  
@ray.remote(num\_cpus=2, num\_gpus=1, memory=4000\*1024\*1024)  
def resource\_intensive\_task():  
return compute()  
# Use custom resources for specialized hardware  
@ray.remote(resources={"accelerator": 1})  
def accelerated\_task():  
return specialized\_compute()

### Actor Placement

1. Consider Resource Lifetime:

# Actors hold resources for their lifetime  
@ray.remote(num\_cpus=4, num\_gpus=1)  
class ModelServer:  
def \_\_init\_\_(self):  
self.model = load\_large\_model()  
def predict(self, data):  
return self.model.predict(data)  
# Create fewer, long-lived actors rather than many short-lived ones  
server = ModelServer.remote()

# Actors hold resources for their lifetime  
@ray.remote(num\_cpus=4, num\_gpus=1)  
class ModelServer:  
def \_\_init\_\_(self):  
self.model = load\_large\_model()  
def predict(self, data):  
return self.model.predict(data)  
# Create fewer, long-lived actors rather than many short-lived ones  
server = ModelServer.remote()

2. Use Placement Groups for Related Actors:

# Group related actors together  
pg = placement\_group([{"CPU": 4}, {"CPU": 4}, {"CPU": 4}], strategy="PACK")  
actors = [  
Actor.options(scheduling\_strategy=PlacementGroupSchedulingStrategy(  
placement\_group=pg, placement\_group\_bundle\_index=i  
)).remote() for i in range(3)  
]

# Group related actors together  
pg = placement\_group([{"CPU": 4}, {"CPU": 4}, {"CPU": 4}], strategy="PACK")  
actors = [  
Actor.options(scheduling\_strategy=PlacementGroupSchedulingStrategy(  
placement\_group=pg, placement\_group\_bundle\_index=i  
)).remote() for i in range(3)  
]

### Placement Group Design

1. Choose Appropriate Strategies:

# For tightly coupled workloads  
pg\_pack = placement\_group([{"CPU": 2, "GPU": 1}] \* 4, strategy="PACK")  
# For fault tolerance  
pg\_spread = placement\_group([{"CPU": 2}] \* 8, strategy="SPREAD")  
# For strict requirements  
pg\_strict = placement\_group([{"CPU": 4}] \* 2, strategy="STRICT\_SPREAD")

# For tightly coupled workloads  
pg\_pack = placement\_group([{"CPU": 2, "GPU": 1}] \* 4, strategy="PACK")  
# For fault tolerance  
pg\_spread = placement\_group([{"CPU": 2}] \* 8, strategy="SPREAD")  
# For strict requirements  
pg\_strict = placement\_group([{"CPU": 4}] \* 2, strategy="STRICT\_SPREAD")

2. Bundle Size Optimization:

# Avoid bundles larger than single node capacity  
# Bad: Bundle requires more than any node has  
bad\_pg = placement\_group([{"CPU": 64, "GPU": 8}]) # If max node has 32 CPU  
# Good: Bundle fits on available nodes  
good\_pg = placement\_group([{"CPU": 16, "GPU": 2}] \* 4)

# Avoid bundles larger than single node capacity  
# Bad: Bundle requires more than any node has  
bad\_pg = placement\_group([{"CPU": 64, "GPU": 8}]) # If max node has 32 CPU  
# Good: Bundle fits on available nodes  
good\_pg = placement\_group([{"CPU": 16, "GPU": 2}] \* 4)

### Autoscaler Optimization

1. Configure Appropriate Limits:

# Set realistic min/max workers  
available\_node\_types:  
ray.worker.default:  
min\_workers: 2 # Always keep some capacity  
max\_workers: 100 # Prevent runaway scaling  
upscaling\_speed: 2.0 # Scale up aggressively

# Set realistic min/max workers  
available\_node\_types:  
ray.worker.default:  
min\_workers: 2 # Always keep some capacity  
max\_workers: 100 # Prevent runaway scaling  
upscaling\_speed: 2.0 # Scale up aggressively

2. Use Resource Constraints:

# Ensure cluster can handle expected workload  
ray.autoscaler.sdk.request\_resources([  
{"CPU": 200, "memory": 500000000000}, # Expected peak usage  
])

# Ensure cluster can handle expected workload  
ray.autoscaler.sdk.request\_resources([  
{"CPU": 200, "memory": 500000000000}, # Expected peak usage  
])

## Troubleshooting

### Common Scheduling Issues

1. Tasks Stuck in Pending State:  
Symptoms: Tasks remain in PENDING\_SCHEDULING state  
Causes:  
- Insufficient cluster resources  
- Infeasible resource requirements  
- Node affinity to unavailable nodes  
Debugging:

# Check cluster resources  
print(ray.cluster\_resources())  
print(ray.available\_resources())  
# Check task resource requirements  
@ray.remote(num\_cpus=1)  
def debug\_task():  
return ray.get\_runtime\_context().get\_assigned\_resources()  
# Check for infeasible tasks  
ray.autoscaler.sdk.request\_resources([{"CPU": 1000}]) # Will show if infeasible

# Check cluster resources  
print(ray.cluster\_resources())  
print(ray.available\_resources())  
# Check task resource requirements  
@ray.remote(num\_cpus=1)  
def debug\_task():  
return ray.get\_runtime\_context().get\_assigned\_resources()  
# Check for infeasible tasks  
ray.autoscaler.sdk.request\_resources([{"CPU": 1000}]) # Will show if infeasible

2. Poor Load Balancing:  
Symptoms: Some nodes overloaded while others idle  
Causes:  
- Inappropriate scheduling strategy  
- Data locality overriding load balancing  
- Sticky worker assignment  
Solutions:

# Use SPREAD strategy for better distribution  
@ray.remote(scheduling\_strategy="SPREAD")  
def distributed\_task():  
return compute()  
# Adjust spread threshold  
import os  
os.environ["RAY\_scheduler\_spread\_threshold"] = "0.3"

# Use SPREAD strategy for better distribution  
@ray.remote(scheduling\_strategy="SPREAD")  
def distributed\_task():  
return compute()  
# Adjust spread threshold  
import os  
os.environ["RAY\_scheduler\_spread\_threshold"] = "0.3"

3. Placement Group Creation Failures:  
Symptoms: Placement groups fail to create or timeout  
Causes:  
- Insufficient cluster capacity  
- Conflicting resource constraints  
- Network partitions  
Debugging:

import ray  
from ray.util.placement\_group import placement\_group  
# Check placement group status  
pg = placement\_group([{"CPU": 2}] \* 4, strategy="STRICT\_SPREAD")  
print(pg.ready()) # False if creation failed  
# Check bundle placement  
print(ray.util.placement\_group\_table())

import ray  
from ray.util.placement\_group import placement\_group  
# Check placement group status  
pg = placement\_group([{"CPU": 2}] \* 4, strategy="STRICT\_SPREAD")  
print(pg.ready()) # False if creation failed  
# Check bundle placement  
print(ray.util.placement\_group\_table())

### Performance Issues

1. High Scheduling Latency:  
Symptoms: Long delays between task submission and execution  
Causes:  
- Large cluster with inefficient node selection  
- Complex placement constraints  
- Resource fragmentation  
Solutions:

# Reduce top-k selection size  
export RAY\_scheduler\_top\_k\_fraction=0.1  
# Increase spread threshold for faster local scheduling  
export RAY\_scheduler\_spread\_threshold=0.7

# Reduce top-k selection size  
export RAY\_scheduler\_top\_k\_fraction=0.1  
# Increase spread threshold for faster local scheduling  
export RAY\_scheduler\_spread\_threshold=0.7

2. Memory Issues in Scheduler:  
Symptoms: Raylet OOM, high memory usage in scheduling components  
Causes:  
- Large number of queued tasks  
- Memory leaks in scheduling data structures  
- Excessive resource tracking overhead  
Solutions:

# Limit concurrent tasks  
export RAY\_num\_workers\_soft\_limit=500  
# Reduce resource reporting frequency  
export RAY\_raylet\_report\_resources\_period\_milliseconds=5000

# Limit concurrent tasks  
export RAY\_num\_workers\_soft\_limit=500  
# Reduce resource reporting frequency  
export RAY\_raylet\_report\_resources\_period\_milliseconds=5000

### Debugging Tools

1. Ray Status Commands:

# Check cluster state  
ray status  
# Check resource usage  
ray status --verbose  
# Check placement groups  
ray status --placement-groups

# Check cluster state  
ray status  
# Check resource usage  
ray status --verbose  
# Check placement groups  
ray status --placement-groups

2. Programmatic Debugging:

# Check scheduling state  
import ray.\_private.state as state  
# Get pending tasks  
pending\_tasks = state.tasks(filters=[("state", "=", "PENDING\_SCHEDULING")])  
# Get resource usage by node  
nodes = state.nodes()  
for node in nodes:  
print(f"Node {node['node\_id']}: {node['resources\_total']}")

# Check scheduling state  
import ray.\_private.state as state  
# Get pending tasks  
pending\_tasks = state.tasks(filters=[("state", "=", "PENDING\_SCHEDULING")])  
# Get resource usage by node  
nodes = state.nodes()  
for node in nodes:  
print(f"Node {node['node\_id']}: {node['resources\_total']}")

3. Logging Configuration:

# Enable debug logging for scheduling  
export RAY\_LOG\_LEVEL=DEBUG  
export RAY\_BACKEND\_LOG\_LEVEL=DEBUG  
# Focus on specific components  
export RAY\_LOG\_TO\_STDERR=1  
ray start --head --log-to-driver

# Enable debug logging for scheduling  
export RAY\_LOG\_LEVEL=DEBUG  
export RAY\_BACKEND\_LOG\_LEVEL=DEBUG  
# Focus on specific components  
export RAY\_LOG\_TO\_STDERR=1  
ray start --head --log-to-driver

### Monitoring and Observability

1. Metrics Collection:

# Custom metrics for scheduling performance  
import ray  
from ray.util.metrics import Counter, Histogram  
scheduling\_latency = Histogram(  
"ray\_scheduling\_latency\_seconds",  
description="Time from task submission to scheduling",  
boundaries=[0.001, 0.01, 0.1, 1.0, 10.0]  
)  
task\_queue\_size = Counter(  
"ray\_task\_queue\_size",  
description="Number of tasks in scheduling queue"  
)

# Custom metrics for scheduling performance  
import ray  
from ray.util.metrics import Counter, Histogram  
scheduling\_latency = Histogram(  
"ray\_scheduling\_latency\_seconds",  
description="Time from task submission to scheduling",  
boundaries=[0.001, 0.01, 0.1, 1.0, 10.0]  
)  
task\_queue\_size = Counter(  
"ray\_task\_queue\_size",  
description="Number of tasks in scheduling queue"  
)

2. Dashboard Integration:  
- Use Ray Dashboard for real-time cluster monitoring  
- Monitor resource utilization trends  
- Track placement group creation success rates  
- Observe task scheduling patterns  
This comprehensive guide covers Ray's distributed scheduling system from architecture to implementation details, providing developers and operators with the knowledge needed to effectively use and optimize Ray's scheduling capabilities in production environments.

# Chapter 2: The Ray Driver System

## Table of Contents

Introduction

Driver Architecture Overview

Driver Lifecycle Deep Dive

Communication Mechanisms

Driver-GCS Integration

Driver-Raylet Communication

Object Management and References

Task and Actor Submission

Error Handling and Fault Tolerance

Performance Optimization

Code Navigation Guide

Common Patterns and Best Practices

Troubleshooting and Debugging

## Introduction

The Ray driver is like the conductor of an orchestra - it coordinates all the distributed computation in your Ray cluster. When you run a Python script with ray.init(), that script becomes the driver process. The driver is responsible for submitting tasks, creating actors, managing object references, and collecting results from the distributed cluster.

ray.init()

### What Makes the Ray Driver Special?

Centralized Control with Distributed Execution: The driver provides a single point of control for your distributed program while execution happens across many machines. Think of it as the "brain" that sends instructions to "hands" (workers) throughout the cluster.  
Seamless Local-to-Distributed: Your Python code looks almost identical whether running locally or on a 1000-node cluster. The driver handles all the complexity of distribution transparently.  
Fault-Tolerant Coordination: The driver can recover from worker failures, network partitions, and other distributed system challenges while maintaining program correctness.

### Core Driver Responsibilities

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

## Driver Architecture Overview

### High-Level Architecture

The Ray driver is built on a multi-layered architecture where each layer handles specific aspects of distributed computing:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Core Components Deep Dive

#### 1. CoreWorker - The Heart of the Driver

Location: src/ray/core\_worker/core\_worker.h and src/ray/core\_worker/core\_worker.cc  
The CoreWorker is the most important component of the driver. Think of it as the driver's "execution engine" that handles all distributed operations.

src/ray/core\_worker/core\_worker.h

src/ray/core\_worker/core\_worker.cc

class CoreWorker {  
public:  
/// Constructor for driver process  
CoreWorker(const CoreWorkerOptions &options, const WorkerID &worker\_id);  
/// Submit a task for remote execution  
Status SubmitTask(const RayFunction &function,  
const std::vector<std::unique\_ptr<TaskArg>> &args,  
const TaskOptions &task\_options,  
std::vector<rpc::ObjectReference> \*returned\_refs);  
/// Create an actor  
Status CreateActor(const RayFunction &function,  
const std::vector<std::unique\_ptr<TaskArg>> &args,  
const ActorCreationOptions &actor\_creation\_options,  
std::vector<rpc::ObjectReference> \*returned\_refs);  
/// Get objects from the object store  
Status Get(const std::vector<ObjectID> &ids,  
int64\_t timeout\_ms,  
std::vector<std::shared\_ptr<RayObject>> \*results);  
/// Put an object into the object store  
Status Put(const RayObject &object,  
const std::vector<ObjectID> &contained\_object\_ids,  
ObjectID \*object\_id);  
};

class CoreWorker {  
public:  
/// Constructor for driver process  
CoreWorker(const CoreWorkerOptions &options, const WorkerID &worker\_id);  
/// Submit a task for remote execution  
Status SubmitTask(const RayFunction &function,  
const std::vector<std::unique\_ptr<TaskArg>> &args,  
const TaskOptions &task\_options,  
std::vector<rpc::ObjectReference> \*returned\_refs);  
/// Create an actor  
Status CreateActor(const RayFunction &function,  
const std::vector<std::unique\_ptr<TaskArg>> &args,  
const ActorCreationOptions &actor\_creation\_options,  
std::vector<rpc::ObjectReference> \*returned\_refs);  
/// Get objects from the object store  
Status Get(const std::vector<ObjectID> &ids,  
int64\_t timeout\_ms,  
std::vector<std::shared\_ptr<RayObject>> \*results);  
/// Put an object into the object store  
Status Put(const RayObject &object,  
const std::vector<ObjectID> &contained\_object\_ids,  
ObjectID \*object\_id);  
};

What the CoreWorker Does (In Simple Terms):  
- Task Coordinator: When you call a @ray.remote function, CoreWorker packages it up and sends it to the right worker  
- Object Tracker: Keeps track of all the data objects your program creates and where they're stored  
- Communication Hub: Manages all the network connections to GCS, raylets, and other workers  
- Memory Manager: Handles garbage collection of distributed objects when they're no longer needed

#### 2. Task Management System

Location: src/ray/core\_worker/task\_manager.h

src/ray/core\_worker/task\_manager.h

class TaskManager {  
private:  
/// Map from task ID to task specification and metadata  
absl::flat\_hash\_map<TaskID, TaskSpec> submittable\_tasks\_;  
/// Tasks that have been submitted but not yet completed  
absl::flat\_hash\_map<TaskID, rpc::TaskStatus> pending\_tasks\_;  
public:  
/// Add a task that is pending execution  
void AddPendingTask(const TaskID &task\_id,  
const TaskSpec &spec,  
const std::string &call\_site);  
/// Mark a task as completed and handle its return values  
void CompletePendingTask(const TaskID &task\_id,  
const rpc::PushTaskReply &reply,  
const rpc::Address &worker\_addr);  
/// Handle task failure and potential retry  
void FailPendingTask(const TaskID &task\_id,  
rpc::ErrorType error\_type,  
const Status \*status);  
};

class TaskManager {  
private:  
/// Map from task ID to task specification and metadata  
absl::flat\_hash\_map<TaskID, TaskSpec> submittable\_tasks\_;  
/// Tasks that have been submitted but not yet completed  
absl::flat\_hash\_map<TaskID, rpc::TaskStatus> pending\_tasks\_;  
public:  
/// Add a task that is pending execution  
void AddPendingTask(const TaskID &task\_id,  
const TaskSpec &spec,  
const std::string &call\_site);  
/// Mark a task as completed and handle its return values  
void CompletePendingTask(const TaskID &task\_id,  
const rpc::PushTaskReply &reply,  
const rpc::Address &worker\_addr);  
/// Handle task failure and potential retry  
void FailPendingTask(const TaskID &task\_id,  
rpc::ErrorType error\_type,  
const Status \*status);  
};

#### 3. Actor Management System

Location: src/ray/core\_worker/actor\_manager.h

src/ray/core\_worker/actor\_manager.h

class ActorManager {  
private:  
/// Map from actor ID to actor handle information  
absl::flat\_hash\_map<ActorID, ActorHandle> actor\_handles\_;  
/// Actors created by this worker  
absl::flat\_hash\_map<ActorID, std::unique\_ptr<ActorCreationState>> created\_actors\_;  
public:  
/// Create a new actor  
Status CreateActor(const TaskSpec &task\_spec,  
const gcs::ActorCreationOptions &options,  
std::vector<rpc::ObjectReference> \*returned\_refs);  
/// Submit a task to an existing actor  
Status SubmitActorTask(const ActorID &actor\_id,  
const TaskSpec &task\_spec,  
std::vector<rpc::ObjectReference> \*returned\_refs);  
/// Handle actor death and cleanup  
void HandleActorStateNotification(const ActorID &actor\_id,  
const gcs::ActorTableData &actor\_data);  
};

class ActorManager {  
private:  
/// Map from actor ID to actor handle information  
absl::flat\_hash\_map<ActorID, ActorHandle> actor\_handles\_;  
/// Actors created by this worker  
absl::flat\_hash\_map<ActorID, std::unique\_ptr<ActorCreationState>> created\_actors\_;  
public:  
/// Create a new actor  
Status CreateActor(const TaskSpec &task\_spec,  
const gcs::ActorCreationOptions &options,  
std::vector<rpc::ObjectReference> \*returned\_refs);  
/// Submit a task to an existing actor  
Status SubmitActorTask(const ActorID &actor\_id,  
const TaskSpec &task\_spec,  
std::vector<rpc::ObjectReference> \*returned\_refs);  
/// Handle actor death and cleanup  
void HandleActorStateNotification(const ActorID &actor\_id,  
const gcs::ActorTableData &actor\_data);  
};

## Driver Lifecycle Deep Dive

### Phase 1: Initialization (ray.init())

ray.init()

When you call ray.init(), a complex initialization sequence begins:  
📊 SEQUENCE DIAGRAM: Process Flow and Interactions  
Detailed Initialization Steps:  
1. Configuration Resolution: Ray determines cluster address, resources, and other settings  
2. CoreWorker Creation: The main driver execution engine is initialized  
3. GCS Connection: Establishes connection to cluster metadata service  
4. Raylet Connection: Connects to local scheduling and execution service  
5. Object Store Connection: Sets up shared memory access for data storage  
6. Driver Registration: Registers with GCS as a special "driver" worker type

ray.init()

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

def init(address=None,  
num\_cpus=None,  
num\_gpus=None,  
resources=None,  
object\_store\_memory=None,  
local\_mode=False,  
\*\*kwargs):  
"""Initialize Ray for distributed computing."""  
# Step 1: Process configuration  
config = \_load\_config(kwargs)  
if address is None:  
# Start local cluster  
\_global\_node = ray.\_private.node.Node(  
head=True,  
shutdown\_at\_exit=True,  
ray\_params=ray\_params)  
else:  
# Connect to existing cluster  
ray\_params.update\_if\_absent(redis\_address=address)  
# Step 3: Initialize CoreWorker  
worker = Worker()  
worker.mode = LOCAL\_MODE if local\_mode else WORKER\_MODE  
# Step 4: Connect to services  
gcs\_client = GcsClient(address=gcs\_address)  
worker.gcs\_client = gcs\_client  
worker.worker\_id = ray.\_private.utils.compute\_driver\_id\_from\_job(  
job\_id, ray\_params.driver\_id)  
\_global\_worker = worker  
worker.check\_connected()

def init(address=None,  
num\_cpus=None,  
num\_gpus=None,  
resources=None,  
object\_store\_memory=None,  
local\_mode=False,  
\*\*kwargs):  
"""Initialize Ray for distributed computing."""  
# Step 1: Process configuration  
config = \_load\_config(kwargs)  
if address is None:  
# Start local cluster  
\_global\_node = ray.\_private.node.Node(  
head=True,  
shutdown\_at\_exit=True,  
ray\_params=ray\_params)  
else:  
# Connect to existing cluster  
ray\_params.update\_if\_absent(redis\_address=address)  
# Step 3: Initialize CoreWorker  
worker = Worker()  
worker.mode = LOCAL\_MODE if local\_mode else WORKER\_MODE  
# Step 4: Connect to services  
gcs\_client = GcsClient(address=gcs\_address)  
worker.gcs\_client = gcs\_client  
worker.worker\_id = ray.\_private.utils.compute\_driver\_id\_from\_job(  
job\_id, ray\_params.driver\_id)  
\_global\_worker = worker  
worker.check\_connected()

### Phase 2: Task and Actor Submission

#### Task Submission Flow

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships  
Code Deep Dive - Task Submission:

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

// From src/ray/core\_worker/core\_worker.cc  
Status CoreWorker::SubmitTask(const RayFunction &function,  
const std::vector<std::unique\_ptr<TaskArg>> &args,  
const TaskOptions &task\_options,  
std::vector<rpc::ObjectReference> \*returned\_refs) {  
// Step 1: Create unique task ID  
const TaskID task\_id = TaskID::FromRandom();  
// Step 2: Build task specification  
TaskSpecBuilder builder;  
builder.SetCommonTaskSpec(task\_id, function.GetLanguage(),  
function.GetFunctionDescriptor(),  
job\_id\_, task\_id, /\*parent\_counter=\*/0,  
caller\_id\_, rpc\_address\_,  
task\_options.resources,  
task\_options.placement\_group\_bundle\_index);  
// Step 3: Add function arguments  
for (const auto &arg : args) {  
if (arg->IsPassedByReference()) {  
builder.AddByRefArg(arg->GetReference());  
} else {  
builder.AddByValueArg(\*arg->GetValue());  
}  
}  
const TaskSpec task\_spec = builder.Build();  
// Step 4: Generate return object references  
for (int i = 0; i < task\_spec.NumReturns(); i++) {  
returned\_refs->emplace\_back();  
returned\_refs->back().set\_object\_id(  
ObjectID::FromIndex(task\_id, i + 1).Binary());  
}  
// Step 5: Submit to task manager for tracking  
task\_manager\_->AddPendingTask(task\_id, task\_spec, "");  
// Step 6: Send to raylet for scheduling  
return raylet\_client\_->SubmitTask(task\_spec, task\_options.concurrency\_group\_name);  
}

// From src/ray/core\_worker/core\_worker.cc  
Status CoreWorker::SubmitTask(const RayFunction &function,  
const std::vector<std::unique\_ptr<TaskArg>> &args,  
const TaskOptions &task\_options,  
std::vector<rpc::ObjectReference> \*returned\_refs) {  
// Step 1: Create unique task ID  
const TaskID task\_id = TaskID::FromRandom();  
// Step 2: Build task specification  
TaskSpecBuilder builder;  
builder.SetCommonTaskSpec(task\_id, function.GetLanguage(),  
function.GetFunctionDescriptor(),  
job\_id\_, task\_id, /\*parent\_counter=\*/0,  
caller\_id\_, rpc\_address\_,  
task\_options.resources,  
task\_options.placement\_group\_bundle\_index);  
// Step 3: Add function arguments  
for (const auto &arg : args) {  
if (arg->IsPassedByReference()) {  
builder.AddByRefArg(arg->GetReference());  
} else {  
builder.AddByValueArg(\*arg->GetValue());  
}  
}  
const TaskSpec task\_spec = builder.Build();  
// Step 4: Generate return object references  
for (int i = 0; i < task\_spec.NumReturns(); i++) {  
returned\_refs->emplace\_back();  
returned\_refs->back().set\_object\_id(  
ObjectID::FromIndex(task\_id, i + 1).Binary());  
}  
// Step 5: Submit to task manager for tracking  
task\_manager\_->AddPendingTask(task\_id, task\_spec, "");  
// Step 6: Send to raylet for scheduling  
return raylet\_client\_->SubmitTask(task\_spec, task\_options.concurrency\_group\_name);  
}

### Phase 3: Result Collection and Object Management

#### Object Reference System

Ray uses a sophisticated object reference system where the driver tracks references to distributed objects:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Phase 4: Cleanup and Shutdown

When the driver shuts down, it must carefully clean up all distributed resources:

def shutdown(verbose=True):  
"""Clean shutdown of Ray driver."""  
# Step 1: Cancel all pending tasks  
\_global\_worker.core\_worker.cancel\_all\_tasks()  
for actor\_id in \_global\_worker.actor\_handles:  
\_global\_worker.core\_worker.kill\_actor(actor\_id, no\_restart=True)  
# Step 3: Clean up object references  
\_global\_worker.core\_worker.shutdown()  
# Step 4: Disconnect from cluster services  
if \_global\_worker.gcs\_client:  
\_global\_worker.gcs\_client.disconnect()  
# Step 5: Cleanup local services if running standalone  
if \_global\_node:  
\_global\_node.kill\_all\_processes()

def shutdown(verbose=True):  
"""Clean shutdown of Ray driver."""  
# Step 1: Cancel all pending tasks  
\_global\_worker.core\_worker.cancel\_all\_tasks()  
for actor\_id in \_global\_worker.actor\_handles:  
\_global\_worker.core\_worker.kill\_actor(actor\_id, no\_restart=True)  
# Step 3: Clean up object references  
\_global\_worker.core\_worker.shutdown()  
# Step 4: Disconnect from cluster services  
if \_global\_worker.gcs\_client:  
\_global\_worker.gcs\_client.disconnect()  
# Step 5: Cleanup local services if running standalone  
if \_global\_node:  
\_global\_node.kill\_all\_processes()

## Communication Mechanisms

The Ray driver uses multiple communication channels optimized for different types of operations:

### 1. Driver-to-GCS Communication

Purpose: Cluster metadata, actor lifecycle, job management  
📊 SEQUENCE DIAGRAM: Process Flow and Interactions  
Code Example - GCS Client:

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

// From src/ray/gcs/gcs\_client/gcs\_client.h  
class GcsClient {  
public:  
/// Create an actor via GCS  
Status CreateActor(const TaskSpec &task\_spec,  
const gcs::ActorCreationOptions &options,  
std::vector<rpc::ObjectReference> \*returned\_refs) {  
rpc::CreateActorRequest request;  
request.mutable\_task\_spec()->CopyFrom(task\_spec.GetMessage());  
request.mutable\_options()->CopyFrom(options);  
return actor\_accessor\_->AsyncCreateActor(  
request,  
[this, returned\_refs](Status status, const rpc::CreateActorReply &reply) {  
if (status.ok()) {  
// Extract actor handle and return references  
for (const auto &ref : reply.returned\_refs()) {  
returned\_refs->push\_back(ref);  
}  
}  
});  
}  
};

// From src/ray/gcs/gcs\_client/gcs\_client.h  
class GcsClient {  
public:  
/// Create an actor via GCS  
Status CreateActor(const TaskSpec &task\_spec,  
const gcs::ActorCreationOptions &options,  
std::vector<rpc::ObjectReference> \*returned\_refs) {  
rpc::CreateActorRequest request;  
request.mutable\_task\_spec()->CopyFrom(task\_spec.GetMessage());  
request.mutable\_options()->CopyFrom(options);  
return actor\_accessor\_->AsyncCreateActor(  
request,  
[this, returned\_refs](Status status, const rpc::CreateActorReply &reply) {  
if (status.ok()) {  
// Extract actor handle and return references  
for (const auto &ref : reply.returned\_refs()) {  
returned\_refs->push\_back(ref);  
}  
}  
});  
}  
};

### 2. Driver-to-Raylet Communication

Purpose: Task submission, resource requests, local scheduling  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### 3. Driver-to-Object Store Communication

Purpose: High-bandwidth data transfer, shared memory access  
The driver accesses the object store through optimized shared memory interfaces:

// From src/ray/object\_store/plasma/client.h  
class PlasmaClient {  
public:  
/// Get objects from local object store  
Status Get(const std::vector<ObjectID> &object\_ids,  
int64\_t timeout\_ms,  
std::vector<ObjectBuffer> \*object\_buffers) {  
// Step 1: Check local availability  
std::vector<plasma::ObjectBuffer> results(object\_ids.size());  
// Step 2: Wait for objects if needed  
Status wait\_status = impl\_->Wait(object\_ids, timeout\_ms, &results);  
// Step 3: Map shared memory segments  
for (size\_t i = 0; i < results.size(); i++) {  
if (results[i].data != nullptr) {  
object\_buffers->emplace\_back(results[i].data, results[i].data\_size);  
}  
}  
return wait\_status;  
}  
/// Put object into local object store  
Status Put(const ray::ObjectID &object\_id,  
const uint8\_t \*data,  
size\_t data\_size) {  
// Step 1: Create plasma object  
std::shared\_ptr<Buffer> buffer;  
Status create\_status = impl\_->Create(object\_id, data\_size, &buffer);  
// Step 2: Copy data into shared memory  
std::memcpy(buffer->mutable\_data(), data, data\_size);  
// Step 3: Seal object (make immutable)  
return impl\_->Seal(object\_id);  
}  
};

// From src/ray/object\_store/plasma/client.h  
class PlasmaClient {  
public:  
/// Get objects from local object store  
Status Get(const std::vector<ObjectID> &object\_ids,  
int64\_t timeout\_ms,  
std::vector<ObjectBuffer> \*object\_buffers) {  
// Step 1: Check local availability  
std::vector<plasma::ObjectBuffer> results(object\_ids.size());  
// Step 2: Wait for objects if needed  
Status wait\_status = impl\_->Wait(object\_ids, timeout\_ms, &results);  
// Step 3: Map shared memory segments  
for (size\_t i = 0; i < results.size(); i++) {  
if (results[i].data != nullptr) {  
object\_buffers->emplace\_back(results[i].data, results[i].data\_size);  
}  
}  
return wait\_status;  
}  
/// Put object into local object store  
Status Put(const ray::ObjectID &object\_id,  
const uint8\_t \*data,  
size\_t data\_size) {  
// Step 1: Create plasma object  
std::shared\_ptr<Buffer> buffer;  
Status create\_status = impl\_->Create(object\_id, data\_size, &buffer);  
// Step 2: Copy data into shared memory  
std::memcpy(buffer->mutable\_data(), data, data\_size);  
// Step 3: Seal object (make immutable)  
return impl\_->Seal(object\_id);  
}  
};

## Driver-GCS Integration

The Global Control Service (GCS) acts as the cluster's "central nervous system" and the driver maintains a close relationship with it:

### Actor Lifecycle Management

📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

### Job Management and Driver Registration

// From src/ray/gcs/gcs\_server/gcs\_job\_manager.h  
class GcsJobManager {  
public:  
/// Register a new driver/job with the cluster  
void HandleAddJob(const rpc::AddJobRequest &request,  
rpc::AddJobReply \*reply,  
rpc::SendReplyCallback send\_reply\_callback) {  
// Extract job information  
const auto &job\_data = request.data();  
const JobID job\_id = JobID::FromBinary(job\_data.job\_id());  
// Store job metadata  
auto job\_table\_data = std::make\_shared<rpc::JobTableData>();  
job\_table\_data->CopyFrom(job\_data);  
// Add to job table in persistent store  
auto status = gcs\_table\_storage\_->JobTable().Put(  
job\_id,  
\*job\_table\_data,  
[send\_reply\_callback, reply](Status status) {  
reply->set\_success(status.ok());  
send\_reply\_callback(status, nullptr, nullptr);  
});  
}  
};

// From src/ray/gcs/gcs\_server/gcs\_job\_manager.h  
class GcsJobManager {  
public:  
/// Register a new driver/job with the cluster  
void HandleAddJob(const rpc::AddJobRequest &request,  
rpc::AddJobReply \*reply,  
rpc::SendReplyCallback send\_reply\_callback) {  
// Extract job information  
const auto &job\_data = request.data();  
const JobID job\_id = JobID::FromBinary(job\_data.job\_id());  
// Store job metadata  
auto job\_table\_data = std::make\_shared<rpc::JobTableData>();  
job\_table\_data->CopyFrom(job\_data);  
// Add to job table in persistent store  
auto status = gcs\_table\_storage\_->JobTable().Put(  
job\_id,  
\*job\_table\_data,  
[send\_reply\_callback, reply](Status status) {  
reply->set\_success(status.ok());  
send\_reply\_callback(status, nullptr, nullptr);  
});  
}  
};

### Resource Management Integration

The driver coordinates with GCS for cluster-wide resource management:

@ray.remote(num\_cpus=4, num\_gpus=1, memory=8000)  
def gpu\_task(data):  
# This task needs specific resources  
return process\_on\_gpu(data)  
# 1. Registers resource requirements with GCS  
# 2. GCS finds nodes with available resources

@ray.remote(num\_cpus=4, num\_gpus=1, memory=8000)  
def gpu\_task(data):  
# This task needs specific resources  
return process\_on\_gpu(data)  
# 1. Registers resource requirements with GCS  
# 2. GCS finds nodes with available resources

## Code Navigation Guide

### Key Entry Points for Driver Functionality

#### 1. Python API Layer

Location: python/ray/\_private/worker.py  
This is where the user-facing Ray API is implemented:

python/ray/\_private/worker.py

# Main initialization  
def init(...) -> ray.init()  
# Task submission  
class RemoteFunction:  
def remote(self, \*args, \*\*kwargs) -> ObjectRef  
# Object operations  
def get(object\_refs, timeout=None) -> ray.get()  
def put(value) -> ray.put()  
def wait(object\_refs, num\_returns=1, timeout=None) -> ray.wait()

# Main initialization  
def init(...) -> ray.init()  
# Task submission  
class RemoteFunction:  
def remote(self, \*args, \*\*kwargs) -> ObjectRef  
# Object operations  
def get(object\_refs, timeout=None) -> ray.get()  
def put(value) -> ray.put()  
def wait(object\_refs, num\_returns=1, timeout=None) -> ray.wait()

#### 2. CoreWorker Implementation

Location: src/ray/core\_worker/core\_worker.{h,cc}  
The main C++ driver implementation:

src/ray/core\_worker/core\_worker.{h,cc}

// Key methods for understanding driver behavior:  
Status CoreWorker::SubmitTask(...) // Task submission logic  
Status CoreWorker::CreateActor(...) // Actor creation logic  
Status CoreWorker::Get(...) // Object retrieval logic  
Status CoreWorker::Put(...) // Object storage logic

// Key methods for understanding driver behavior:  
Status CoreWorker::SubmitTask(...) // Task submission logic  
Status CoreWorker::CreateActor(...) // Actor creation logic  
Status CoreWorker::Get(...) // Object retrieval logic  
Status CoreWorker::Put(...) // Object storage logic

#### 3. Task and Actor Management

Location: src/ray/core\_worker/task\_manager.{h,cc} and src/ray/core\_worker/actor\_manager.{h,cc}

src/ray/core\_worker/task\_manager.{h,cc}

src/ray/core\_worker/actor\_manager.{h,cc}

class TaskManager {  
void AddPendingTask(...) // Track submitted tasks  
void CompletePendingTask(...) // Handle task completion  
void FailPendingTask(...) // Handle task failures  
};  
class ActorManager {  
Status CreateActor(...) // Actor lifecycle start  
Status SubmitActorTask(...) // Send methods to actors  
void HandleActorStateNotification(...) // React to actor events  
};

class TaskManager {  
void AddPendingTask(...) // Track submitted tasks  
void CompletePendingTask(...) // Handle task completion  
void FailPendingTask(...) // Handle task failures  
};  
class ActorManager {  
Status CreateActor(...) // Actor lifecycle start  
Status SubmitActorTask(...) // Send methods to actors  
void HandleActorStateNotification(...) // React to actor events  
};

#### 4. Communication Layers

Location: src/ray/rpc/ and src/ray/core\_worker/transport/

src/ray/rpc/

src/ray/core\_worker/transport/

// GCS communication  
class GcsClient : public GcsClientInterface {...}  
// Raylet communication  
class CoreWorkerRayletTaskSubmitter {...}  
// Direct worker communication  
class CoreWorkerDirectTaskSubmitter {...}

// GCS communication  
class GcsClient : public GcsClientInterface {...}  
// Raylet communication  
class CoreWorkerRayletTaskSubmitter {...}  
// Direct worker communication  
class CoreWorkerDirectTaskSubmitter {...}

### Debugging and Instrumentation Points

#### 1. Driver State Inspection

import ray  
worker = ray.\_private.worker.global\_worker  
# View pending tasks  
print(f"Pending tasks: {len(worker.core\_worker.get\_all\_pending\_tasks())}")  
# View actor handles  
print(f"Actor handles: {len(worker.actor\_handles)}")  
# View object references  
print(f"Object refs in scope: {worker.core\_worker.get\_objects\_in\_scope()}")

import ray  
worker = ray.\_private.worker.global\_worker  
# View pending tasks  
print(f"Pending tasks: {len(worker.core\_worker.get\_all\_pending\_tasks())}")  
# View actor handles  
print(f"Actor handles: {len(worker.actor\_handles)}")  
# View object references  
print(f"Object refs in scope: {worker.core\_worker.get\_objects\_in\_scope()}")

#### 2. Enable Detailed Logging

import logging  
logging.getLogger("ray.core\_worker").setLevel(logging.DEBUG)  
logging.getLogger("ray.gcs\_client").setLevel(logging.DEBUG)

import logging  
logging.getLogger("ray.core\_worker").setLevel(logging.DEBUG)  
logging.getLogger("ray.gcs\_client").setLevel(logging.DEBUG)

#### 3. Ray Status and Debugging Tools

ray status  
ray logs --actor-id <driver-worker-id>  
# Monitor object references  
ray memory --stats-only

ray status  
ray logs --actor-id <driver-worker-id>  
# Monitor object references  
ray memory --stats-only

This comprehensive guide provides the foundation for understanding Ray's driver implementation. The driver serves as the central coordinator for distributed Ray applications, managing task submission, actor lifecycles, object references, and communication with cluster services through sophisticated APIs and communication protocols.

# Chapter 3: Task Lifecycle and Management

## Table of Contents

Introduction

Task Architecture Overview

Task Creation and Submission

Task Scheduling and Placement

Task Execution Engine

Task Dependencies and Lineage

Error Handling and Retry Logic

Performance Optimization

Code Navigation Guide

## Introduction

Ray tasks are the fundamental units of computation in the Ray ecosystem. Think of a task as a function call that can run anywhere in your cluster - it could execute on your local machine, a machine in another data center, or even on a different cloud provider. Tasks are stateless, immutable, and designed for maximum parallelism.

### What Makes Ray Tasks Special?

Stateless Execution: Tasks don't maintain state between calls, making them easy to distribute, retry, and scale horizontally.  
Automatic Parallelism: When you call a remote function, Ray automatically distributes the work across available workers without you having to think about threads, processes, or network communication.  
Fault Tolerance: If a task fails, Ray can automatically retry it on different machines, ensuring your computation completes even in the face of hardware failures.  
Efficient Data Sharing: Tasks can share large datasets efficiently through Ray's distributed object store without copying data unnecessarily.

### Core Task Concepts

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

## Task Architecture Overview

### High-Level Task System Architecture

Ray's task system is built on multiple layers that handle different aspects of distributed task execution:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Task vs Actor Comparison

Understanding the differences between tasks and actors is crucial for designing Ray applications:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

## Task Creation and Submission

### Phase 1: Function Registration

When you decorate a function with @ray.remote, Ray prepares it for distributed execution:

@ray.remote

# User code  
@ray.remote(num\_cpus=2, memory=1000)  
def process\_data(data\_chunk, model\_params):  
"""Example computation-intensive task"""  
import numpy as np  
# Simulate data processing  
processed = np.array(data\_chunk) \* np.array(model\_params)  
result = np.sum(processed \*\* 2)  
return {  
'result': result,  
'chunk\_size': len(data\_chunk),  
'processing\_time': time.time()  
}  
data\_chunks = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
model\_params = [0.1, 0.2, 0.3]  
# These calls return immediately with ObjectRefs  
futures = [process\_data.remote(chunk, model\_params) for chunk in data\_chunks]  
# Retrieve results when needed  
results = ray.get(futures)

# User code  
@ray.remote(num\_cpus=2, memory=1000)  
def process\_data(data\_chunk, model\_params):  
"""Example computation-intensive task"""  
import numpy as np  
# Simulate data processing  
processed = np.array(data\_chunk) \* np.array(model\_params)  
result = np.sum(processed \*\* 2)  
return {  
'result': result,  
'chunk\_size': len(data\_chunk),  
'processing\_time': time.time()  
}  
data\_chunks = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
model\_params = [0.1, 0.2, 0.3]  
# These calls return immediately with ObjectRefs  
futures = [process\_data.remote(chunk, model\_params) for chunk in data\_chunks]  
# Retrieve results when needed  
results = ray.get(futures)

Behind the Scenes - Function Registration:

# From python/ray/\_private/worker.py  
def make\_function\_remote(function, num\_cpus, num\_gpus, memory, \*\*kwargs):  
"""Convert a regular function into a Ray remote function."""  
# Step 1: Create function metadata  
function\_id = compute\_function\_id(function)  
# Step 2: Register function with driver's core worker  
driver\_worker = ray.\_private.worker.global\_worker  
driver\_worker.function\_actor\_manager.export\_function(  
function, function\_id, num\_cpus, num\_gpus, memory)  
def remote(\*args, \*\*kwargs):  
return RemoteFunction.\_remote(  
args=args, kwargs=kwargs,  
num\_cpus=num\_cpus, num\_gpus=num\_gpus, memory=memory)  
# Step 4: Return enhanced function  
function.remote = remote  
return function

# From python/ray/\_private/worker.py  
def make\_function\_remote(function, num\_cpus, num\_gpus, memory, \*\*kwargs):  
"""Convert a regular function into a Ray remote function."""  
# Step 1: Create function metadata  
function\_id = compute\_function\_id(function)  
# Step 2: Register function with driver's core worker  
driver\_worker = ray.\_private.worker.global\_worker  
driver\_worker.function\_actor\_manager.export\_function(  
function, function\_id, num\_cpus, num\_gpus, memory)  
def remote(\*args, \*\*kwargs):  
return RemoteFunction.\_remote(  
args=args, kwargs=kwargs,  
num\_cpus=num\_cpus, num\_gpus=num\_gpus, memory=memory)  
# Step 4: Return enhanced function  
function.remote = remote  
return function

### Phase 2: Task Specification Creation

When you call function.remote(), Ray creates a detailed task specification:  
📊 SEQUENCE DIAGRAM: Process Flow and Interactions  
Detailed Task Specification Code:

function.remote()

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

// From src/ray/core\_worker/core\_worker.cc  
Status CoreWorker::SubmitTask(const RayFunction &function,  
const std::vector<std::unique\_ptr<TaskArg>> &args,  
const TaskOptions &task\_options,  
std::vector<rpc::ObjectReference> \*returned\_refs) {  
// Step 1: Generate unique task ID  
const TaskID task\_id = TaskID::FromRandom();  
// Step 2: Build comprehensive task specification  
TaskSpecBuilder builder;  
builder.SetCommonTaskSpec(  
task\_id, // Unique identifier  
function.GetLanguage(), // Python/Java/C++  
function.GetFunctionDescriptor(), // Function metadata  
job\_id\_, // Current job  
TaskID::Nil(), // Parent task (for nested)  
/\*parent\_counter=\*/0, // Ordering within parent  
caller\_id\_, // Calling worker ID  
rpc\_address\_, // Return address  
task\_options.resources, // Resource requirements  
task\_options.placement\_group\_bundle\_index // Placement constraints  
);  
// Step 3: Process function arguments  
for (size\_t i = 0; i < args.size(); i++) {  
const auto &arg = args[i];  
if (arg->IsPassedByReference()) {  
// Argument is an ObjectRef from another task  
builder.AddByRefArg(arg->GetReference());  
} else {  
// Argument is a direct value (serialized)  
builder.AddByValueArg(\*arg->GetValue());  
}  
}  
const TaskSpec task\_spec = builder.Build();  
// Step 4: Create return object references  
for (int i = 0; i < task\_spec.NumReturns(); i++) {  
returned\_refs->emplace\_back();  
returned\_refs->back().set\_object\_id(  
ObjectID::FromIndex(task\_id, i + 1).Binary());  
returned\_refs->back().set\_owner\_id(GetWorkerID().Binary());  
}  
// Step 5: Submit to task manager for tracking  
task\_manager\_->AddPendingTask(task\_id, task\_spec, "user\_task");  
// Step 6: Forward to appropriate scheduler  
return raylet\_client\_->SubmitTask(task\_spec, "");  
}

// From src/ray/core\_worker/core\_worker.cc  
Status CoreWorker::SubmitTask(const RayFunction &function,  
const std::vector<std::unique\_ptr<TaskArg>> &args,  
const TaskOptions &task\_options,  
std::vector<rpc::ObjectReference> \*returned\_refs) {  
// Step 1: Generate unique task ID  
const TaskID task\_id = TaskID::FromRandom();  
// Step 2: Build comprehensive task specification  
TaskSpecBuilder builder;  
builder.SetCommonTaskSpec(  
task\_id, // Unique identifier  
function.GetLanguage(), // Python/Java/C++  
function.GetFunctionDescriptor(), // Function metadata  
job\_id\_, // Current job  
TaskID::Nil(), // Parent task (for nested)  
/\*parent\_counter=\*/0, // Ordering within parent  
caller\_id\_, // Calling worker ID  
rpc\_address\_, // Return address  
task\_options.resources, // Resource requirements  
task\_options.placement\_group\_bundle\_index // Placement constraints  
);  
// Step 3: Process function arguments  
for (size\_t i = 0; i < args.size(); i++) {  
const auto &arg = args[i];  
if (arg->IsPassedByReference()) {  
// Argument is an ObjectRef from another task  
builder.AddByRefArg(arg->GetReference());  
} else {  
// Argument is a direct value (serialized)  
builder.AddByValueArg(\*arg->GetValue());  
}  
}  
const TaskSpec task\_spec = builder.Build();  
// Step 4: Create return object references  
for (int i = 0; i < task\_spec.NumReturns(); i++) {  
returned\_refs->emplace\_back();  
returned\_refs->back().set\_object\_id(  
ObjectID::FromIndex(task\_id, i + 1).Binary());  
returned\_refs->back().set\_owner\_id(GetWorkerID().Binary());  
}  
// Step 5: Submit to task manager for tracking  
task\_manager\_->AddPendingTask(task\_id, task\_spec, "user\_task");  
// Step 6: Forward to appropriate scheduler  
return raylet\_client\_->SubmitTask(task\_spec, "");  
}

### Phase 3: Argument Processing and Serialization

Ray carefully handles different types of task arguments:

# Example: Different argument types  
@ray.remote  
def complex\_task(  
simple\_value, # Serialized directly  
numpy\_array, # Efficient serialization  
object\_ref, # Reference to distributed object  
large\_dataset, # Stored in object store  
custom\_object # User-defined class  
):  
# Function body  
pass  
# Different ways to pass arguments  
simple\_result = ray.put("large data") # Explicit put  
array\_result = other\_task.remote() # Task dependency  
large\_data = np.random.random((1000000,)) # Auto-stored  
# All argument types in one call  
result = complex\_task.remote(  
42, # Simple value  
np.array([1, 2, 3]), # Small array (serialized)  
array\_result, # ObjectRef dependency  
large\_data, # Large data (auto-put)  
MyCustomClass() # Custom object  
)

# Example: Different argument types  
@ray.remote  
def complex\_task(  
simple\_value, # Serialized directly  
numpy\_array, # Efficient serialization  
object\_ref, # Reference to distributed object  
large\_dataset, # Stored in object store  
custom\_object # User-defined class  
):  
# Function body  
pass  
# Different ways to pass arguments  
simple\_result = ray.put("large data") # Explicit put  
array\_result = other\_task.remote() # Task dependency  
large\_data = np.random.random((1000000,)) # Auto-stored  
# All argument types in one call  
result = complex\_task.remote(  
42, # Simple value  
np.array([1, 2, 3]), # Small array (serialized)  
array\_result, # ObjectRef dependency  
large\_data, # Large data (auto-put)  
MyCustomClass() # Custom object  
)

Argument Processing Logic:

// From src/ray/core\_worker/core\_worker.cc  
std::unique\_ptr<TaskArg> CreateTaskArg(const py::object &obj) {  
// Check if object is already an ObjectRef  
if (IsObjectRef(obj)) {  
ObjectID object\_id = GetObjectID(obj);  
return std::make\_unique<TaskArgByReference>(object\_id);  
}  
// Check object size to decide on storage strategy  
size\_t serialized\_size = GetSerializedSize(obj);  
if (serialized\_size > kObjectStoreThreshold) {  
// Large object: store in object store and pass by reference  
ObjectID object\_id;  
Status status = Put(obj, &object\_id);  
RAY\_CHECK\_OK(status);  
return std::make\_unique<TaskArgByReference>(object\_id);  
} else {  
// Small object: serialize and pass by value  
auto serialized\_obj = SerializeObject(obj);  
return std::make\_unique<TaskArgByValue>(std::move(serialized\_obj));  
}  
}

// From src/ray/core\_worker/core\_worker.cc  
std::unique\_ptr<TaskArg> CreateTaskArg(const py::object &obj) {  
// Check if object is already an ObjectRef  
if (IsObjectRef(obj)) {  
ObjectID object\_id = GetObjectID(obj);  
return std::make\_unique<TaskArgByReference>(object\_id);  
}  
// Check object size to decide on storage strategy  
size\_t serialized\_size = GetSerializedSize(obj);  
if (serialized\_size > kObjectStoreThreshold) {  
// Large object: store in object store and pass by reference  
ObjectID object\_id;  
Status status = Put(obj, &object\_id);  
RAY\_CHECK\_OK(status);  
return std::make\_unique<TaskArgByReference>(object\_id);  
} else {  
// Small object: serialize and pass by value  
auto serialized\_obj = SerializeObject(obj);  
return std::make\_unique<TaskArgByValue>(std::move(serialized\_obj));  
}  
}

## Task Scheduling and Placement

### Cluster-Level Task Scheduling

Ray's task scheduler makes intelligent decisions about where to run tasks:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Local Task Scheduling (Raylet)

Once a task arrives at a raylet, local scheduling decisions are made:

// From src/ray/raylet/local\_task\_manager.cc  
void LocalTaskManager::ScheduleAndDispatchTasks() {  
// Step 1: Process tasks waiting for dependencies  
SchedulePendingTasks();  
// Step 2: Dispatch ready tasks to workers  
DispatchScheduledTasksToWorkers();  
// Step 3: Handle task completion and cleanup  
ProcessTaskCompletion();  
}  
void LocalTaskManager::SchedulePendingTasks() {  
auto it = tasks\_to\_schedule\_.begin();  
while (it != tasks\_to\_schedule\_.end()) {  
const auto &task\_id = it->first;  
const auto &task\_spec = it->second;  
// Check if all dependencies are satisfied  
if (task\_dependency\_manager\_->CheckTaskReady(task\_id)) {  
// Check if resources are available  
if (cluster\_resource\_scheduler\_->HasSufficientResource(  
task\_spec.GetRequiredResources())) {  
// Move to dispatch queue  
tasks\_to\_dispatch\_[task\_id] = task\_spec;  
it = tasks\_to\_schedule\_.erase(it);  
// Reserve resources for this task  
cluster\_resource\_scheduler\_->AllocateTaskResources(  
task\_id, task\_spec.GetRequiredResources());  
} else {  
++it; // Keep waiting for resources  
}  
} else {  
++it; // Keep waiting for dependencies  
}  
}  
}

// From src/ray/raylet/local\_task\_manager.cc  
void LocalTaskManager::ScheduleAndDispatchTasks() {  
// Step 1: Process tasks waiting for dependencies  
SchedulePendingTasks();  
// Step 2: Dispatch ready tasks to workers  
DispatchScheduledTasksToWorkers();  
// Step 3: Handle task completion and cleanup  
ProcessTaskCompletion();  
}  
void LocalTaskManager::SchedulePendingTasks() {  
auto it = tasks\_to\_schedule\_.begin();  
while (it != tasks\_to\_schedule\_.end()) {  
const auto &task\_id = it->first;  
const auto &task\_spec = it->second;  
// Check if all dependencies are satisfied  
if (task\_dependency\_manager\_->CheckTaskReady(task\_id)) {  
// Check if resources are available  
if (cluster\_resource\_scheduler\_->HasSufficientResource(  
task\_spec.GetRequiredResources())) {  
// Move to dispatch queue  
tasks\_to\_dispatch\_[task\_id] = task\_spec;  
it = tasks\_to\_schedule\_.erase(it);  
// Reserve resources for this task  
cluster\_resource\_scheduler\_->AllocateTaskResources(  
task\_id, task\_spec.GetRequiredResources());  
} else {  
++it; // Keep waiting for resources  
}  
} else {  
++it; // Keep waiting for dependencies  
}  
}  
}

### Intelligent Worker Selection

The scheduler considers multiple factors when selecting workers:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

## Task Execution Engine

### Worker Process Task Execution

Once a task is assigned to a worker, a sophisticated execution engine takes over:  
📊 SEQUENCE DIAGRAM: Process Flow and Interactions  
Task Execution Implementation:

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

# From python/ray/\_private/worker.py (worker process)  
class TaskExecutor:  
def execute\_task(self, task\_spec, task\_execution\_spec):  
"""Execute a single task in the worker process."""  
function\_descriptor = task\_spec.function\_descriptor  
args = task\_spec.args  
task\_id = task\_spec.task\_id  
# Step 2: Resolve function from registry  
function = worker.function\_actor\_manager.get\_function(function\_descriptor)  
resolved\_args = []  
for arg in args:  
if arg.is\_by\_ref:  
# Resolve ObjectRef to actual value  
obj = ray.get(ObjectRef(arg.object\_ref.object\_id))  
resolved\_args.append(obj)  
else:  
# Deserialize direct value  
obj = ray.\_private.serialization.deserialize(arg.data)  
resolved\_args.append(obj)  
# Step 4: Execute the function  
try:  
with ray.\_private.profiling.profile\_task(task\_id):  
result = function(\*resolved\_args)  
# Step 5: Store result in object store  
if isinstance(result, tuple):  
# Multiple return values  
return\_refs = []  
for i, ret\_val in enumerate(result):  
object\_id = ObjectID.from\_task\_and\_index(task\_id, i + 1)  
ray.put(ret\_val, object\_id=object\_id)  
return\_refs.append(object\_id)  
return return\_refs  
else:  
# Single return value  
object\_id = ObjectID.from\_task\_and\_index(task\_id, 1)  
ray.put(result, object\_id=object\_id)  
return [object\_id]  
except Exception as e:  
error\_info = TaskExecutionError(e, traceback.format\_exc())  
self.\_store\_task\_error(task\_id, error\_info)  
raise

# From python/ray/\_private/worker.py (worker process)  
class TaskExecutor:  
def execute\_task(self, task\_spec, task\_execution\_spec):  
"""Execute a single task in the worker process."""  
function\_descriptor = task\_spec.function\_descriptor  
args = task\_spec.args  
task\_id = task\_spec.task\_id  
# Step 2: Resolve function from registry  
function = worker.function\_actor\_manager.get\_function(function\_descriptor)  
resolved\_args = []  
for arg in args:  
if arg.is\_by\_ref:  
# Resolve ObjectRef to actual value  
obj = ray.get(ObjectRef(arg.object\_ref.object\_id))  
resolved\_args.append(obj)  
else:  
# Deserialize direct value  
obj = ray.\_private.serialization.deserialize(arg.data)  
resolved\_args.append(obj)  
# Step 4: Execute the function  
try:  
with ray.\_private.profiling.profile\_task(task\_id):  
result = function(\*resolved\_args)  
# Step 5: Store result in object store  
if isinstance(result, tuple):  
# Multiple return values  
return\_refs = []  
for i, ret\_val in enumerate(result):  
object\_id = ObjectID.from\_task\_and\_index(task\_id, i + 1)  
ray.put(ret\_val, object\_id=object\_id)  
return\_refs.append(object\_id)  
return return\_refs  
else:  
# Single return value  
object\_id = ObjectID.from\_task\_and\_index(task\_id, 1)  
ray.put(result, object\_id=object\_id)  
return [object\_id]  
except Exception as e:  
error\_info = TaskExecutionError(e, traceback.format\_exc())  
self.\_store\_task\_error(task\_id, error\_info)  
raise

### Dependency Resolution System

Ray automatically resolves task dependencies before execution:

# Example: Complex dependency chain  
@ray.remote  
def load\_data(filename):  
"""Load data from file"""  
import pandas as pd  
return pd.read\_csv(filename)  
@ray.remote  
def preprocess\_data(data):  
"""Clean and prepare data"""  
# Remove nulls, normalize, etc.  
cleaned = data.dropna()  
normalized = (cleaned - cleaned.mean()) / cleaned.std()  
return normalized  
@ray.remote  
def train\_model(train\_data, test\_data):  
"""Train ML model"""  
from sklearn.linear\_model import LinearRegression  
model = LinearRegression()  
model.fit(train\_data[['feature1', 'feature2']], train\_data['target'])  
score = model.score(test\_data[['feature1', 'feature2']], test\_data['target'])  
return {'model': model, 'score': score}  
@ray.remote  
def evaluate\_model(model\_data, validation\_data):  
"""Evaluate trained model"""  
model = model\_data['model']  
predictions = model.predict(validation\_data[['feature1', 'feature2']])  
accuracy = calculate\_accuracy(predictions, validation\_data['target'])  
return accuracy  
# Create dependency graph automatically  
raw\_train = load\_data.remote("train.csv") # Independent  
raw\_test = load\_data.remote("test.csv") # Independent  
raw\_val = load\_data.remote("validation.csv") # Independent  
clean\_train = preprocess\_data.remote(raw\_train) # Depends on raw\_train  
clean\_test = preprocess\_data.remote(raw\_test) # Depends on raw\_test  
clean\_val = preprocess\_data.remote(raw\_val) # Depends on raw\_val  
model\_result = train\_model.remote(clean\_train, clean\_test) # Depends on both  
final\_accuracy = evaluate\_model.remote(model\_result, clean\_val) # Depends on all  
# Ray automatically manages the entire dependency graph  
print(f"Final model accuracy: {ray.get(final\_accuracy)}")

# Example: Complex dependency chain  
@ray.remote  
def load\_data(filename):  
"""Load data from file"""  
import pandas as pd  
return pd.read\_csv(filename)  
@ray.remote  
def preprocess\_data(data):  
"""Clean and prepare data"""  
# Remove nulls, normalize, etc.  
cleaned = data.dropna()  
normalized = (cleaned - cleaned.mean()) / cleaned.std()  
return normalized  
@ray.remote  
def train\_model(train\_data, test\_data):  
"""Train ML model"""  
from sklearn.linear\_model import LinearRegression  
model = LinearRegression()  
model.fit(train\_data[['feature1', 'feature2']], train\_data['target'])  
score = model.score(test\_data[['feature1', 'feature2']], test\_data['target'])  
return {'model': model, 'score': score}  
@ray.remote  
def evaluate\_model(model\_data, validation\_data):  
"""Evaluate trained model"""  
model = model\_data['model']  
predictions = model.predict(validation\_data[['feature1', 'feature2']])  
accuracy = calculate\_accuracy(predictions, validation\_data['target'])  
return accuracy  
# Create dependency graph automatically  
raw\_train = load\_data.remote("train.csv") # Independent  
raw\_test = load\_data.remote("test.csv") # Independent  
raw\_val = load\_data.remote("validation.csv") # Independent  
clean\_train = preprocess\_data.remote(raw\_train) # Depends on raw\_train  
clean\_test = preprocess\_data.remote(raw\_test) # Depends on raw\_test  
clean\_val = preprocess\_data.remote(raw\_val) # Depends on raw\_val  
model\_result = train\_model.remote(clean\_train, clean\_test) # Depends on both  
final\_accuracy = evaluate\_model.remote(model\_result, clean\_val) # Depends on all  
# Ray automatically manages the entire dependency graph  
print(f"Final model accuracy: {ray.get(final\_accuracy)}")

## Task Dependencies and Lineage

### Dependency Graph Management

Ray maintains a sophisticated dependency graph for tasks:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Lineage Tracking and Fault Tolerance

Ray tracks the complete lineage of objects to enable fault tolerance:

// From src/ray/core\_worker/reference\_count.h  
class ReferenceCounter {  
private:  
// Maps object ID to its lineage information  
absl::flat\_hash\_map<ObjectID, ObjectLineage> object\_lineage\_map\_;  
// Maps object ID to the task that created it  
absl::flat\_hash\_map<ObjectID, TaskID> object\_to\_task\_map\_;  
public:  
/// Add lineage information when object is created  
void AddObjectLineage(const ObjectID &object\_id,  
const TaskID &task\_id,  
const std::vector<ObjectID> &dependencies) {  
ObjectLineage lineage;  
lineage.task\_id = task\_id;  
lineage.dependencies = dependencies;  
lineage.creation\_time = absl::Now();  
object\_lineage\_map\_[object\_id] = lineage;  
object\_to\_task\_map\_[object\_id] = task\_id;  
}  
/// Reconstruct object by re-executing its task  
Status ReconstructObject(const ObjectID &object\_id) {  
auto it = object\_lineage\_map\_.find(object\_id);  
if (it == object\_lineage\_map\_.end()) {  
return Status::NotFound("Object lineage not found");  
}  
const auto &lineage = it->second;  
// First ensure all dependencies are available  
for (const auto &dep\_id : lineage.dependencies) {  
if (!IsObjectAvailable(dep\_id)) {  
// Recursively reconstruct dependencies  
auto status = ReconstructObject(dep\_id);  
if (!status.ok()) {  
return status;  
}  
}  
}  
// Re-execute the task that created this object  
return ReExecuteTask(lineage.task\_id);  
}  
};

// From src/ray/core\_worker/reference\_count.h  
class ReferenceCounter {  
private:  
// Maps object ID to its lineage information  
absl::flat\_hash\_map<ObjectID, ObjectLineage> object\_lineage\_map\_;  
// Maps object ID to the task that created it  
absl::flat\_hash\_map<ObjectID, TaskID> object\_to\_task\_map\_;  
public:  
/// Add lineage information when object is created  
void AddObjectLineage(const ObjectID &object\_id,  
const TaskID &task\_id,  
const std::vector<ObjectID> &dependencies) {  
ObjectLineage lineage;  
lineage.task\_id = task\_id;  
lineage.dependencies = dependencies;  
lineage.creation\_time = absl::Now();  
object\_lineage\_map\_[object\_id] = lineage;  
object\_to\_task\_map\_[object\_id] = task\_id;  
}  
/// Reconstruct object by re-executing its task  
Status ReconstructObject(const ObjectID &object\_id) {  
auto it = object\_lineage\_map\_.find(object\_id);  
if (it == object\_lineage\_map\_.end()) {  
return Status::NotFound("Object lineage not found");  
}  
const auto &lineage = it->second;  
// First ensure all dependencies are available  
for (const auto &dep\_id : lineage.dependencies) {  
if (!IsObjectAvailable(dep\_id)) {  
// Recursively reconstruct dependencies  
auto status = ReconstructObject(dep\_id);  
if (!status.ok()) {  
return status;  
}  
}  
}  
// Re-execute the task that created this object  
return ReExecuteTask(lineage.task\_id);  
}  
};

This comprehensive guide covers the essential aspects of Ray's task system, from creation through execution to fault tolerance. Tasks form the foundation of Ray's distributed computing model, enabling scalable and fault-tolerant parallel computation.

# Chapter 4: Actor Lifecycle and Management

## Table of Contents

Introduction

Actor Architecture Overview

Actor Creation Deep Dive

Method Invocation and Execution

Fault Tolerance and Recovery

Performance Optimization

## Introduction

Ray actors are long-running, stateful workers that live somewhere in your cluster and can be called like remote objects. Think of an actor as a combination of a server process and a Python object - it has its own memory, state, and can handle multiple requests over time.

### What Makes Ray Actors Special?

Stateful Distributed Computing: Unlike functions that are stateless, actors maintain state between calls. Imagine having a database connection, machine learning model, or game state that persists across multiple operations.  
Location Transparency: You interact with actors using handles that look like regular Python objects, even though the actor might be running on a machine thousands of miles away.

### Core Actor Concepts

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

## Actor Architecture Overview

### High-Level Actor System Architecture

Ray's actor system is built on several layers that work together to provide the illusion of stateful, distributed objects:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

## Actor Creation Deep Dive

### Phase 1: Actor Definition and Registration

When you define an actor class, Ray prepares it for distributed execution:

# User code  
@ray.remote(num\_cpus=2, num\_gpus=1)  
class GameServer:  
def \_\_init\_\_(self, max\_players=100):  
self.players = {}  
self.max\_players = max\_players  
self.game\_state = "waiting"  
def add\_player(self, player\_id, player\_data):  
if len(self.players) < self.max\_players:  
self.players[player\_id] = player\_data  
return True  
return False  
game\_server = GameServer.remote(max\_players=50)

# User code  
@ray.remote(num\_cpus=2, num\_gpus=1)  
class GameServer:  
def \_\_init\_\_(self, max\_players=100):  
self.players = {}  
self.max\_players = max\_players  
self.game\_state = "waiting"  
def add\_player(self, player\_id, player\_data):  
if len(self.players) < self.max\_players:  
self.players[player\_id] = player\_data  
return True  
return False  
game\_server = GameServer.remote(max\_players=50)

Behind the Scenes - Class Registration:

# From python/ray/\_private/worker.py  
def make\_actor(cls, num\_cpus, num\_gpus, memory, \*\*kwargs):  
"""Convert a regular class into a Ray actor class."""  
class\_id = compute\_class\_id(cls)  
# Step 2: Register class with driver's core worker  
driver\_worker = ray.\_private.worker.global\_worker  
driver\_worker.function\_actor\_manager.export\_actor\_class(  
cls, class\_id, num\_cpus, num\_gpus, memory)  
def remote(\*args, \*\*kwargs):  
return ActorHandle.\_remote(args=args, kwargs=kwargs)  
cls.remote = remote  
return cls

# From python/ray/\_private/worker.py  
def make\_actor(cls, num\_cpus, num\_gpus, memory, \*\*kwargs):  
"""Convert a regular class into a Ray actor class."""  
class\_id = compute\_class\_id(cls)  
# Step 2: Register class with driver's core worker  
driver\_worker = ray.\_private.worker.global\_worker  
driver\_worker.function\_actor\_manager.export\_actor\_class(  
cls, class\_id, num\_cpus, num\_gpus, memory)  
def remote(\*args, \*\*kwargs):  
return ActorHandle.\_remote(args=args, kwargs=kwargs)  
cls.remote = remote  
return cls

### Phase 2: Actor Instance Creation

When you call ClassName.remote(), a complex creation process begins:  
📊 SEQUENCE DIAGRAM: Process Flow and Interactions  
Detailed Actor Creation Code:

ClassName.remote()

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

// From src/ray/core\_worker/core\_worker.cc  
Status CoreWorker::CreateActor(const RayFunction &function,  
const std::vector<std::unique\_ptr<TaskArg>> &args,  
const ActorCreationOptions &actor\_creation\_options,  
std::vector<rpc::ObjectReference> \*returned\_refs) {  
// Step 1: Generate unique actor ID  
const ActorID actor\_id = ActorID::FromRandom();  
// Step 2: Build actor creation task spec  
TaskSpecBuilder builder;  
builder.SetActorCreationTask(  
actor\_id, function, args,  
actor\_creation\_options.max\_restarts,  
actor\_creation\_options.resources);  
const TaskSpec task\_spec = builder.Build();  
// Step 3: Register with actor manager for tracking  
actor\_manager\_->RegisterActorHandle(actor\_id, task\_spec);  
// Step 4: Submit to GCS for global scheduling  
return gcs\_client\_->actor\_accessor\_->AsyncCreateActor(task\_spec);  
}

// From src/ray/core\_worker/core\_worker.cc  
Status CoreWorker::CreateActor(const RayFunction &function,  
const std::vector<std::unique\_ptr<TaskArg>> &args,  
const ActorCreationOptions &actor\_creation\_options,  
std::vector<rpc::ObjectReference> \*returned\_refs) {  
// Step 1: Generate unique actor ID  
const ActorID actor\_id = ActorID::FromRandom();  
// Step 2: Build actor creation task spec  
TaskSpecBuilder builder;  
builder.SetActorCreationTask(  
actor\_id, function, args,  
actor\_creation\_options.max\_restarts,  
actor\_creation\_options.resources);  
const TaskSpec task\_spec = builder.Build();  
// Step 3: Register with actor manager for tracking  
actor\_manager\_->RegisterActorHandle(actor\_id, task\_spec);  
// Step 4: Submit to GCS for global scheduling  
return gcs\_client\_->actor\_accessor\_->AsyncCreateActor(task\_spec);  
}

## Method Invocation and Execution

### Method Call Flow

When you call a method on an actor handle, a sophisticated routing and execution process occurs:  
📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

### Method Execution Engine

Inside the actor worker, methods are executed by a specialized runtime:

class ActorMethodExecutor:  
def \_\_init\_\_(self, actor\_instance):  
self.actor\_instance = actor\_instance  
self.method\_queue = queue.Queue()  
def \_execute\_methods(self):  
"""Main execution loop for actor methods"""  
while True:  
try:  
# Get next method call  
method\_call = self.method\_queue.get()  
if method\_call is None: # Shutdown signal  
break  
# Extract method info  
method\_name = method\_call.function\_name  
args = method\_call.args  
kwargs = method\_call.kwargs  
method = getattr(self.actor\_instance, method\_name)  
result = method(\*args, \*\*kwargs)  
# Store result in object store  
self.\_store\_result(method\_call.task\_id, result)  
except Exception as e:  
self.\_store\_error(method\_call.task\_id, e)

class ActorMethodExecutor:  
def \_\_init\_\_(self, actor\_instance):  
self.actor\_instance = actor\_instance  
self.method\_queue = queue.Queue()  
def \_execute\_methods(self):  
"""Main execution loop for actor methods"""  
while True:  
try:  
# Get next method call  
method\_call = self.method\_queue.get()  
if method\_call is None: # Shutdown signal  
break  
# Extract method info  
method\_name = method\_call.function\_name  
args = method\_call.args  
kwargs = method\_call.kwargs  
method = getattr(self.actor\_instance, method\_name)  
result = method(\*args, \*\*kwargs)  
# Store result in object store  
self.\_store\_result(method\_call.task\_id, result)  
except Exception as e:  
self.\_store\_error(method\_call.task\_id, e)

## Fault Tolerance and Recovery

### Actor Restart Policies

Ray provides sophisticated fault tolerance mechanisms for actors:

# Different restart policies  
@ray.remote(max\_restarts=3, max\_task\_retries=2)  
class FaultTolerantActor:  
def \_\_init\_\_(self):  
self.state = {"counter": 0, "last\_update": time.time()}  
def increment(self):  
self.state["counter"] += 1  
self.state["last\_update"] = time.time()  
# Simulate occasional failures  
if random.random() < 0.1:  
raise Exception("Simulated failure")  
return self.state["counter"]

# Different restart policies  
@ray.remote(max\_restarts=3, max\_task\_retries=2)  
class FaultTolerantActor:  
def \_\_init\_\_(self):  
self.state = {"counter": 0, "last\_update": time.time()}  
def increment(self):  
self.state["counter"] += 1  
self.state["last\_update"] = time.time()  
# Simulate occasional failures  
if random.random() < 0.1:  
raise Exception("Simulated failure")  
return self.state["counter"]

### Failure Detection and Recovery

📊 SEQUENCE DIAGRAM: Process Flow and Interactions  
This comprehensive guide covers the fundamental aspects of Ray's actor system. Actors provide a powerful abstraction for building stateful, distributed applications with strong consistency guarantees and fault tolerance features.

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

# Chapter 5: Memory and Object Reference System

## Introduction

Ray's memory and object reference system is like having a distributed, shared memory across your entire cluster. Instead of copying data between machines, Ray creates smart "pointers" (ObjectRefs) that can reference data stored anywhere in the cluster. This enables efficient sharing of large datasets and computation results.

### What Makes Ray's Memory System Special?

Zero-Copy Data Sharing: Large objects are stored once and referenced many times without copying.  
Automatic Garbage Collection: Objects are cleaned up automatically when no longer needed.  
Location Transparency: Your code doesn't need to know where data is physically stored.  
Fault Tolerance: Objects can be reconstructed if they're lost due to node failures.

## Architecture Overview

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

## Object References (ObjectRefs)

### What is an ObjectRef?

An ObjectRef is like a "smart pointer" that references data stored somewhere in your Ray cluster:

import ray  
import numpy as np  
# Create a large dataset  
large\_array = np.random.random((1000000, 100))  
object\_ref = ray.put(large\_array)  
print(f"ObjectRef: {object\_ref}")  
# The actual data is stored in the cluster, not in this variable  
print(f"ObjectRef size in memory: {sys.getsizeof(object\_ref)} bytes")  
# Retrieve the data when needed  
retrieved\_array = ray.get(object\_ref)  
print(f"Retrieved array shape: {retrieved\_array.shape}")  
# Output: Retrieved array shape: (1000000, 100)

import ray  
import numpy as np  
# Create a large dataset  
large\_array = np.random.random((1000000, 100))  
object\_ref = ray.put(large\_array)  
print(f"ObjectRef: {object\_ref}")  
# The actual data is stored in the cluster, not in this variable  
print(f"ObjectRef size in memory: {sys.getsizeof(object\_ref)} bytes")  
# Retrieve the data when needed  
retrieved\_array = ray.get(object\_ref)  
print(f"Retrieved array shape: {retrieved\_array.shape}")  
# Output: Retrieved array shape: (1000000, 100)

### ObjectRef Lifecycle

📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

### Automatic Object Creation

Objects are automatically stored when returned from remote functions:

@ray.remote  
def create\_large\_dataset():  
return np.random.random((100000, 1000))  
@ray.remote  
def process\_dataset(data):  
# Process the data  
return np.mean(data, axis=0)  
dataset\_ref = create\_large\_dataset.remote() # Returns ObjectRef immediately  
result\_ref = process\_dataset.remote(dataset\_ref)  
# Only retrieve when final result is needed  
final\_result = ray.get(result\_ref)

@ray.remote  
def create\_large\_dataset():  
return np.random.random((100000, 1000))  
@ray.remote  
def process\_dataset(data):  
# Process the data  
return np.mean(data, axis=0)  
dataset\_ref = create\_large\_dataset.remote() # Returns ObjectRef immediately  
result\_ref = process\_dataset.remote(dataset\_ref)  
# Only retrieve when final result is needed  
final\_result = ray.get(result\_ref)

## Distributed Object Store

### Plasma Object Store

Ray uses Apache Plasma for high-performance object storage:

// From src/ray/object\_store/plasma/client.h  
class PlasmaClient {  
public:  
/// Store an object in the plasma store  
Status Put(const ObjectID &object\_id,  
const uint8\_t \*data,  
size\_t data\_size,  
const uint8\_t \*metadata = nullptr,  
size\_t metadata\_size = 0) {  
// Step 1: Create plasma object buffer  
std::shared\_ptr<Buffer> buffer;  
Status create\_status = Create(object\_id, data\_size, &buffer);  
if (!create\_status.ok()) {  
return create\_status;  
}  
// Step 2: Copy data into shared memory  
std::memcpy(buffer->mutable\_data(), data, data\_size);  
// Step 3: Seal object (make it immutable and available)  
return Seal(object\_id);  
}  
/// Get objects from the plasma store  
Status Get(const std::vector<ObjectID> &object\_ids,  
int64\_t timeout\_ms,  
std::vector<ObjectBuffer> \*object\_buffers) {  
// Wait for objects to become available  
return impl\_->Wait(object\_ids, timeout\_ms, object\_buffers);  
}  
};

// From src/ray/object\_store/plasma/client.h  
class PlasmaClient {  
public:  
/// Store an object in the plasma store  
Status Put(const ObjectID &object\_id,  
const uint8\_t \*data,  
size\_t data\_size,  
const uint8\_t \*metadata = nullptr,  
size\_t metadata\_size = 0) {  
// Step 1: Create plasma object buffer  
std::shared\_ptr<Buffer> buffer;  
Status create\_status = Create(object\_id, data\_size, &buffer);  
if (!create\_status.ok()) {  
return create\_status;  
}  
// Step 2: Copy data into shared memory  
std::memcpy(buffer->mutable\_data(), data, data\_size);  
// Step 3: Seal object (make it immutable and available)  
return Seal(object\_id);  
}  
/// Get objects from the plasma store  
Status Get(const std::vector<ObjectID> &object\_ids,  
int64\_t timeout\_ms,  
std::vector<ObjectBuffer> \*object\_buffers) {  
// Wait for objects to become available  
return impl\_->Wait(object\_ids, timeout\_ms, object\_buffers);  
}  
};

### Multi-Node Object Access

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

## Memory Management Patterns

### Efficient Data Sharing

# Example: Sharing large datasets efficiently  
@ray.remote  
def load\_model():  
"""Load a large ML model once"""  
import joblib  
model = joblib.load('large\_model.pkl') # 2GB model  
return model  
@ray.remote  
def predict\_batch(model\_ref, data\_batch):  
"""Use shared model for prediction"""  
model = ray.get(model\_ref) # Gets reference, not copy  
return model.predict(data\_batch)  
# Load model once, share across many tasks  
model\_ref = load\_model.remote()  
# All tasks share the same model (no copying!)  
predictions = []  
for batch in data\_batches:  
pred\_ref = predict\_batch.remote(model\_ref, batch)  
predictions.append(pred\_ref)  
results = ray.get(predictions)

# Example: Sharing large datasets efficiently  
@ray.remote  
def load\_model():  
"""Load a large ML model once"""  
import joblib  
model = joblib.load('large\_model.pkl') # 2GB model  
return model  
@ray.remote  
def predict\_batch(model\_ref, data\_batch):  
"""Use shared model for prediction"""  
model = ray.get(model\_ref) # Gets reference, not copy  
return model.predict(data\_batch)  
# Load model once, share across many tasks  
model\_ref = load\_model.remote()  
# All tasks share the same model (no copying!)  
predictions = []  
for batch in data\_batches:  
pred\_ref = predict\_batch.remote(model\_ref, batch)  
predictions.append(pred\_ref)  
results = ray.get(predictions)

### Memory-Efficient Processing

@ray.remote  
def process\_chunk(data\_chunk):  
"""Process a chunk of data"""  
processed = expensive\_computation(data\_chunk)  
return summarize(processed) # Return summary, not full data  
# Split large dataset into chunks  
large\_dataset = load\_huge\_dataset() # 100GB dataset  
chunk\_size = len(large\_dataset) // num\_workers  
chunk\_refs = []  
for i in range(0, len(large\_dataset), chunk\_size):  
chunk = large\_dataset[i:i + chunk\_size]  
chunk\_ref = ray.put(chunk) # Store chunk in object store  
chunk\_refs.append(chunk\_ref)  
# Process chunks in parallel  
result\_refs = [process\_chunk.remote(chunk\_ref) for chunk\_ref in chunk\_refs]  
# Combine results (much smaller than original data)  
results = ray.get(result\_refs)  
final\_result = combine\_results(results)

@ray.remote  
def process\_chunk(data\_chunk):  
"""Process a chunk of data"""  
processed = expensive\_computation(data\_chunk)  
return summarize(processed) # Return summary, not full data  
# Split large dataset into chunks  
large\_dataset = load\_huge\_dataset() # 100GB dataset  
chunk\_size = len(large\_dataset) // num\_workers  
chunk\_refs = []  
for i in range(0, len(large\_dataset), chunk\_size):  
chunk = large\_dataset[i:i + chunk\_size]  
chunk\_ref = ray.put(chunk) # Store chunk in object store  
chunk\_refs.append(chunk\_ref)  
# Process chunks in parallel  
result\_refs = [process\_chunk.remote(chunk\_ref) for chunk\_ref in chunk\_refs]  
# Combine results (much smaller than original data)  
results = ray.get(result\_refs)  
final\_result = combine\_results(results)

## Reference Counting and Garbage Collection

### Automatic Cleanup

Ray automatically cleans up objects when they're no longer needed:

// From src/ray/core\_worker/reference\_count.h  
class ReferenceCounter {  
private:  
// Track reference counts for each object  
absl::flat\_hash\_map<ObjectID, int> object\_ref\_counts\_;  
// Track which worker owns each object  
absl::flat\_hash\_map<ObjectID, WorkerID> object\_owners\_;  
public:  
/// Add a reference to an object  
void AddObjectRef(const ObjectID &object\_id, const WorkerID &owner\_id) {  
object\_ref\_counts\_[object\_id]++;  
object\_owners\_[object\_id] = owner\_id;  
}  
/// Remove a reference to an object  
void RemoveObjectRef(const ObjectID &object\_id) {  
auto it = object\_ref\_counts\_.find(object\_id);  
if (it != object\_ref\_counts\_.end()) {  
it->second--;  
if (it->second == 0) {  
// No more references - schedule for deletion  
ScheduleObjectDeletion(object\_id);  
object\_ref\_counts\_.erase(it);  
object\_owners\_.erase(object\_id);  
}  
}  
}  
private:  
void ScheduleObjectDeletion(const ObjectID &object\_id) {  
// Send deletion request to object store  
deletion\_queue\_.push(object\_id);  
}  
};

// From src/ray/core\_worker/reference\_count.h  
class ReferenceCounter {  
private:  
// Track reference counts for each object  
absl::flat\_hash\_map<ObjectID, int> object\_ref\_counts\_;  
// Track which worker owns each object  
absl::flat\_hash\_map<ObjectID, WorkerID> object\_owners\_;  
public:  
/// Add a reference to an object  
void AddObjectRef(const ObjectID &object\_id, const WorkerID &owner\_id) {  
object\_ref\_counts\_[object\_id]++;  
object\_owners\_[object\_id] = owner\_id;  
}  
/// Remove a reference to an object  
void RemoveObjectRef(const ObjectID &object\_id) {  
auto it = object\_ref\_counts\_.find(object\_id);  
if (it != object\_ref\_counts\_.end()) {  
it->second--;  
if (it->second == 0) {  
// No more references - schedule for deletion  
ScheduleObjectDeletion(object\_id);  
object\_ref\_counts\_.erase(it);  
object\_owners\_.erase(object\_id);  
}  
}  
}  
private:  
void ScheduleObjectDeletion(const ObjectID &object\_id) {  
// Send deletion request to object store  
deletion\_queue\_.push(object\_id);  
}  
};

### Manual Memory Management

You can also manually control object lifecycle:

import ray  
data = ray.put(large\_dataset)  
result = process\_data.remote(data)  
final\_result = ray.get(result)  
# Manually delete when done (optional - Ray will do this automatically)  
del data # Remove reference  
ray.internal.free([data]) # Force cleanup

import ray  
data = ray.put(large\_dataset)  
result = process\_data.remote(data)  
final\_result = ray.get(result)  
# Manually delete when done (optional - Ray will do this automatically)  
del data # Remove reference  
ray.internal.free([data]) # Force cleanup

## Object Reconstruction and Fault Tolerance

### Lineage-Based Recovery

Ray can reconstruct lost objects using lineage information:

# Example: Fault-tolerant computation chain  
@ray.remote  
def step1():  
return expensive\_computation\_1()  
@ray.remote  
def step2(data1):  
return expensive\_computation\_2(data1)  
@ray.remote  
def step3(data2):  
return expensive\_computation\_3(data2)  
# Build computation chain  
result1 = step1.remote()  
result2 = step2.remote(result1)  
result3 = step3.remote(result2)  
# If any intermediate result is lost due to node failure,  
# Ray can reconstruct it by re-running the necessary tasks  
final\_result = ray.get(result3) # Handles reconstruction transparently

# Example: Fault-tolerant computation chain  
@ray.remote  
def step1():  
return expensive\_computation\_1()  
@ray.remote  
def step2(data1):  
return expensive\_computation\_2(data1)  
@ray.remote  
def step3(data2):  
return expensive\_computation\_3(data2)  
# Build computation chain  
result1 = step1.remote()  
result2 = step2.remote(result1)  
result3 = step3.remote(result2)  
# If any intermediate result is lost due to node failure,  
# Ray can reconstruct it by re-running the necessary tasks  
final\_result = ray.get(result3) # Handles reconstruction transparently

### Reconstruction Process

📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

## Performance Optimization

### Object Store Memory Management

ray.init(object\_store\_memory=8\_000\_000\_000) # 8GB for object store  
print(ray.cluster\_resources())  
import psutil  
object\_store\_memory = ray.\_private.utils.get\_system\_memory() // 2  
print(f"Object store memory limit: {object\_store\_memory / 1e9:.1f} GB")

ray.init(object\_store\_memory=8\_000\_000\_000) # 8GB for object store  
print(ray.cluster\_resources())  
import psutil  
object\_store\_memory = ray.\_private.utils.get\_system\_memory() // 2  
print(f"Object store memory limit: {object\_store\_memory / 1e9:.1f} GB")

### Best Practices

large\_model = load\_model()  
model\_ref = ray.put(large\_model) # Store once  
results = [predict.remote(model\_ref, batch) for batch in batches]  
@ray.remote  
def process\_large\_data(big\_data\_ref):  
big\_data = ray.get(big\_data\_ref)  
result = expensive\_processing(big\_data)  
return summarize(result) # Return summary, not full result  
@ray.remote  
def pipeline\_step1(data):  
return process\_step1(data)  
@ray.remote  
def pipeline\_step2(step1\_result\_ref):  
step1\_result = ray.get(step1\_result\_ref)  
return process\_step2(step1\_result)

large\_model = load\_model()  
model\_ref = ray.put(large\_model) # Store once  
results = [predict.remote(model\_ref, batch) for batch in batches]  
@ray.remote  
def process\_large\_data(big\_data\_ref):  
big\_data = ray.get(big\_data\_ref)  
result = expensive\_processing(big\_data)  
return summarize(result) # Return summary, not full result  
@ray.remote  
def pipeline\_step1(data):  
return process\_step1(data)  
@ray.remote  
def pipeline\_step2(step1\_result\_ref):  
step1\_result = ray.get(step1\_result\_ref)  
return process\_step2(step1\_result)

This comprehensive guide covers Ray's sophisticated memory management system that enables efficient distributed computing with automatic garbage collection and fault tolerance.

# Chapter 6: Global Control Service (GCS)

# Ray GCS Server: Comprehensive Technical Guide

## Table of Contents

Introduction

Architecture Overview

Core Components

Node Lifecycle Management

Resource Management

Actor Management

Job Management

Storage and Persistence

Communication and RPC

Fault Tolerance and Recovery

Performance Characteristics

Implementation Details

Code Modification Guidelines

## Introduction

The GCS (Global Control Service) server is the central coordination hub of a Ray cluster. It maintains authoritative global state about all cluster resources, nodes, actors, jobs, and placement groups. The GCS serves as the single source of truth for cluster-wide metadata and coordinates distributed operations across the entire Ray cluster.

### Key Responsibilities

Node Registration and Health Monitoring: Track all nodes joining/leaving the cluster

Resource Management: Coordinate cluster-wide resource allocation and scheduling

Actor Management: Handle actor creation, placement, and lifecycle

Job Coordination: Manage job submission, tracking, and cleanup

Metadata Storage: Persist critical cluster state and configuration

Service Discovery: Provide endpoints for cluster services

## Architecture Overview

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### GCS Server Design Principles

Single Source of Truth: All authoritative cluster state lives in GCS

Event-Driven Architecture: State changes trigger cascading updates

Scalable Storage: Pluggable backend storage (Redis, Memory)

Fault Recovery: Persistent state enables cluster recovery

Performance Optimization: Caching and batching for high throughput

## Core Components

The GCS server consists of several specialized managers working together:

### Component Initialization Order

From src/ray/gcs/gcs\_server/gcs\_server.h:140-180:  
📊 SEQUENCE DIAGRAM: Process Flow and Interactions

src/ray/gcs/gcs\_server/gcs\_server.h:140-180

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

### GCS Server Configuration

From src/ray/gcs/gcs\_server/gcs\_server.h:47-62:

src/ray/gcs/gcs\_server/gcs\_server.h:47-62

struct GcsServerConfig {  
std::string grpc\_server\_name = "GcsServer";  
uint16\_t grpc\_server\_port = 0; // GCS RPC port  
uint16\_t grpc\_server\_thread\_num = 1; // RPC thread pool size  
std::string redis\_username; // Redis authentication  
std::string redis\_password;  
std::string redis\_address; // Redis host address  
uint16\_t redis\_port = 6379; // Redis port  
bool enable\_redis\_ssl = false; // TLS encryption  
bool retry\_redis = true; // Connection retry logic  
bool enable\_sharding\_conn = false; // Redis sharding  
std::string node\_ip\_address; // GCS server IP  
std::string log\_dir; // Logging directory  
std::string raylet\_config\_list; // Raylet configurations  
std::string session\_name; // Cluster session ID  
};

struct GcsServerConfig {  
std::string grpc\_server\_name = "GcsServer";  
uint16\_t grpc\_server\_port = 0; // GCS RPC port  
uint16\_t grpc\_server\_thread\_num = 1; // RPC thread pool size  
std::string redis\_username; // Redis authentication  
std::string redis\_password;  
std::string redis\_address; // Redis host address  
uint16\_t redis\_port = 6379; // Redis port  
bool enable\_redis\_ssl = false; // TLS encryption  
bool retry\_redis = true; // Connection retry logic  
bool enable\_sharding\_conn = false; // Redis sharding  
std::string node\_ip\_address; // GCS server IP  
std::string log\_dir; // Logging directory  
std::string raylet\_config\_list; // Raylet configurations  
std::string session\_name; // Cluster session ID  
};

## Node Lifecycle Management

The GCS Node Manager is responsible for tracking all nodes in the cluster and their health status.

### Node State Machine

🔧 TECHNICAL DIAGRAM: System Architecture

[DIAGRAM: 🔧 TECHNICAL DIAGRAM: System Architecture]

### Node Registration Protocol

From src/ray/gcs/gcs\_server/gcs\_node\_manager.h:54-62:  
📊 SEQUENCE DIAGRAM: Process Flow and Interactions  
Node Information Structure:

src/ray/gcs/gcs\_server/gcs\_node\_manager.h:54-62

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

// From gcs.proto - rpc::GcsNodeInfo  
message GcsNodeInfo {  
bytes node\_id = 1; // Unique node identifier  
string node\_manager\_address = 2; // Node IP address  
int32 node\_manager\_port = 3; // Node manager port  
int32 object\_manager\_port = 4; // Object manager port  
string node\_name = 5; // Human-readable name  
map<string, double> resources\_total = 6; // Total node resources  
GcsNodeState state = 7; // Current node state  
NodeDeathInfo death\_info = 8; // Death information if dead  
int64 start\_time\_ms = 9; // Node startup timestamp  
}  
enum GcsNodeState {  
ALIVE = 0; // Node operational  
DEAD = 1; // Node failed/removed  
DRAINING = 2; // Node shutting down gracefully  
}

// From gcs.proto - rpc::GcsNodeInfo  
message GcsNodeInfo {  
bytes node\_id = 1; // Unique node identifier  
string node\_manager\_address = 2; // Node IP address  
int32 node\_manager\_port = 3; // Node manager port  
int32 object\_manager\_port = 4; // Object manager port  
string node\_name = 5; // Human-readable name  
map<string, double> resources\_total = 6; // Total node resources  
GcsNodeState state = 7; // Current node state  
NodeDeathInfo death\_info = 8; // Death information if dead  
int64 start\_time\_ms = 9; // Node startup timestamp  
}  
enum GcsNodeState {  
ALIVE = 0; // Node operational  
DEAD = 1; // Node failed/removed  
DRAINING = 2; // Node shutting down gracefully  
}

### Health Monitoring and Failure Detection

Health Check Mechanisms:  
1. Periodic Heartbeats: Raylets send regular health updates  
2. Resource Reports: Nodes report resource usage changes  
3. Task Status Updates: Monitor task execution health  
4. Network Connectivity: Detect network partitions  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

## Resource Management

The GCS Resource Manager maintains a global view of all cluster resources and coordinates scheduling decisions.

### Resource Architecture

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Resource Types and Management

Core Resource Types:

// Resource categories managed by GCS  
enum ResourceType {  
CPU, // Compute cores  
GPU, // Graphics processors  
MEMORY, // RAM allocation  
OBJECT\_STORE\_MEMORY, // Plasma store memory  
CUSTOM // User-defined resources  
};  
// Resource scheduling information  
struct ResourceSchedulingState {  
map<string, double> total; // Total available resources  
map<string, double> available; // Currently available resources  
map<string, double> used; // Currently used resources  
vector<TaskSpec> pending\_tasks; // Tasks waiting for resources  
};

// Resource categories managed by GCS  
enum ResourceType {  
CPU, // Compute cores  
GPU, // Graphics processors  
MEMORY, // RAM allocation  
OBJECT\_STORE\_MEMORY, // Plasma store memory  
CUSTOM // User-defined resources  
};  
// Resource scheduling information  
struct ResourceSchedulingState {  
map<string, double> total; // Total available resources  
map<string, double> available; // Currently available resources  
map<string, double> used; // Currently used resources  
vector<TaskSpec> pending\_tasks; // Tasks waiting for resources  
};

### Resource Synchronization Protocol

📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

## Actor Management

The GCS Actor Manager handles the distributed coordination of Ray actors, including creation, placement, and lifecycle management.

### Actor Lifecycle Management

🔧 TECHNICAL DIAGRAM: System Architecture

[DIAGRAM: 🔧 TECHNICAL DIAGRAM: System Architecture]

### Actor Creation Protocol

📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

### Actor Placement Strategies

Placement Group Integration:

// Actor placement within placement groups  
struct ActorPlacementSpec {  
PlacementGroupID placement\_group\_id; // Target placement group  
int bundle\_index; // Specific bundle in group  
PlacementStrategy strategy; // PACK, SPREAD, STRICT\_PACK  
map<string, double> resource\_requirements; // Resource needs  
vector<NodeID> blacklist\_nodes; // Nodes to avoid  
};

// Actor placement within placement groups  
struct ActorPlacementSpec {  
PlacementGroupID placement\_group\_id; // Target placement group  
int bundle\_index; // Specific bundle in group  
PlacementStrategy strategy; // PACK, SPREAD, STRICT\_PACK  
map<string, double> resource\_requirements; // Resource needs  
vector<NodeID> blacklist\_nodes; // Nodes to avoid  
};

## Job Management

The GCS Job Manager coordinates job submission, tracking, and resource cleanup across the cluster.

### Job Lifecycle Architecture

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Job State Management

// Job states tracked by GCS  
enum JobState {  
PENDING = 0; // Job submitted, awaiting resources  
RUNNING = 1; // Job executing tasks  
STOPPED = 2; // Job terminated normally  
FAILED = 3; // Job failed due to error  
};  
// Job information maintained by GCS  
struct JobInfo {  
JobID job\_id; // Unique job identifier  
JobState state; // Current job state  
string driver\_ip\_address; // Driver node IP  
int64\_t driver\_pid; // Driver process ID  
int64\_t start\_time; // Job start timestamp  
int64\_t end\_time; // Job end timestamp (if finished)  
map<string, double> resource\_mapping; // Allocated resources  
JobConfig config; // Job configuration  
};

// Job states tracked by GCS  
enum JobState {  
PENDING = 0; // Job submitted, awaiting resources  
RUNNING = 1; // Job executing tasks  
STOPPED = 2; // Job terminated normally  
FAILED = 3; // Job failed due to error  
};  
// Job information maintained by GCS  
struct JobInfo {  
JobID job\_id; // Unique job identifier  
JobState state; // Current job state  
string driver\_ip\_address; // Driver node IP  
int64\_t driver\_pid; // Driver process ID  
int64\_t start\_time; // Job start timestamp  
int64\_t end\_time; // Job end timestamp (if finished)  
map<string, double> resource\_mapping; // Allocated resources  
JobConfig config; // Job configuration  
};

## Storage and Persistence

The GCS uses pluggable storage backends to persist critical cluster state and enable recovery.

### Storage Architecture

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Storage Configuration Options

From src/ray/gcs/gcs\_server/gcs\_server.h:98-104:

src/ray/gcs/gcs\_server/gcs\_server.h:98-104

enum class StorageType {  
UNKNOWN = 0,  
IN\_MEMORY = 1, // Fast, non-persistent storage  
REDIS\_PERSIST = 2, // Persistent Redis storage  
};  
// Storage configuration constants  
static constexpr char kInMemoryStorage[] = "memory";  
static constexpr char kRedisStorage[] = "redis";

enum class StorageType {  
UNKNOWN = 0,  
IN\_MEMORY = 1, // Fast, non-persistent storage  
REDIS\_PERSIST = 2, // Persistent Redis storage  
};  
// Storage configuration constants  
static constexpr char kInMemoryStorage[] = "memory";  
static constexpr char kRedisStorage[] = "redis";

Storage Type Selection:  
| Storage Type | Use Case | Persistence | Performance | Fault Tolerance |  
|-------------|----------|-------------|-------------|-----------------|  
| Memory | Development/Testing | No | Highest | None |  
| Redis | Production | Yes | High | Full recovery |  
| File | Local debugging | Yes | Medium | Local only |

### Data Persistence Patterns

Critical Data Categories:  
1. Node Registry: All registered nodes and their states  
2. Actor Registry: Actor metadata and placement information  
3. Job Registry: Job specifications and execution state  
4. Resource State: Cluster resource allocation and usage  
5. Configuration: Cluster and component configurations  
📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

## Communication and RPC

The GCS server provides gRPC-based APIs for all cluster components to interact with global state.

### RPC Service Architecture

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Key RPC Interfaces

Node Management RPCs:

// From gcs\_service.proto  
service NodeInfoGcsService {  
rpc RegisterNode(RegisterNodeRequest) returns (RegisterNodeReply);  
rpc UnregisterNode(UnregisterNodeRequest) returns (UnregisterNodeReply);  
rpc GetAllNodeInfo(GetAllNodeInfoRequest) returns (GetAllNodeInfoReply);  
rpc CheckAlive(CheckAliveRequest) returns (CheckAliveReply);  
rpc DrainNode(DrainNodeRequest) returns (DrainNodeReply);  
}

// From gcs\_service.proto  
service NodeInfoGcsService {  
rpc RegisterNode(RegisterNodeRequest) returns (RegisterNodeReply);  
rpc UnregisterNode(UnregisterNodeRequest) returns (UnregisterNodeReply);  
rpc GetAllNodeInfo(GetAllNodeInfoRequest) returns (GetAllNodeInfoReply);  
rpc CheckAlive(CheckAliveRequest) returns (CheckAliveReply);  
rpc DrainNode(DrainNodeRequest) returns (DrainNodeReply);  
}

Actor Management RPCs:

service ActorInfoGcsService {  
rpc CreateActor(CreateActorRequest) returns (CreateActorReply);  
rpc GetActorInfo(GetActorInfoRequest) returns (GetActorInfoReply);  
rpc KillActorViaGcs(KillActorViaGcsRequest) returns (KillActorViaGcsReply);  
rpc ListNamedActors(ListNamedActorsRequest) returns (ListNamedActorsReply);  
}

service ActorInfoGcsService {  
rpc CreateActor(CreateActorRequest) returns (CreateActorReply);  
rpc GetActorInfo(GetActorInfoRequest) returns (GetActorInfoReply);  
rpc KillActorViaGcs(KillActorViaGcsRequest) returns (KillActorViaGcsReply);  
rpc ListNamedActors(ListNamedActorsRequest) returns (ListNamedActorsReply);  
}

### Performance Optimization

RPC Performance Characteristics:  
| Operation Type | Typical Latency | Throughput | Optimization |  
|---------------|-----------------|------------|--------------|  
| Node registration | 1-5ms | 1K ops/s | Batched updates |  
| Actor creation | 5-20ms | 500 ops/s | Async processing |  
| Resource queries | < 1ms | 10K ops/s | Local caching |  
| Job submission | 2-10ms | 1K ops/s | Pipeline processing |

## Fault Tolerance and Recovery

The GCS implements comprehensive fault tolerance mechanisms to ensure cluster resilience.

### Recovery Architecture

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### GCS Server Recovery Process

📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

### Recovery Scenarios

1. GCS Server Crash:  
- Persistent storage preserves critical state  
- New GCS instance loads saved data  
- Nodes re-register and update status  
- Clients reconnect automatically  
2. Storage Backend Failure:  
- GCS switches to backup storage  
- In-memory state provides temporary continuity  
- Storage recovery restores full persistence  
3. Network Partition:  
- GCS maintains authoritative state  
- Nodes operate in degraded mode  
- State synchronization on partition heal

## Performance Characteristics

### Scalability Metrics

GCS Server Performance:  
| Metric | Small Cluster (10 nodes) | Medium Cluster (100 nodes) | Large Cluster (1000 nodes) |  
|--------|---------------------------|-----------------------------|-----------------------------|  
| Node registration throughput | 100 ops/s | 500 ops/s | 1K ops/s |  
| Actor creation latency | 5ms | 10ms | 20ms |  
| Resource query latency | 0.5ms | 1ms | 2ms |  
| Memory usage | 100MB | 500MB | 2GB |  
| Storage size | 10MB | 100MB | 1GB |

### Optimization Strategies

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

## Implementation Details

### Core Code Structure

GCS Server Main Loop:  
From src/ray/gcs/gcs\_server/gcs\_server\_main.cc:45-190:

src/ray/gcs/gcs\_server/gcs\_server\_main.cc:45-190

int main(int argc, char \*argv[]) {  
// Parse command line arguments  
gflags::ParseCommandLineFlags(&argc, &argv, true);  
// Configure logging and stream redirection  
InitShutdownRAII ray\_log\_shutdown\_raii(/\*...\*/);  
// Initialize configuration  
RayConfig::instance().initialize(config\_list);  
// Create main IO service  
instrumented\_io\_context main\_service(/\*enable\_lag\_probe=\*/true);  
// Initialize metrics collection  
ray::stats::Init(global\_tags, metrics\_agent\_port, WorkerID::Nil());  
// Create and configure GCS server  
ray::gcs::GcsServerConfig gcs\_server\_config;  
ray::gcs::GcsServer gcs\_server(gcs\_server\_config, main\_service);  
// Set up signal handlers for graceful shutdown  
boost::asio::signal\_set signals(main\_service);  
signals.async\_wait(shutdown\_handler);  
// Start the server and run main loop  
gcs\_server.Start();  
main\_service.run();  
}

int main(int argc, char \*argv[]) {  
// Parse command line arguments  
gflags::ParseCommandLineFlags(&argc, &argv, true);  
// Configure logging and stream redirection  
InitShutdownRAII ray\_log\_shutdown\_raii(/\*...\*/);  
// Initialize configuration  
RayConfig::instance().initialize(config\_list);  
// Create main IO service  
instrumented\_io\_context main\_service(/\*enable\_lag\_probe=\*/true);  
// Initialize metrics collection  
ray::stats::Init(global\_tags, metrics\_agent\_port, WorkerID::Nil());  
// Create and configure GCS server  
ray::gcs::GcsServerConfig gcs\_server\_config;  
ray::gcs::GcsServer gcs\_server(gcs\_server\_config, main\_service);  
// Set up signal handlers for graceful shutdown  
boost::asio::signal\_set signals(main\_service);  
signals.async\_wait(shutdown\_handler);  
// Start the server and run main loop  
gcs\_server.Start();  
main\_service.run();  
}

Component Initialization Pattern:

class GcsServer {  
void DoStart(const GcsInitData &gcs\_init\_data) {  
// Initialize storage backend first  
gcs\_table\_storage\_ = CreateStorage();  
// Initialize core managers  
InitGcsNodeManager(gcs\_init\_data);  
InitGcsResourceManager(gcs\_init\_data);  
InitGcsJobManager(gcs\_init\_data);  
InitGcsActorManager(gcs\_init\_data);  
InitGcsPlacementGroupManager(gcs\_init\_data);  
// Initialize supporting services  
InitKVManager();  
InitPubSubHandler();  
InitRuntimeEnvManager();  
// Install cross-component event listeners  
InstallEventListeners();  
// Start RPC server  
rpc\_server\_.Run();  
}  
};

class GcsServer {  
void DoStart(const GcsInitData &gcs\_init\_data) {  
// Initialize storage backend first  
gcs\_table\_storage\_ = CreateStorage();  
// Initialize core managers  
InitGcsNodeManager(gcs\_init\_data);  
InitGcsResourceManager(gcs\_init\_data);  
InitGcsJobManager(gcs\_init\_data);  
InitGcsActorManager(gcs\_init\_data);  
InitGcsPlacementGroupManager(gcs\_init\_data);  
// Initialize supporting services  
InitKVManager();  
InitPubSubHandler();  
InitRuntimeEnvManager();  
// Install cross-component event listeners  
InstallEventListeners();  
// Start RPC server  
rpc\_server\_.Run();  
}  
};

### Critical Code Paths

Node Registration Handler:

void GcsNodeManager::HandleRegisterNode(  
rpc::RegisterNodeRequest request,  
rpc::RegisterNodeReply \*reply,  
rpc::SendReplyCallback send\_reply\_callback) {  
NodeID node\_id = NodeID::FromBinary(request.node\_info().node\_id());  
// Create node info from request  
auto node = std::make\_shared<rpc::GcsNodeInfo>(request.node\_info());  
// Add to alive nodes and storage  
AddNode(node);  
// Publish node added event  
RAY\_CHECK\_OK(gcs\_publisher\_->PublishNodeInfo(node\_id, \*node, nullptr));  
// Notify listeners  
for (auto &listener : node\_added\_listeners\_) {  
listener(node);  
}  
send\_reply\_callback(Status::OK(), nullptr, nullptr);  
}

void GcsNodeManager::HandleRegisterNode(  
rpc::RegisterNodeRequest request,  
rpc::RegisterNodeReply \*reply,  
rpc::SendReplyCallback send\_reply\_callback) {  
NodeID node\_id = NodeID::FromBinary(request.node\_info().node\_id());  
// Create node info from request  
auto node = std::make\_shared<rpc::GcsNodeInfo>(request.node\_info());  
// Add to alive nodes and storage  
AddNode(node);  
// Publish node added event  
RAY\_CHECK\_OK(gcs\_publisher\_->PublishNodeInfo(node\_id, \*node, nullptr));  
// Notify listeners  
for (auto &listener : node\_added\_listeners\_) {  
listener(node);  
}  
send\_reply\_callback(Status::OK(), nullptr, nullptr);  
}

### Error Handling Patterns

Graceful Degradation:

// Example error handling in resource management  
Status GcsResourceManager::UpdateResourceUsage(const NodeID &node\_id,  
const ResourceUsageMap &usage) {  
// Try to update local state first  
auto status = UpdateLocalResourceView(node\_id, usage);  
if (!status.ok()) {  
RAY\_LOG(WARNING) << "Failed to update local resource view: " << status;  
// Continue with degraded functionality  
}  
// Try to persist to storage  
status = PersistResourceUsage(node\_id, usage);  
if (!status.ok()) {  
RAY\_LOG(ERROR) << "Failed to persist resource usage: " << status;  
// Queue for retry  
retry\_queue\_.push({node\_id, usage});  
}  
return Status::OK(); // Always succeed for availability  
}

// Example error handling in resource management  
Status GcsResourceManager::UpdateResourceUsage(const NodeID &node\_id,  
const ResourceUsageMap &usage) {  
// Try to update local state first  
auto status = UpdateLocalResourceView(node\_id, usage);  
if (!status.ok()) {  
RAY\_LOG(WARNING) << "Failed to update local resource view: " << status;  
// Continue with degraded functionality  
}  
// Try to persist to storage  
status = PersistResourceUsage(node\_id, usage);  
if (!status.ok()) {  
RAY\_LOG(ERROR) << "Failed to persist resource usage: " << status;  
// Queue for retry  
retry\_queue\_.push({node\_id, usage});  
}  
return Status::OK(); // Always succeed for availability  
}

## Code Modification Guidelines

### Adding New GCS Components

1. Manager Component Pattern:  
To add a new manager (e.g., GcsCustomManager):

// 1. Create header file: gcs\_custom\_manager.h  
class GcsCustomManager : public rpc::CustomServiceHandler {  
public:  
GcsCustomManager(GcsPublisher \*publisher,  
GcsTableStorage \*storage,  
instrumented\_io\_context &io\_context);  
// Implement RPC handlers  
void HandleCustomRequest(rpc::CustomRequest request,  
rpc::CustomReply \*reply,  
rpc::SendReplyCallback callback) override;  
// Initialize from persistent data  
void Initialize(const GcsInitData &init\_data);  
private:  
GcsPublisher \*gcs\_publisher\_;  
GcsTableStorage \*gcs\_table\_storage\_;  
// Component-specific state  
};  
// 2. Add to GcsServer initialization  
void GcsServer::InitGcsCustomManager(const GcsInitData &init\_data) {  
gcs\_custom\_manager\_ = std::make\_unique<GcsCustomManager>(  
gcs\_publisher\_.get(), gcs\_table\_storage\_.get(), main\_service\_);  
gcs\_custom\_manager\_->Initialize(init\_data);  
}

// 1. Create header file: gcs\_custom\_manager.h  
class GcsCustomManager : public rpc::CustomServiceHandler {  
public:  
GcsCustomManager(GcsPublisher \*publisher,  
GcsTableStorage \*storage,  
instrumented\_io\_context &io\_context);  
// Implement RPC handlers  
void HandleCustomRequest(rpc::CustomRequest request,  
rpc::CustomReply \*reply,  
rpc::SendReplyCallback callback) override;  
// Initialize from persistent data  
void Initialize(const GcsInitData &init\_data);  
private:  
GcsPublisher \*gcs\_publisher\_;  
GcsTableStorage \*gcs\_table\_storage\_;  
// Component-specific state  
};  
// 2. Add to GcsServer initialization  
void GcsServer::InitGcsCustomManager(const GcsInitData &init\_data) {  
gcs\_custom\_manager\_ = std::make\_unique<GcsCustomManager>(  
gcs\_publisher\_.get(), gcs\_table\_storage\_.get(), main\_service\_);  
gcs\_custom\_manager\_->Initialize(init\_data);  
}

2. Adding New RPC Services:

// 1. Define in protobuf (gcs\_service.proto)  
service CustomGcsService {  
rpc CustomOperation(CustomRequest) returns (CustomReply);  
}  
// 2. Register in RPC server  
void GcsServer::StartRpcServer() {  
rpc\_server\_.RegisterService(gcs\_custom\_manager\_.get());  
rpc\_server\_.Run();  
}

// 1. Define in protobuf (gcs\_service.proto)  
service CustomGcsService {  
rpc CustomOperation(CustomRequest) returns (CustomReply);  
}  
// 2. Register in RPC server  
void GcsServer::StartRpcServer() {  
rpc\_server\_.RegisterService(gcs\_custom\_manager\_.get());  
rpc\_server\_.Run();  
}

3. State Persistence Integration:

// Add to storage initialization  
void GcsCustomManager::Initialize(const GcsInitData &init\_data) {  
// Load persistent state  
auto custom\_data = gcs\_table\_storage\_->CustomTable().GetAll();  
// Rebuild in-memory state  
for (const auto &[key, value] : custom\_data) {  
RestoreCustomState(key, value);  
}  
}  
// Persist state changes  
void GcsCustomManager::PersistCustomData(const Key &key, const Value &value) {  
auto status = gcs\_table\_storage\_->CustomTable().Put(key, value, nullptr);  
if (!status.ok()) {  
RAY\_LOG(ERROR) << "Failed to persist custom data: " << status;  
}  
}

// Add to storage initialization  
void GcsCustomManager::Initialize(const GcsInitData &init\_data) {  
// Load persistent state  
auto custom\_data = gcs\_table\_storage\_->CustomTable().GetAll();  
// Rebuild in-memory state  
for (const auto &[key, value] : custom\_data) {  
RestoreCustomState(key, value);  
}  
}  
// Persist state changes  
void GcsCustomManager::PersistCustomData(const Key &key, const Value &value) {  
auto status = gcs\_table\_storage\_->CustomTable().Put(key, value, nullptr);  
if (!status.ok()) {  
RAY\_LOG(ERROR) << "Failed to persist custom data: " << status;  
}  
}

### Testing and Validation

Unit Testing Pattern:

class GcsCustomManagerTest : public ::testing::Test {  
protected:  
void SetUp() override {  
gcs\_publisher\_ = std::make\_shared<GcsPublisher>(/\*...\*/);  
store\_client\_ = std::make\_shared<MemoryStoreClient>();  
gcs\_table\_storage\_ = std::make\_shared<GcsTableStorage>(store\_client\_);  
manager\_ = std::make\_unique<GcsCustomManager>(  
gcs\_publisher\_.get(), gcs\_table\_storage\_.get(), io\_context\_);  
}  
instrumented\_io\_context io\_context\_;  
std::unique\_ptr<GcsCustomManager> manager\_;  
// Test fixtures  
};  
TEST\_F(GcsCustomManagerTest, HandleCustomRequest) {  
// Test RPC handling logic  
rpc::CustomRequest request;  
rpc::CustomReply reply;  
auto callback = [](Status status,  
std::function<void()> success,  
std::function<void()> failure) {  
EXPECT\_TRUE(status.ok());  
};  
manager\_->HandleCustomRequest(request, &reply, callback);  
}

class GcsCustomManagerTest : public ::testing::Test {  
protected:  
void SetUp() override {  
gcs\_publisher\_ = std::make\_shared<GcsPublisher>(/\*...\*/);  
store\_client\_ = std::make\_shared<MemoryStoreClient>();  
gcs\_table\_storage\_ = std::make\_shared<GcsTableStorage>(store\_client\_);  
manager\_ = std::make\_unique<GcsCustomManager>(  
gcs\_publisher\_.get(), gcs\_table\_storage\_.get(), io\_context\_);  
}  
instrumented\_io\_context io\_context\_;  
std::unique\_ptr<GcsCustomManager> manager\_;  
// Test fixtures  
};  
TEST\_F(GcsCustomManagerTest, HandleCustomRequest) {  
// Test RPC handling logic  
rpc::CustomRequest request;  
rpc::CustomReply reply;  
auto callback = [](Status status,  
std::function<void()> success,  
std::function<void()> failure) {  
EXPECT\_TRUE(status.ok());  
};  
manager\_->HandleCustomRequest(request, &reply, callback);  
}

Integration Testing:

# Test GCS server functionality  
cd /home/ssiddique/ray  
bazel test //src/ray/gcs/gcs\_server/test:gcs\_server\_test  
bazel test //src/ray/gcs/gcs\_server/test:gcs\_server\_integration\_test  
# Test specific managers  
bazel test //src/ray/gcs/gcs\_server/test:gcs\_node\_manager\_test  
bazel test //src/ray/gcs/gcs\_server/test:gcs\_actor\_manager\_test

# Test GCS server functionality  
cd /home/ssiddique/ray  
bazel test //src/ray/gcs/gcs\_server/test:gcs\_server\_test  
bazel test //src/ray/gcs/gcs\_server/test:gcs\_server\_integration\_test  
# Test specific managers  
bazel test //src/ray/gcs/gcs\_server/test:gcs\_node\_manager\_test  
bazel test //src/ray/gcs/gcs\_server/test:gcs\_actor\_manager\_test

Performance Testing:

# GCS server load testing  
import ray  
import time  
import concurrent.futures  
@ray.remote  
def stress\_test\_actor():  
return "alive"  
# Test actor creation throughput  
start\_time = time.time()  
actors = [stress\_test\_actor.remote() for \_ in range(1000)]  
results = ray.get(actors)  
end\_time = time.time()  
throughput = len(actors) / (end\_time - start\_time)  
print(f"Actor creation throughput: {throughput:.2f} actors/sec")

# GCS server load testing  
import ray  
import time  
import concurrent.futures  
@ray.remote  
def stress\_test\_actor():  
return "alive"  
# Test actor creation throughput  
start\_time = time.time()  
actors = [stress\_test\_actor.remote() for \_ in range(1000)]  
results = ray.get(actors)  
end\_time = time.time()  
throughput = len(actors) / (end\_time - start\_time)  
print(f"Actor creation throughput: {throughput:.2f} actors/sec")

This comprehensive guide is based on Ray's GCS server source code, particularly files in src/ray/gcs/gcs\_server/. For the most current implementation details, refer to the source files and protobuf definitions in the Ray repository.

src/ray/gcs/gcs\_server/

# Chapter 7: Raylet Implementation and Lifecycle

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

Error Handling and Fault Tolerance

Performance Optimization

Code References and Implementation Details

## Introduction

The Raylet is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- Task scheduling and execution within a node  
- Resource management (CPU, GPU, memory)  
- Object management and storage coordination  
- Worker process lifecycle management  
- Communication coordination between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.

## Raylet Architecture Overview

Click to expand: High-level Architecture Diagram

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

### Core Components

The raylet consists of several interconnected components:  
  
  
graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

## Table of Contents  
1. [Introduction](#introduction)  
2. [Raylet Architecture Overview](#raylet-architecture-overview)  
3. [Raylet Lifecycle](#raylet-lifecycle)  
4. [Communication Mechanisms](#communication-mechanisms)  
5. [Task Scheduling and Load Handling](#task-scheduling-and-load-handling)  
6. [Worker Management](#worker-management)  
7. [Object Management](#object-management)  
8. [Resource Management](#resource-management)  
9. [Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance)  
10. [Performance Optimization](#performance-optimization)  
11. [Code References and Implementation Details](#code-references-and-implementation-details)  
## Introduction  
The \*\*Raylet\*\* is the core component of the Ray distributed computing framework that acts as the per-node manager responsible for:  
- \*\*Task scheduling and execution\*\* within a node  
- \*\*Resource management\*\* (CPU, GPU, memory)  
- \*\*Object management\*\* and storage coordination  
- \*\*Worker process lifecycle\*\* management  
- \*\*Communication coordination\*\* between workers and the global control store (GCS)  
This document provides a deep technical dive into how the raylet operates, its internal mechanisms, and implementation details for developers who need to understand or modify the raylet codebase.  
## Raylet Architecture Overview  
<details>  
<summary>Click to expand: High-level Architecture Diagram</summary>

┌─────────────────────────────────────────────────────────────────┐  
│ Ray Cluster │  
│ ┌─────────────────┐ ┌─────────────────┐ │  
│ │ Head Node │ │ Worker Nodes │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ GCS │ │◄──────────────────►│ │ Raylet │ │ │  
│ │ │ (Global │ │ │ │ │ │ │  
│ │ │ Control │ │ │ │ │ │ │  
│ │ │ Store) │ │ │ │ │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Raylet │ │ │ │ Object │ │ │  
│ │ │ │ │ │ │ Manager │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ │ │ │ │ │  
│ │ ┌───────────┐ │ │ ┌───────────┐ │ │  
│ │ │ Object │ │ │ │ Workers │ │ │  
│ │ │ Manager │ │ │ │ Pool │ │ │  
│ │ └───────────┘ │ │ └───────────┘ │ │  
│ └─────────────────┘ └─────────────────┘ │  
└─────────────────────────────────────────────────────────────────┘

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

</details>  
### Core Components  
The raylet consists of several interconnected components:  
<function\_calls>  
<invoke name="create\_diagram">  
<parameter name="content">graph TB  
subgraph "Raylet Process"  
A[Raylet Main] --> B[Node Manager]  
A --> C[Object Manager]  
B --> D[Local Task Manager]  
B --> E[Cluster Task Manager]  
B --> F[Worker Pool]  
B --> G[Resource Manager]  
D --> H[Dependency Manager]  
D --> I[Local Object Manager]  
E --> J[Cluster Resource Scheduler]  
F --> K[Worker Processes]  
F --> L[I/O Workers]  
C --> M[Plasma Store]  
C --> N[Object Directory]  
subgraph "External Interfaces"  
O[GCS Client]  
P[Core Worker RPC]  
Q[Node Manager RPC]  
end  
B --> O  
B --> P  
B --> Q  
end

## Table of Contents

Introduction

Raylet Architecture Overview

Raylet Lifecycle

Communication Mechanisms

Task Scheduling and Load Handling

Worker Management

Object Management

Resource Management

[Error Handling and Fault Tolerance](#error-handling-and-fault-tolerance

# Chapter 8: Distributed Object Store

## Table of Contents

Introduction

Architecture Overview

Local Storage: Plasma Store

Distributed Management: Object Manager

Global Coordination: Object Directory

Object Lifecycle Management

Memory Management and Spilling

Performance Characteristics

Implementation Details

Code Modification Guidelines

## Introduction

Ray's distributed object store is a sophisticated system that provides efficient storage, retrieval, and movement of large data objects across a distributed cluster. The system consists of three main components:  
1. Plasma Store: High-performance local object storage using shared memory  
2. Object Manager: Distributed object transfer and coordination  
3. Object Directory: Global metadata tracking via GCS (Global Control Service)  
The object store is designed to handle massive datasets efficiently while providing transparent access patterns for Ray applications.

## Architecture Overview

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Key Design Principles

Zero-Copy Access: Objects stored in shared memory for direct access

Distributed Transparency: Objects appear local regardless of actual location

Automatic Spilling: Graceful handling of memory pressure

Fault Tolerance: Reconstruction and replication capabilities

Performance Optimization: Chunked transfers and bandwidth management

## Local Storage: Plasma Store

The Plasma Store provides high-performance local object storage using memory-mapped shared memory.

### Plasma Architecture

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Object Storage Structure

From src/ray/object\_manager/plasma/plasma.h:35-70:

src/ray/object\_manager/plasma/plasma.h:35-70

struct PlasmaObject {  
MEMFD\_TYPE store\_fd; // Memory-mapped file descriptor  
ptrdiff\_t header\_offset; // Object header location  
ptrdiff\_t data\_offset; // Object data location  
ptrdiff\_t metadata\_offset; // Object metadata location  
int64\_t data\_size; // Size of object data  
int64\_t metadata\_size; // Size of object metadata  
int64\_t allocated\_size; // Total allocated space  
int device\_num; // Device identifier  
int64\_t mmap\_size; // Memory-mapped region size  
bool fallback\_allocated; // Whether using fallback storage  
bool is\_experimental\_mutable\_object; // Mutable object flag  
};

struct PlasmaObject {  
MEMFD\_TYPE store\_fd; // Memory-mapped file descriptor  
ptrdiff\_t header\_offset; // Object header location  
ptrdiff\_t data\_offset; // Object data location  
ptrdiff\_t metadata\_offset; // Object metadata location  
int64\_t data\_size; // Size of object data  
int64\_t metadata\_size; // Size of object metadata  
int64\_t allocated\_size; // Total allocated space  
int device\_num; // Device identifier  
int64\_t mmap\_size; // Memory-mapped region size  
bool fallback\_allocated; // Whether using fallback storage  
bool is\_experimental\_mutable\_object; // Mutable object flag  
};

### Memory Allocation Strategy

Block-Based Allocation:  
- Objects allocated in 64-byte aligned blocks (kBlockSize = 64)  
- Minimizes fragmentation through power-of-2 sizing  
- Supports both main memory and fallback filesystem storage  
Memory Layout:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

kBlockSize = 64

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

## Distributed Management: Object Manager

The Object Manager handles inter-node object transfers and distributed coordination.

### Object Manager Architecture

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Object Transfer Protocol

Ray uses a sophisticated chunked transfer protocol for large objects:  
📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

### Configuration and Performance Tuning

From src/ray/object\_manager/object\_manager.h:40-75:

src/ray/object\_manager/object\_manager.h:40-75

struct ObjectManagerConfig {  
std::string object\_manager\_address; // Network address  
int object\_manager\_port; // Listening port  
unsigned int timer\_freq\_ms; // Timer frequency  
unsigned int pull\_timeout\_ms; // Pull request timeout  
uint64\_t object\_chunk\_size; // Chunk size for transfers  
uint64\_t max\_bytes\_in\_flight; // Max concurrent transfer bytes  
std::string store\_socket\_name; // Plasma store socket  
int push\_timeout\_ms; // Push timeout  
int rpc\_service\_threads\_number; // RPC thread pool size  
int64\_t object\_store\_memory; // Total memory allocation  
std::string plasma\_directory; // Shared memory directory  
std::string fallback\_directory; // Fallback storage directory  
bool huge\_pages; // Enable huge page support  
};

struct ObjectManagerConfig {  
std::string object\_manager\_address; // Network address  
int object\_manager\_port; // Listening port  
unsigned int timer\_freq\_ms; // Timer frequency  
unsigned int pull\_timeout\_ms; // Pull request timeout  
uint64\_t object\_chunk\_size; // Chunk size for transfers  
uint64\_t max\_bytes\_in\_flight; // Max concurrent transfer bytes  
std::string store\_socket\_name; // Plasma store socket  
int push\_timeout\_ms; // Push timeout  
int rpc\_service\_threads\_number; // RPC thread pool size  
int64\_t object\_store\_memory; // Total memory allocation  
std::string plasma\_directory; // Shared memory directory  
std::string fallback\_directory; // Fallback storage directory  
bool huge\_pages; // Enable huge page support  
};

Key Performance Parameters:  
| Parameter | Default | Impact |  
|-----------|---------|---------|  
| object\_chunk\_size | 1MB | Transfer granularity, affects latency/throughput |  
| max\_bytes\_in\_flight | 256MB | Max concurrent transfer bandwidth |  
| pull\_timeout\_ms | 10s | Request timeout, affects fault tolerance |  
| rpc\_service\_threads\_number | min(max(2, cpu/4), 8) | Concurrency level |

object\_chunk\_size

max\_bytes\_in\_flight

pull\_timeout\_ms

rpc\_service\_threads\_number

## Global Coordination: Object Directory

The Object Directory provides cluster-wide object location tracking and metadata management.

### Object Directory Design

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Object Location Subscription Model

From src/ray/object\_manager/object\_directory.h:33-70:

src/ray/object\_manager/object\_directory.h:33-70

using OnLocationsFound = std::function<void(  
const ObjectID &object\_id,  
const std::unordered\_set<NodeID> &node\_locations,  
const std::string &spilled\_url,  
const NodeID &spilled\_node\_id,  
bool pending\_creation,  
size\_t object\_size)>;  
class IObjectDirectory {  
virtual Status SubscribeObjectLocations(  
const UniqueID &callback\_id,  
const ObjectID &object\_id,  
const rpc::Address &owner\_address,  
const OnLocationsFound &callback) = 0;  
virtual void ReportObjectAdded(  
const ObjectID &object\_id,  
const NodeID &node\_id,  
const ObjectInfo &object\_info) = 0;  
virtual void ReportObjectSpilled(  
const ObjectID &object\_id,  
const NodeID &node\_id,  
const rpc::Address &owner\_address,  
const std::string &spilled\_url,  
const ObjectID &generator\_id,  
bool spilled\_to\_local\_storage) = 0;  
};

using OnLocationsFound = std::function<void(  
const ObjectID &object\_id,  
const std::unordered\_set<NodeID> &node\_locations,  
const std::string &spilled\_url,  
const NodeID &spilled\_node\_id,  
bool pending\_creation,  
size\_t object\_size)>;  
class IObjectDirectory {  
virtual Status SubscribeObjectLocations(  
const UniqueID &callback\_id,  
const ObjectID &object\_id,  
const rpc::Address &owner\_address,  
const OnLocationsFound &callback) = 0;  
virtual void ReportObjectAdded(  
const ObjectID &object\_id,  
const NodeID &node\_id,  
const ObjectInfo &object\_info) = 0;  
virtual void ReportObjectSpilled(  
const ObjectID &object\_id,  
const NodeID &node\_id,  
const rpc::Address &owner\_address,  
const std::string &spilled\_url,  
const ObjectID &generator\_id,  
bool spilled\_to\_local\_storage) = 0;  
};

Location Update Flow:  
1. Object Creation: Node reports object addition to directory  
2. Subscription: Interested nodes subscribe to object locations  
3. Notification: Directory notifies subscribers of location changes  
4. Transfer: Subscribers initiate object transfers as needed

## Object Lifecycle Management

Ray objects go through a well-defined lifecycle from creation to deletion.

### Object Lifecycle States

🔧 TECHNICAL DIAGRAM: System Architecture

[DIAGRAM: 🔧 TECHNICAL DIAGRAM: System Architecture]

### Object Pinning and Reference Counting

From src/ray/raylet/local\_object\_manager.h:67-75:

src/ray/raylet/local\_object\_manager.h:67-75

void PinObjectsAndWaitForFree(  
const std::vector<ObjectID> &object\_ids,  
std::vector<std::unique\_ptr<RayObject>> &&objects,  
const rpc::Address &owner\_address,  
const ObjectID &generator\_id = ObjectID::Nil());  
struct LocalObjectInfo {  
rpc::Address owner\_address; // Object owner for reference counting  
bool is\_freed = false; // Whether object can be freed  
std::optional<ObjectID> generator\_id; // For dynamically created objects  
size\_t object\_size; // Object size for memory tracking  
};

void PinObjectsAndWaitForFree(  
const std::vector<ObjectID> &object\_ids,  
std::vector<std::unique\_ptr<RayObject>> &&objects,  
const rpc::Address &owner\_address,  
const ObjectID &generator\_id = ObjectID::Nil());  
struct LocalObjectInfo {  
rpc::Address owner\_address; // Object owner for reference counting  
bool is\_freed = false; // Whether object can be freed  
std::optional<ObjectID> generator\_id; // For dynamically created objects  
size\_t object\_size; // Object size for memory tracking  
};

Reference Counting Protocol:  
📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

## Memory Management and Spilling

Ray implements sophisticated memory management with automatic spilling to external storage.

### Memory Management Architecture

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Spilling Algorithm

From src/ray/raylet/local\_object\_manager.h:206-228:

src/ray/raylet/local\_object\_manager.h:206-228

// Spill objects asynchronously when space is needed  
bool TryToSpillObjects();  
// Internal spilling implementation with batching  
void SpillObjectsInternal(  
const std::vector<ObjectID> &objects\_ids,  
std::function<void(const ray::Status &)> callback);  
// Handle spilling completion and update metadata  
void OnObjectSpilled(  
const std::vector<ObjectID> &object\_ids,  
const rpc::SpillObjectsReply &worker\_reply);

// Spill objects asynchronously when space is needed  
bool TryToSpillObjects();  
// Internal spilling implementation with batching  
void SpillObjectsInternal(  
const std::vector<ObjectID> &objects\_ids,  
std::function<void(const ray::Status &)> callback);  
// Handle spilling completion and update metadata  
void OnObjectSpilled(  
const std::vector<ObjectID> &object\_ids,  
const rpc::SpillObjectsReply &worker\_reply);

Spilling Decision Algorithm:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Restoration and Fused Operations

Fused Restoration combines multiple small objects into single operations for efficiency:

// Maximum number of objects to fuse in single operation  
int64\_t max\_fused\_object\_count\_;  
// Restore spilled object from external storage  
void AsyncRestoreSpilledObject(  
const ObjectID &object\_id,  
int64\_t object\_size,  
const std::string &object\_url,  
std::function<void(const ray::Status &)> callback);

// Maximum number of objects to fuse in single operation  
int64\_t max\_fused\_object\_count\_;  
// Restore spilled object from external storage  
void AsyncRestoreSpilledObject(  
const ObjectID &object\_id,  
int64\_t object\_size,  
const std::string &object\_url,  
std::function<void(const ray::Status &)> callback);

## Performance Characteristics

### Throughput and Latency Analysis

Local Operations:  
| Operation | Latency | Throughput | Notes |  
|-----------|---------|------------|-------|  
| Local object access | < 1μs | ~50 GB/s | Direct shared memory access |  
| Object creation | 1-10μs | ~10 GB/s | Memory allocation + metadata |  
| Object deletion | < 1μs | ~20 GB/s | Reference counting + cleanup |  
Distributed Operations:  
| Operation | Latency | Throughput | Notes |  
|-----------|---------|------------|-------|  
| Remote object pull | 1-10ms + transfer\_time | ~1-5 GB/s per node | Network + chunking overhead |  
| Object location lookup | 0.1-1ms | ~10K ops/s | Object directory query |  
| Spilling to S3 | 10-100ms + transfer\_time | ~100-500 MB/s | Network + storage latency |  
Memory Management:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

## Implementation Details

### Critical Code Paths

Object Manager Core Loop (src/ray/object\_manager/object\_manager.cc):

src/ray/object\_manager/object\_manager.cc

class ObjectManager : public ObjectManagerInterface {  
// Handle pull request from remote nodes  
void HandlePull(rpc::PullRequest request,  
rpc::PullReply \*reply,  
rpc::SendReplyCallback send\_reply\_callback) override;  
// Handle push from remote nodes  
void HandlePush(rpc::PushRequest request,  
rpc::PushReply \*reply,  
rpc::SendReplyCallback send\_reply\_callback) override;  
// Pull objects from remote nodes  
uint64\_t Pull(const std::vector<rpc::ObjectReference> &object\_refs,  
BundlePriority prio,  
const TaskMetricsKey &task\_key) override;  
};

class ObjectManager : public ObjectManagerInterface {  
// Handle pull request from remote nodes  
void HandlePull(rpc::PullRequest request,  
rpc::PullReply \*reply,  
rpc::SendReplyCallback send\_reply\_callback) override;  
// Handle push from remote nodes  
void HandlePush(rpc::PushRequest request,  
rpc::PushReply \*reply,  
rpc::SendReplyCallback send\_reply\_callback) override;  
// Pull objects from remote nodes  
uint64\_t Pull(const std::vector<rpc::ObjectReference> &object\_refs,  
BundlePriority prio,  
const TaskMetricsKey &task\_key) override;  
};

Local Object Manager Operations:

class LocalObjectManager {  
// Pin objects and wait for owner to free them  
void PinObjectsAndWaitForFree(  
const std::vector<ObjectID> &object\_ids,  
std::vector<std::unique\_ptr<RayObject>> &&objects,  
const rpc::Address &owner\_address,  
const ObjectID &generator\_id);  
// Spill objects to external storage  
void SpillObjectUptoMaxThroughput();  
// Restore objects from external storage  
void AsyncRestoreSpilledObject(  
const ObjectID &object\_id,  
int64\_t object\_size,  
const std::string &object\_url,  
std::function<void(const ray::Status &)> callback);  
};

class LocalObjectManager {  
// Pin objects and wait for owner to free them  
void PinObjectsAndWaitForFree(  
const std::vector<ObjectID> &object\_ids,  
std::vector<std::unique\_ptr<RayObject>> &&objects,  
const rpc::Address &owner\_address,  
const ObjectID &generator\_id);  
// Spill objects to external storage  
void SpillObjectUptoMaxThroughput();  
// Restore objects from external storage  
void AsyncRestoreSpilledObject(  
const ObjectID &object\_id,  
int64\_t object\_size,  
const std::string &object\_url,  
std::function<void(const ray::Status &)> callback);  
};

### Error Handling and Recovery

Fault Tolerance Mechanisms:  
1. Object Reconstruction: If objects are lost, Ray can reconstruct them by re-executing the tasks that created them  
2. Replication: Critical objects can be replicated across multiple nodes  
3. Spill Redundancy: Objects spilled to external storage maintain multiple copies  
4. Network Resilience: Failed transfers are automatically retried with exponential backoff  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

## Code Modification Guidelines

### Adding New Object Store Features

1. Local Storage Modifications:  
To modify Plasma store behavior, focus on these key files:  
- src/ray/object\_manager/plasma/plasma.cc - Core storage logic  
- src/ray/object\_manager/plasma/plasma\_allocator.cc - Memory allocation  
- src/ray/raylet/local\_object\_manager.cc - Raylet integration  
2. Distributed Transfer Modifications:  
For object transfer improvements:  
- src/ray/object\_manager/object\_manager.cc - Main transfer logic  
- src/ray/object\_manager/pull\_manager.cc - Pull request handling  
- src/ray/object\_manager/push\_manager.cc - Push request handling  
3. Spilling and External Storage:  
For spilling enhancements:  
- src/ray/raylet/local\_object\_manager.cc - Spilling coordination  
- External storage interfaces in worker processes

src/ray/object\_manager/plasma/plasma.cc

src/ray/object\_manager/plasma/plasma\_allocator.cc

src/ray/raylet/local\_object\_manager.cc

src/ray/object\_manager/object\_manager.cc

src/ray/object\_manager/pull\_manager.cc

src/ray/object\_manager/push\_manager.cc

src/ray/raylet/local\_object\_manager.cc

### Example: Adding a New Spilling Strategy

// In LocalObjectManager class  
bool TryToSpillObjectsCustomStrategy() {  
// 1. Implement custom object selection logic  
std::vector<ObjectID> objects\_to\_spill = SelectObjectsCustomCriteria();  
// 2. Check if objects meet spilling requirements  
if (objects\_to\_spill.empty() ||  
total\_size < min\_spilling\_size\_) {  
return false;  
}  
// 3. Initiate spilling with custom parameters  
SpillObjectsInternal(objects\_to\_spill,  
[this](const ray::Status &status) {  
// Custom completion handling  
});  
return true;  
}

// In LocalObjectManager class  
bool TryToSpillObjectsCustomStrategy() {  
// 1. Implement custom object selection logic  
std::vector<ObjectID> objects\_to\_spill = SelectObjectsCustomCriteria();  
// 2. Check if objects meet spilling requirements  
if (objects\_to\_spill.empty() ||  
total\_size < min\_spilling\_size\_) {  
return false;  
}  
// 3. Initiate spilling with custom parameters  
SpillObjectsInternal(objects\_to\_spill,  
[this](const ray::Status &status) {  
// Custom completion handling  
});  
return true;  
}

### Testing and Validation

Key Testing Areas:  
1. Unit Tests: Individual component functionality  
2. Integration Tests: Cross-component interactions  
3. Performance Tests: Throughput and latency benchmarks  
4. Fault Injection: Network failures, storage failures, node crashes  
5. Scale Tests: Large object handling, many-node clusters  
Performance Validation Commands:

ray start --head --object-store-memory=8000000000  
python -c "  
import ray  
import numpy as np  
ray.init()  
obj = ray.put(np.random.rand(100000000)) # ~800MB object  
result = ray.get(obj)  
"  
ray status --verbose

ray start --head --object-store-memory=8000000000  
python -c "  
import ray  
import numpy as np  
ray.init()  
obj = ray.put(np.random.rand(100000000)) # ~800MB object  
result = ray.get(obj)  
"  
ray status --verbose

This guide is based on Ray's source code, particularly the object manager, plasma store, and local object manager implementations. For the most current details, refer to the source files in src/ray/object\_manager/ and src/ray/raylet/.

src/ray/object\_manager/

src/ray/raylet/

# Chapter 9: Distributed Scheduling Implementation

## Table of Contents

Introduction

Scheduling Architecture Overview

Core Scheduling Components

Resource Management and Allocation

Task Scheduling Algorithms

Actor Placement and Scheduling

Placement Group Scheduling

Scheduling Strategies

Node Affinity and Label-Based Scheduling

Locality-Aware Scheduling

Cluster Resource Scheduling

Autoscaler Integration

Performance Characteristics

Configuration and Tuning

Implementation Deep Dive

Testing and Verification

Best Practices

Troubleshooting

## Introduction

Ray's distributed scheduling system is a sophisticated multi-layered scheduler designed to efficiently allocate resources and place tasks/actors across a distributed cluster. This chapter dives deep into the scheduling implementation, covering complex scheduling scenarios including resource constraints, placement groups, locality preferences, and autoscaling decisions while maintaining high performance and fault tolerance.

### What is Ray?

Ray is an open-source unified framework for scaling AI workloads. It provides:  
- Distributed Computing: Scale Python workloads across multiple machines  
- Unified API: Single interface for tasks, actors, and data processing  
- Fault Tolerance: Built-in error handling and recovery mechanisms  
- Resource Management: Efficient allocation of CPU, GPU, and memory resources  
- Ecosystem: Libraries for ML (Ray Train), reinforcement learning (Ray RLlib), hyperparameter tuning (Ray Tune), and more

### Key Features

Multi-level Scheduling: Task-level, actor-level, and placement group scheduling

Resource-Aware: CPU, GPU, memory, and custom resource scheduling

Placement Strategies: PACK, SPREAD, STRICT\_PACK, STRICT\_SPREAD

Locality Optimization: Data locality-aware task placement

Dynamic Scaling: Integration with autoscaler for cluster growth/shrinkage

Label-Based Scheduling: Node affinity and label constraints

Performance Optimization: Efficient algorithms for large-scale clusters

### Scheduling Hierarchy

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

## Scheduling Architecture Overview

### Multi-Level Scheduling Architecture

Ray implements a hierarchical scheduling architecture with multiple decision points:

#### 1. Client-Side Scheduling

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships  
Location: src/ray/core\_worker/lease\_policy.cc  
The client-side scheduling makes initial placement decisions based on:  
- Data locality (object location)  
- Scheduling strategies (spread, node affinity)  
- Resource requirements

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

src/ray/core\_worker/lease\_policy.cc

#### 2. Raylet-Level Scheduling

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships  
Location: src/ray/raylet/scheduling/cluster\_task\_manager.cc

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

src/ray/raylet/scheduling/cluster\_task\_manager.cc

#### 3. GCS-Level Scheduling

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships  
Location: src/ray/gcs/gcs\_server/gcs\_actor\_scheduler.cc

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

src/ray/gcs/gcs\_server/gcs\_actor\_scheduler.cc

### Core Scheduling Flow

📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

## Core Scheduling Components

### ClusterResourceScheduler

Location: src/ray/raylet/scheduling/cluster\_resource\_scheduler.h  
The central coordinator for cluster-wide resource scheduling decisions.

src/ray/raylet/scheduling/cluster\_resource\_scheduler.h

class ClusterResourceScheduler {  
// Core scheduling method  
scheduling::NodeID GetBestSchedulableNode(  
const ResourceRequest &resource\_request,  
const rpc::SchedulingStrategy &scheduling\_strategy,  
bool actor\_creation,  
bool force\_spillback,  
const std::string &preferred\_node\_id,  
int64\_t \*total\_violations,  
bool \*is\_infeasible);  
// Bundle scheduling for placement groups  
SchedulingResult Schedule(  
const std::vector<const ResourceRequest \*> &resource\_request\_list,  
SchedulingOptions options);  
}

class ClusterResourceScheduler {  
// Core scheduling method  
scheduling::NodeID GetBestSchedulableNode(  
const ResourceRequest &resource\_request,  
const rpc::SchedulingStrategy &scheduling\_strategy,  
bool actor\_creation,  
bool force\_spillback,  
const std::string &preferred\_node\_id,  
int64\_t \*total\_violations,  
bool \*is\_infeasible);  
// Bundle scheduling for placement groups  
SchedulingResult Schedule(  
const std::vector<const ResourceRequest \*> &resource\_request\_list,  
SchedulingOptions options);  
}

Key Responsibilities:  
- Node feasibility checking  
- Resource availability tracking  
- Scheduling strategy implementation  
- Placement group bundle scheduling

### ClusterTaskManager

Location: src/ray/raylet/scheduling/cluster\_task\_manager.h  
Manages task queuing and scheduling at the cluster level.

src/ray/raylet/scheduling/cluster\_task\_manager.h

class ClusterTaskManager {  
void QueueAndScheduleTask(  
RayTask task,  
bool grant\_or\_reject,  
bool is\_selected\_based\_on\_locality,  
rpc::RequestWorkerLeaseReply \*reply,  
rpc::SendReplyCallback send\_reply\_callback);  
void ScheduleAndDispatchTasks();  
}

class ClusterTaskManager {  
void QueueAndScheduleTask(  
RayTask task,  
bool grant\_or\_reject,  
bool is\_selected\_based\_on\_locality,  
rpc::RequestWorkerLeaseReply \*reply,  
rpc::SendReplyCallback send\_reply\_callback);  
void ScheduleAndDispatchTasks();  
}

Scheduling Queues:  
- tasks\_to\_schedule\_: Tasks waiting for resources  
- infeasible\_tasks\_: Tasks that cannot be scheduled

tasks\_to\_schedule\_

infeasible\_tasks\_

### LocalTaskManager

Location: src/ray/raylet/local\_task\_manager.h  
Handles local task execution and worker management.

src/ray/raylet/local\_task\_manager.h

class LocalTaskManager {  
void QueueAndScheduleTask(std::shared\_ptr<internal::Work> work);  
void ScheduleAndDispatchTasks();  
bool TrySpillback(const std::shared\_ptr<internal::Work> &work,  
bool &is\_infeasible);  
}

class LocalTaskManager {  
void QueueAndScheduleTask(std::shared\_ptr<internal::Work> work);  
void ScheduleAndDispatchTasks();  
bool TrySpillback(const std::shared\_ptr<internal::Work> &work,  
bool &is\_infeasible);  
}

Fairness Policy: Implements CPU-fair scheduling to prevent resource starvation:

// From src/ray/raylet/local\_task\_manager.cc  
if (total\_cpu\_requests\_ > total\_cpus) {  
RAY\_LOG(DEBUG) << "Applying fairness policy. Total CPU requests ("  
<< total\_cpu\_requests\_ << ") exceed total CPUs ("  
<< total\_cpus << ")";  
// Apply fair dispatching logic  
}

// From src/ray/raylet/local\_task\_manager.cc  
if (total\_cpu\_requests\_ > total\_cpus) {  
RAY\_LOG(DEBUG) << "Applying fairness policy. Total CPU requests ("  
<< total\_cpu\_requests\_ << ") exceed total CPUs ("  
<< total\_cpus << ")";  
// Apply fair dispatching logic  
}

### Scheduling Policies

Location: src/ray/raylet/scheduling/policy/  
Ray implements multiple scheduling policies:

src/ray/raylet/scheduling/policy/

#### HybridSchedulingPolicy

Default scheduling strategy

Balances locality and load distribution

Configurable spread threshold

#### SpreadSchedulingPolicy

Distributes tasks across nodes

Minimizes resource contention

Used for embarrassingly parallel workloads

#### NodeAffinitySchedulingPolicy

Hard/soft node constraints

Supports spillback on unavailability

Critical for stateful workloads

#### NodeLabelSchedulingPolicy

class NodeLabelSchedulingPolicy : public ISchedulingPolicy {  
scheduling::NodeID Schedule(const ResourceRequest &resource\_request,  
SchedulingOptions options) override;  
private:  
bool IsNodeMatchLabelExpression(const Node &node,  
const rpc::LabelMatchExpression &expression);  
};

class NodeLabelSchedulingPolicy : public ISchedulingPolicy {  
scheduling::NodeID Schedule(const ResourceRequest &resource\_request,  
SchedulingOptions options) override;  
private:  
bool IsNodeMatchLabelExpression(const Node &node,  
const rpc::LabelMatchExpression &expression);  
};

### Scheduling Context and Options

Location: src/ray/raylet/scheduling/policy/scheduling\_options.h

src/ray/raylet/scheduling/policy/scheduling\_options.h

struct SchedulingOptions {  
SchedulingType scheduling\_type;  
float spread\_threshold;  
bool avoid\_local\_node;  
bool require\_node\_available;  
bool avoid\_gpu\_nodes;  
double max\_cpu\_fraction\_per\_node; // For placement groups  
static SchedulingOptions Hybrid(bool avoid\_local\_node,  
bool require\_node\_available,  
const std::string &preferred\_node\_id);  
static SchedulingOptions BundlePack(double max\_cpu\_fraction\_per\_node = 1.0);  
static SchedulingOptions BundleStrictSpread(double max\_cpu\_fraction\_per\_node = 1.0);  
};

struct SchedulingOptions {  
SchedulingType scheduling\_type;  
float spread\_threshold;  
bool avoid\_local\_node;  
bool require\_node\_available;  
bool avoid\_gpu\_nodes;  
double max\_cpu\_fraction\_per\_node; // For placement groups  
static SchedulingOptions Hybrid(bool avoid\_local\_node,  
bool require\_node\_available,  
const std::string &preferred\_node\_id);  
static SchedulingOptions BundlePack(double max\_cpu\_fraction\_per\_node = 1.0);  
static SchedulingOptions BundleStrictSpread(double max\_cpu\_fraction\_per\_node = 1.0);  
};

## Resource Management and Allocation

### Resource Model

Ray uses a multi-dimensional resource model:

// Resource types from src/ray/common/scheduling/scheduling\_ids.h  
enum PredefinedResources {  
CPU = 0,  
MEM = 1,  
GPU = 2,  
OBJECT\_STORE\_MEM = 3,  
// Custom resources start from 4  
};

// Resource types from src/ray/common/scheduling/scheduling\_ids.h  
enum PredefinedResources {  
CPU = 0,  
MEM = 1,  
GPU = 2,  
OBJECT\_STORE\_MEM = 3,  
// Custom resources start from 4  
};

### Resource Request Structure

class ResourceRequest {  
ResourceSet resource\_set\_; // Required resources  
LabelSelector label\_selector\_; // Node label requirements  
bool requires\_object\_store\_memory\_; // Memory constraint flag  
bool IsEmpty() const;  
const ResourceSet &GetResourceSet() const;  
bool RequiresObjectStoreMemory() const;  
};

class ResourceRequest {  
ResourceSet resource\_set\_; // Required resources  
LabelSelector label\_selector\_; // Node label requirements  
bool requires\_object\_store\_memory\_; // Memory constraint flag  
bool IsEmpty() const;  
const ResourceSet &GetResourceSet() const;  
bool RequiresObjectStoreMemory() const;  
};

### NodeResources

Location: src/ray/common/scheduling/cluster\_resource\_data.h

src/ray/common/scheduling/cluster\_resource\_data.h

struct NodeResources {  
NodeResourceSet total; // Total node capacity  
NodeResourceSet available; // Currently available  
NodeResourceSet normal\_task\_resources; // Reserved for tasks  
absl::flat\_hash\_map<std::string, std::string> labels; // Node labels  
bool object\_pulls\_queued; // Object store status  
bool IsAvailable(const ResourceRequest &resource\_request) const;  
bool IsFeasible(const ResourceRequest &resource\_request) const;  
bool HasRequiredLabels(const LabelSelector &label\_selector) const;  
float CalculateCriticalResourceUtilization() const;  
};

struct NodeResources {  
NodeResourceSet total; // Total node capacity  
NodeResourceSet available; // Currently available  
NodeResourceSet normal\_task\_resources; // Reserved for tasks  
absl::flat\_hash\_map<std::string, std::string> labels; // Node labels  
bool object\_pulls\_queued; // Object store status  
bool IsAvailable(const ResourceRequest &resource\_request) const;  
bool IsFeasible(const ResourceRequest &resource\_request) const;  
bool HasRequiredLabels(const LabelSelector &label\_selector) const;  
float CalculateCriticalResourceUtilization() const;  
};

### Resource Allocation Algorithm

bool ClusterResourceScheduler::IsSchedulable(  
const ResourceRequest &resource\_request,  
scheduling::NodeID node\_id) const {  
return cluster\_resource\_manager\_->HasAvailableResources(  
node\_id,  
resource\_request,  
/\*ignore\_object\_store\_memory\_requirement\*/  
node\_id == local\_node\_id\_) &&  
NodeAvailable(node\_id);  
}

bool ClusterResourceScheduler::IsSchedulable(  
const ResourceRequest &resource\_request,  
scheduling::NodeID node\_id) const {  
return cluster\_resource\_manager\_->HasAvailableResources(  
node\_id,  
resource\_request,  
/\*ignore\_object\_store\_memory\_requirement\*/  
node\_id == local\_node\_id\_) &&  
NodeAvailable(node\_id);  
}

### Dynamic Resource Management

// From src/ray/raylet/scheduling/cluster\_resource\_scheduler\_test.cc  
TEST\_F(ClusterResourceSchedulerTest, DynamicResourceTest) {  
// Add dynamic resources at runtime  
resource\_scheduler.GetLocalResourceManager().AddLocalResourceInstances(  
scheduling::ResourceID("custom123"), {0., 1.0, 1.0});  
// Verify schedulability  
auto result = resource\_scheduler.GetBestSchedulableNode(resource\_request, ...);  
ASSERT\_FALSE(result.IsNil());  
}

// From src/ray/raylet/scheduling/cluster\_resource\_scheduler\_test.cc  
TEST\_F(ClusterResourceSchedulerTest, DynamicResourceTest) {  
// Add dynamic resources at runtime  
resource\_scheduler.GetLocalResourceManager().AddLocalResourceInstances(  
scheduling::ResourceID("custom123"), {0., 1.0, 1.0});  
// Verify schedulability  
auto result = resource\_scheduler.GetBestSchedulableNode(resource\_request, ...);  
ASSERT\_FALSE(result.IsNil());  
}

### Resource Binpacking

Ray implements sophisticated binpacking for resource allocation:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

## Task Scheduling Algorithms

### Hybrid Scheduling Algorithm

Default Strategy: Balances locality and load distribution

// Configuration from src/ray/raylet/scheduling/cluster\_resource\_scheduler.cc  
best\_node\_id = scheduling\_policy\_->Schedule(  
resource\_request,  
SchedulingOptions::Hybrid(  
/\*avoid\_local\_node\*/ force\_spillback,  
/\*require\_node\_available\*/ force\_spillback,  
preferred\_node\_id));

// Configuration from src/ray/raylet/scheduling/cluster\_resource\_scheduler.cc  
best\_node\_id = scheduling\_policy\_->Schedule(  
resource\_request,  
SchedulingOptions::Hybrid(  
/\*avoid\_local\_node\*/ force\_spillback,  
/\*require\_node\_available\*/ force\_spillback,  
preferred\_node\_id));

Algorithm Steps:  
1. Score Calculation: Based on resource utilization  
2. Top-K Selection: Choose from best k nodes (default: 20% of cluster)  
3. Random Selection: Within top-k for load balancing  
Scoring Function:

float NodeResources::CalculateCriticalResourceUtilization() const {  
float highest = 0;  
for (const auto &i : {CPU, MEM, OBJECT\_STORE\_MEM}) {  
float utilization = 1 - (available / total);  
if (utilization > highest) {  
highest = utilization;  
}  
}  
return highest;  
}

float NodeResources::CalculateCriticalResourceUtilization() const {  
float highest = 0;  
for (const auto &i : {CPU, MEM, OBJECT\_STORE\_MEM}) {  
float utilization = 1 - (available / total);  
if (utilization > highest) {  
highest = utilization;  
}  
}  
return highest;  
}

### Spread Scheduling Algorithm

Purpose: Distribute tasks across maximum number of nodes

// From scheduling policy tests  
TEST\_F(SchedulingPolicyTest, SpreadSchedulingStrategyTest) {  
rpc::SchedulingStrategy scheduling\_strategy;  
scheduling\_strategy.mutable\_spread\_scheduling\_strategy();  
auto node\_id = resource\_scheduler.GetBestSchedulableNode(  
resource\_request, LabelSelector(), scheduling\_strategy, ...);  
}

// From scheduling policy tests  
TEST\_F(SchedulingPolicyTest, SpreadSchedulingStrategyTest) {  
rpc::SchedulingStrategy scheduling\_strategy;  
scheduling\_strategy.mutable\_spread\_scheduling\_strategy();  
auto node\_id = resource\_scheduler.GetBestSchedulableNode(  
resource\_request, LabelSelector(), scheduling\_strategy, ...);  
}

Implementation:  
- Prioritizes nodes with lowest task count  
- Avoids resource hotspots  
- Maximizes fault tolerance

### Node Affinity Scheduling

Hard Affinity: Must run on specific node

if (IsHardNodeAffinitySchedulingStrategy(scheduling\_strategy)) {  
// Must schedule on specified node or fail  
best\_node\_id = scheduling\_policy\_->Schedule(  
resource\_request,  
SchedulingOptions::NodeAffinity(  
force\_spillback, force\_spillback,  
scheduling\_strategy.node\_affinity\_scheduling\_strategy().node\_id(),  
/\*soft=\*/false, /\*spill\_on\_unavailable=\*/false,  
/\*fail\_on\_unavailable=\*/true));  
}

if (IsHardNodeAffinitySchedulingStrategy(scheduling\_strategy)) {  
// Must schedule on specified node or fail  
best\_node\_id = scheduling\_policy\_->Schedule(  
resource\_request,  
SchedulingOptions::NodeAffinity(  
force\_spillback, force\_spillback,  
scheduling\_strategy.node\_affinity\_scheduling\_strategy().node\_id(),  
/\*soft=\*/false, /\*spill\_on\_unavailable=\*/false,  
/\*fail\_on\_unavailable=\*/true));  
}

Soft Affinity: Prefer specific node but allow spillback

scheduling\_strategy.mutable\_node\_affinity\_scheduling\_strategy()->set\_soft(true);  
// Will try preferred node first, then other nodes

scheduling\_strategy.mutable\_node\_affinity\_scheduling\_strategy()->set\_soft(true);  
// Will try preferred node first, then other nodes

### Fair Scheduling

CPU Fair Scheduling: Prevents starvation across scheduling classes

// From src/ray/raylet/local\_task\_manager.cc  
if (total\_cpu\_requests\_ > total\_cpus) {  
// Calculate fair share per scheduling class  
double fair\_share = total\_cpus / num\_classes\_with\_cpu;  
// Apply throttling based on fair share  
for (auto &[scheduling\_class, dispatch\_queue] : tasks\_to\_dispatch\_) {  
double cpu\_request = /\* CPU required by this class \*/;  
if (cpu\_request > fair\_share) {  
// Throttle this class  
next\_update\_time = current\_time + throttle\_delay;  
}  
}  
}

// From src/ray/raylet/local\_task\_manager.cc  
if (total\_cpu\_requests\_ > total\_cpus) {  
// Calculate fair share per scheduling class  
double fair\_share = total\_cpus / num\_classes\_with\_cpu;  
// Apply throttling based on fair share  
for (auto &[scheduling\_class, dispatch\_queue] : tasks\_to\_dispatch\_) {  
double cpu\_request = /\* CPU required by this class \*/;  
if (cpu\_request > fair\_share) {  
// Throttle this class  
next\_update\_time = current\_time + throttle\_delay;  
}  
}  
}

## Actor Placement and Scheduling

### Actor Scheduling Architecture

Location: src/ray/gcs/gcs\_server/gcs\_actor\_scheduler.cc  
Ray provides two actor scheduling modes:

src/ray/gcs/gcs\_server/gcs\_actor\_scheduler.cc

#### 1. GCS-Based Actor Scheduling

void GcsActorScheduler::ScheduleByGcs(std::shared\_ptr<GcsActor> actor) {  
// Create task for actor creation  
auto task = std::make\_shared<RayTask>(actor->GetCreationTaskSpecification());  
// Use cluster task manager for scheduling  
cluster\_task\_manager\_.QueueAndScheduleTask(  
std::move(task),  
/\*grant\_or\_reject\*/ false,  
/\*is\_selected\_based\_on\_locality\*/ false,  
reply.get(),  
send\_reply\_callback);  
}

void GcsActorScheduler::ScheduleByGcs(std::shared\_ptr<GcsActor> actor) {  
// Create task for actor creation  
auto task = std::make\_shared<RayTask>(actor->GetCreationTaskSpecification());  
// Use cluster task manager for scheduling  
cluster\_task\_manager\_.QueueAndScheduleTask(  
std::move(task),  
/\*grant\_or\_reject\*/ false,  
/\*is\_selected\_based\_on\_locality\*/ false,  
reply.get(),  
send\_reply\_callback);  
}

#### 2. Raylet-Based Actor Scheduling

void GcsActorScheduler::ScheduleByRaylet(std::shared\_ptr<GcsActor> actor) {  
// Select forwarding node  
auto node\_id = SelectForwardingNode(actor);  
// Lease worker directly from node  
LeaseWorkerFromNode(actor, node.value());  
}

void GcsActorScheduler::ScheduleByRaylet(std::shared\_ptr<GcsActor> actor) {  
// Select forwarding node  
auto node\_id = SelectForwardingNode(actor);  
// Lease worker directly from node  
LeaseWorkerFromNode(actor, node.value());  
}

### Actor Resource Requirements

Placement vs Execution Resources:

// From src/ray/common/task/task\_spec.cc  
const auto &resource\_set =  
(is\_actor\_creation\_task && should\_report\_placement\_resources)  
? GetRequiredPlacementResources() // For scheduling decisions  
: GetRequiredResources(); // For execution

// From src/ray/common/task/task\_spec.cc  
const auto &resource\_set =  
(is\_actor\_creation\_task && should\_report\_placement\_resources)  
? GetRequiredPlacementResources() // For scheduling decisions  
: GetRequiredResources(); // For execution

Actor Creation Example:

@ray.remote(num\_cpus=2, num\_gpus=1, memory=1000)  
class MyActor:  
def \_\_init\_\_(self):  
pass  
def method(self):  
pass  
# Actor placement considers both creation and method resources  
actor = MyActor.remote()

@ray.remote(num\_cpus=2, num\_gpus=1, memory=1000)  
class MyActor:  
def \_\_init\_\_(self):  
pass  
def method(self):  
pass  
# Actor placement considers both creation and method resources  
actor = MyActor.remote()

### Actor Lifecycle and Scheduling

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Actor Scheduling Considerations

Resource Lifetime: Actors hold resources for their entire lifetime

if (task\_spec.IsActorCreationTask()) {  
// The actor belongs to this worker now  
worker->SetLifetimeAllocatedInstances(allocated\_instances);  
} else {  
worker->SetAllocatedInstances(allocated\_instances);  
}

if (task\_spec.IsActorCreationTask()) {  
// The actor belongs to this worker now  
worker->SetLifetimeAllocatedInstances(allocated\_instances);  
} else {  
worker->SetAllocatedInstances(allocated\_instances);  
}

Scheduling Class: Actors use placement resources for scheduling decisions

TEST(TaskSpecTest, TestActorSchedulingClass) {  
// Actor's scheduling class determined by placement resources  
TaskSpecification actor\_task(actor\_task\_spec\_proto);  
TaskSpecification regular\_task(regular\_task\_spec\_proto);  
ASSERT\_EQ(regular\_task.GetSchedulingClass(), actor\_task.GetSchedulingClass());  
}

TEST(TaskSpecTest, TestActorSchedulingClass) {  
// Actor's scheduling class determined by placement resources  
TaskSpecification actor\_task(actor\_task\_spec\_proto);  
TaskSpecification regular\_task(regular\_task\_spec\_proto);  
ASSERT\_EQ(regular\_task.GetSchedulingClass(), actor\_task.GetSchedulingClass());  
}

## Placement Group Scheduling

### Placement Group Architecture

Location: src/ray/gcs/gcs\_server/gcs\_placement\_group\_scheduler.cc  
Placement groups enable gang scheduling of related resources across multiple nodes.

src/ray/gcs/gcs\_server/gcs\_placement\_group\_scheduler.cc

class GcsPlacementGroupScheduler {  
void SchedulePlacementGroup(  
std::shared\_ptr<GcsPlacementGroup> placement\_group,  
PGSchedulingFailureCallback failure\_callback,  
PGSchedulingSuccessfulCallback success\_callback);  
}

class GcsPlacementGroupScheduler {  
void SchedulePlacementGroup(  
std::shared\_ptr<GcsPlacementGroup> placement\_group,  
PGSchedulingFailureCallback failure\_callback,  
PGSchedulingSuccessfulCallback success\_callback);  
}

### Bundle Specification

Location: src/ray/common/bundle\_spec.h

src/ray/common/bundle\_spec.h

class BundleSpecification {  
BundleID BundleId() const;  
PlacementGroupID PlacementGroupId() const;  
NodeID NodeId() const;  
int64\_t Index() const;  
const ResourceRequest &GetRequiredResources() const;  
const absl::flat\_hash\_map<std::string, double> &GetFormattedResources() const;  
};

class BundleSpecification {  
BundleID BundleId() const;  
PlacementGroupID PlacementGroupId() const;  
NodeID NodeId() const;  
int64\_t Index() const;  
const ResourceRequest &GetRequiredResources() const;  
const absl::flat\_hash\_map<std::string, double> &GetFormattedResources() const;  
};

### Placement Strategies

#### PACK Strategy

case rpc::PlacementStrategy::PACK:  
return SchedulingOptions::BundlePack(max\_cpu\_fraction\_per\_node);

case rpc::PlacementStrategy::PACK:  
return SchedulingOptions::BundlePack(max\_cpu\_fraction\_per\_node);

Goal: Minimize number of nodes used

Use Case: Maximize locality, minimize network overhead

Algorithm: First-fit decreasing binpacking

#### SPREAD Strategy

case rpc::PlacementStrategy::SPREAD:  
return SchedulingOptions::BundleSpread(max\_cpu\_fraction\_per\_node);

case rpc::PlacementStrategy::SPREAD:  
return SchedulingOptions::BundleSpread(max\_cpu\_fraction\_per\_node);

Goal: Distribute bundles across nodes

Use Case: Fault tolerance, load distribution

Algorithm: Round-robin placement with load balancing

#### STRICT\_PACK Strategy

case rpc::PlacementStrategy::STRICT\_PACK:  
return SchedulingOptions::BundleStrictPack(  
max\_cpu\_fraction\_per\_node,  
soft\_target\_node\_id);

case rpc::PlacementStrategy::STRICT\_PACK:  
return SchedulingOptions::BundleStrictPack(  
max\_cpu\_fraction\_per\_node,  
soft\_target\_node\_id);

Goal: All bundles on single node (if possible)

Use Case: Shared memory, minimal latency

Algorithm: Single-node placement with fallback

#### STRICT\_SPREAD Strategy

case rpc::PlacementStrategy::STRICT\_SPREAD:  
return SchedulingOptions::BundleStrictSpread(  
max\_cpu\_fraction\_per\_node,  
CreateSchedulingContext(placement\_group\_id));

case rpc::PlacementStrategy::STRICT\_SPREAD:  
return SchedulingOptions::BundleStrictSpread(  
max\_cpu\_fraction\_per\_node,  
CreateSchedulingContext(placement\_group\_id));

Goal: Each bundle on different node

Use Case: Maximum fault tolerance

Algorithm: One bundle per node constraint

### Bundle Scheduling Algorithm

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Bundle Resource Formatting

Ray formats placement group resources with special naming:

// From src/ray/common/bundle\_spec.h  
std::string FormatPlacementGroupResource(  
const std::string &original\_resource\_name,  
const std::string &group\_id\_str,  
int64\_t bundle\_index) {  
if (bundle\_index == -1) {  
// Wildcard resource: CPU\_group\_<group\_id>  
return original\_resource\_name + "\_group\_" + group\_id\_str;  
} else {  
// Indexed resource: CPU\_group\_<bundle\_index>\_<group\_id>  
return original\_resource\_name + "\_group\_" +  
std::to\_string(bundle\_index) + "\_" + group\_id\_str;  
}  
}

// From src/ray/common/bundle\_spec.h  
std::string FormatPlacementGroupResource(  
const std::string &original\_resource\_name,  
const std::string &group\_id\_str,  
int64\_t bundle\_index) {  
if (bundle\_index == -1) {  
// Wildcard resource: CPU\_group\_<group\_id>  
return original\_resource\_name + "\_group\_" + group\_id\_str;  
} else {  
// Indexed resource: CPU\_group\_<bundle\_index>\_<group\_id>  
return original\_resource\_name + "\_group\_" +  
std::to\_string(bundle\_index) + "\_" + group\_id\_str;  
}  
}

### CPU Fraction Limits

Purpose: Prevent placement groups from monopolizing nodes

bool AllocationWillExceedMaxCpuFraction(  
const NodeResources &node\_resources,  
const ResourceRequest &bundle\_resource\_request,  
double max\_cpu\_fraction\_per\_node,  
double available\_cpus\_before\_current\_pg\_request) {  
if (max\_cpu\_fraction\_per\_node == 1.0) {  
return false; // No limit  
}  
auto max\_reservable\_cpus =  
max\_cpu\_fraction\_per\_node \* node\_resources.total.Get(cpu\_id).Double();  
// Ensure at least 1 CPU is excluded from placement groups  
if (max\_reservable\_cpus > total\_cpus - 1) {  
max\_reservable\_cpus = total\_cpus - 1;  
}  
return cpus\_used\_by\_pg\_after > max\_reservable\_cpus;  
}

bool AllocationWillExceedMaxCpuFraction(  
const NodeResources &node\_resources,  
const ResourceRequest &bundle\_resource\_request,  
double max\_cpu\_fraction\_per\_node,  
double available\_cpus\_before\_current\_pg\_request) {  
if (max\_cpu\_fraction\_per\_node == 1.0) {  
return false; // No limit  
}  
auto max\_reservable\_cpus =  
max\_cpu\_fraction\_per\_node \* node\_resources.total.Get(cpu\_id).Double();  
// Ensure at least 1 CPU is excluded from placement groups  
if (max\_reservable\_cpus > total\_cpus - 1) {  
max\_reservable\_cpus = total\_cpus - 1;  
}  
return cpus\_used\_by\_pg\_after > max\_reservable\_cpus;  
}

### Placement Group Lifecycle

📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

## Scheduling Strategies

### Strategy Types and Implementation

Ray supports multiple scheduling strategies through the rpc::SchedulingStrategy protocol buffer:

rpc::SchedulingStrategy

// From src/ray/raylet/scheduling/cluster\_resource\_scheduler.cc  
scheduling::NodeID ClusterResourceScheduler::GetBestSchedulableNode(  
const ResourceRequest &resource\_request,  
const rpc::SchedulingStrategy &scheduling\_strategy,  
bool actor\_creation,  
bool force\_spillback,  
const std::string &preferred\_node\_id,  
int64\_t \*total\_violations,  
bool \*is\_infeasible) {  
if (scheduling\_strategy.scheduling\_strategy\_case() ==  
rpc::SchedulingStrategy::SchedulingStrategyCase::kSpreadSchedulingStrategy) {  
best\_node\_id = scheduling\_policy\_->Schedule(  
resource\_request,  
SchedulingOptions::Spread(force\_spillback, force\_spillback));  
} else if (scheduling\_strategy.scheduling\_strategy\_case() ==  
rpc::SchedulingStrategy::SchedulingStrategyCase::  
kNodeAffinitySchedulingStrategy) {  
best\_node\_id = scheduling\_policy\_->Schedule(  
resource\_request,  
SchedulingOptions::NodeAffinity(/\* ... \*/));  
} else if (scheduling\_strategy.has\_node\_label\_scheduling\_strategy()) {  
best\_node\_id = scheduling\_policy\_->Schedule(  
resource\_request,  
SchedulingOptions::NodeLabelScheduling(scheduling\_strategy));  
}  
}

// From src/ray/raylet/scheduling/cluster\_resource\_scheduler.cc  
scheduling::NodeID ClusterResourceScheduler::GetBestSchedulableNode(  
const ResourceRequest &resource\_request,  
const rpc::SchedulingStrategy &scheduling\_strategy,  
bool actor\_creation,  
bool force\_spillback,  
const std::string &preferred\_node\_id,  
int64\_t \*total\_violations,  
bool \*is\_infeasible) {  
if (scheduling\_strategy.scheduling\_strategy\_case() ==  
rpc::SchedulingStrategy::SchedulingStrategyCase::kSpreadSchedulingStrategy) {  
best\_node\_id = scheduling\_policy\_->Schedule(  
resource\_request,  
SchedulingOptions::Spread(force\_spillback, force\_spillback));  
} else if (scheduling\_strategy.scheduling\_strategy\_case() ==  
rpc::SchedulingStrategy::SchedulingStrategyCase::  
kNodeAffinitySchedulingStrategy) {  
best\_node\_id = scheduling\_policy\_->Schedule(  
resource\_request,  
SchedulingOptions::NodeAffinity(/\* ... \*/));  
} else if (scheduling\_strategy.has\_node\_label\_scheduling\_strategy()) {  
best\_node\_id = scheduling\_policy\_->Schedule(  
resource\_request,  
SchedulingOptions::NodeLabelScheduling(scheduling\_strategy));  
}  
}

### DEFAULT Strategy

Implementation: Hybrid policy with configurable parameters

# Environment variables controlling DEFAULT strategy  
RAY\_scheduler\_spread\_threshold = 0.5 # Utilization threshold  
RAY\_scheduler\_top\_k\_fraction = 0.2 # Top-k selection ratio  
RAY\_scheduler\_top\_k\_absolute = 5 # Minimum top-k count

# Environment variables controlling DEFAULT strategy  
RAY\_scheduler\_spread\_threshold = 0.5 # Utilization threshold  
RAY\_scheduler\_top\_k\_fraction = 0.2 # Top-k selection ratio  
RAY\_scheduler\_top\_k\_absolute = 5 # Minimum top-k count

Algorithm:  
1. Calculate node scores based on resource utilization  
2. Select top-k nodes with lowest scores  
3. Randomly choose from top-k for load balancing

### SPREAD Strategy

Purpose: Maximize distribution across nodes

import ray  
@ray.remote(scheduling\_strategy="SPREAD")  
def distributed\_task():  
return "Running on different nodes"  
futures = [distributed\_task.remote() for \_ in range(100)]

import ray  
@ray.remote(scheduling\_strategy="SPREAD")  
def distributed\_task():  
return "Running on different nodes"  
futures = [distributed\_task.remote() for \_ in range(100)]

Implementation Details:  
- Prioritizes nodes with fewer running tasks  
- Considers resource utilization as secondary factor  
- Useful for embarrassingly parallel workloads

### Node Affinity Strategy

Hard Affinity: Must run on specific node

import ray  
from ray.util.scheduling\_strategies import NodeAffinitySchedulingStrategy  
@ray.remote(scheduling\_strategy=NodeAffinitySchedulingStrategy(  
node\_id="specific-node-id",  
soft=False  
))  
def pinned\_task():  
return "Must run on specific node"

import ray  
from ray.util.scheduling\_strategies import NodeAffinitySchedulingStrategy  
@ray.remote(scheduling\_strategy=NodeAffinitySchedulingStrategy(  
node\_id="specific-node-id",  
soft=False  
))  
def pinned\_task():  
return "Must run on specific node"

Soft Affinity: Prefer specific node with fallback

@ray.remote(scheduling\_strategy=NodeAffinitySchedulingStrategy(  
node\_id="preferred-node-id",  
soft=True  
))  
def preferred\_task():  
return "Prefers specific node but can run elsewhere"

@ray.remote(scheduling\_strategy=NodeAffinitySchedulingStrategy(  
node\_id="preferred-node-id",  
soft=True  
))  
def preferred\_task():  
return "Prefers specific node but can run elsewhere"

### Placement Group Strategy

Bundle-Specific Scheduling:

import ray  
from ray.util.placement\_group import placement\_group  
from ray.util.scheduling\_strategies import PlacementGroupSchedulingStrategy  
# Create placement group  
pg = placement\_group([{"CPU": 2}, {"CPU": 2}], strategy="PACK")  
@ray.remote(scheduling\_strategy=PlacementGroupSchedulingStrategy(  
placement\_group=pg,  
placement\_group\_bundle\_index=0  
))  
def task\_on\_bundle\_0():  
return "Running on bundle 0"  
@ray.remote(scheduling\_strategy=PlacementGroupSchedulingStrategy(  
placement\_group=pg,  
placement\_group\_bundle\_index=-1 # Any bundle  
))  
def task\_on\_any\_bundle():  
return "Running on any available bundle"

import ray  
from ray.util.placement\_group import placement\_group  
from ray.util.scheduling\_strategies import PlacementGroupSchedulingStrategy  
# Create placement group  
pg = placement\_group([{"CPU": 2}, {"CPU": 2}], strategy="PACK")  
@ray.remote(scheduling\_strategy=PlacementGroupSchedulingStrategy(  
placement\_group=pg,  
placement\_group\_bundle\_index=0  
))  
def task\_on\_bundle\_0():  
return "Running on bundle 0"  
@ray.remote(scheduling\_strategy=PlacementGroupSchedulingStrategy(  
placement\_group=pg,  
placement\_group\_bundle\_index=-1 # Any bundle  
))  
def task\_on\_any\_bundle():  
return "Running on any available bundle"

## Node Affinity and Label-Based Scheduling

### Node Label Scheduling Policy

Location: src/ray/raylet/scheduling/policy/node\_label\_scheduling\_policy.cc  
Ray supports sophisticated label-based scheduling for fine-grained node selection:

src/ray/raylet/scheduling/policy/node\_label\_scheduling\_policy.cc

scheduling::NodeID NodeLabelSchedulingPolicy::Schedule(  
const ResourceRequest &resource\_request,  
SchedulingOptions options) {  
// 1. Select feasible nodes  
auto hard\_match\_nodes = SelectFeasibleNodes(resource\_request);  
// 2. Filter by hard expressions  
if (node\_label\_scheduling\_strategy.hard().expressions().size() > 0) {  
hard\_match\_nodes = FilterNodesByLabelMatchExpressions(  
hard\_match\_nodes, node\_label\_scheduling\_strategy.hard());  
}  
// 3. Filter by soft expressions  
auto hard\_and\_soft\_match\_nodes = FilterNodesByLabelMatchExpressions(  
hard\_match\_nodes, node\_label\_scheduling\_strategy.soft());  
return SelectBestNode(hard\_match\_nodes, hard\_and\_soft\_match\_nodes, resource\_request);  
}

scheduling::NodeID NodeLabelSchedulingPolicy::Schedule(  
const ResourceRequest &resource\_request,  
SchedulingOptions options) {  
// 1. Select feasible nodes  
auto hard\_match\_nodes = SelectFeasibleNodes(resource\_request);  
// 2. Filter by hard expressions  
if (node\_label\_scheduling\_strategy.hard().expressions().size() > 0) {  
hard\_match\_nodes = FilterNodesByLabelMatchExpressions(  
hard\_match\_nodes, node\_label\_scheduling\_strategy.hard());  
}  
// 3. Filter by soft expressions  
auto hard\_and\_soft\_match\_nodes = FilterNodesByLabelMatchExpressions(  
hard\_match\_nodes, node\_label\_scheduling\_strategy.soft());  
return SelectBestNode(hard\_match\_nodes, hard\_and\_soft\_match\_nodes, resource\_request);  
}

### Label Matching Implementation

bool NodeLabelSchedulingPolicy::IsNodeMatchLabelExpression(  
const Node &node, const rpc::LabelMatchExpression &expression) const {  
const auto &key = expression.key();  
const auto &operator\_type = expression.operator\_();  
const auto &values = expression.values();  
switch (operator\_type) {  
case rpc::LabelMatchExpression::IN:  
return IsNodeLabelInValues(node, key, values);  
case rpc::LabelMatchExpression::NOT\_IN:  
return !IsNodeLabelInValues(node, key, values);  
case rpc::LabelMatchExpression::EXISTS:  
return IsNodeLabelKeyExists(node, key);  
case rpc::LabelMatchExpression::DOES\_NOT\_EXIST:  
return !IsNodeLabelKeyExists(node, key);  
}  
}

bool NodeLabelSchedulingPolicy::IsNodeMatchLabelExpression(  
const Node &node, const rpc::LabelMatchExpression &expression) const {  
const auto &key = expression.key();  
const auto &operator\_type = expression.operator\_();  
const auto &values = expression.values();  
switch (operator\_type) {  
case rpc::LabelMatchExpression::IN:  
return IsNodeLabelInValues(node, key, values);  
case rpc::LabelMatchExpression::NOT\_IN:  
return !IsNodeLabelInValues(node, key, values);  
case rpc::LabelMatchExpression::EXISTS:  
return IsNodeLabelKeyExists(node, key);  
case rpc::LabelMatchExpression::DOES\_NOT\_EXIST:  
return !IsNodeLabelKeyExists(node, key);  
}  
}

### Label Selector Usage

import ray  
from ray.util.scheduling\_strategies import NodeLabelSchedulingStrategy  
# Hard constraints (must match)  
hard\_constraints = {  
"ray.io/node-type": "gpu-node",  
"zone": "us-west-1a"  
}  
# Soft constraints (preferred)  
soft\_constraints = {  
"instance-type": "p3.2xlarge"  
}  
@ray.remote(scheduling\_strategy=NodeLabelSchedulingStrategy(  
hard=hard\_constraints,  
soft=soft\_constraints  
))  
def gpu\_task():  
return "Running on GPU node in preferred zone"

import ray  
from ray.util.scheduling\_strategies import NodeLabelSchedulingStrategy  
# Hard constraints (must match)  
hard\_constraints = {  
"ray.io/node-type": "gpu-node",  
"zone": "us-west-1a"  
}  
# Soft constraints (preferred)  
soft\_constraints = {  
"instance-type": "p3.2xlarge"  
}  
@ray.remote(scheduling\_strategy=NodeLabelSchedulingStrategy(  
hard=hard\_constraints,  
soft=soft\_constraints  
))  
def gpu\_task():  
return "Running on GPU node in preferred zone"

### Node Label Management

Static Labels: Set during node startup

# Set node labels via environment  
export RAY\_NODE\_LABELS='{"zone":"us-west-1a","instance-type":"m5.large"}'  
ray start --head

# Set node labels via environment  
export RAY\_NODE\_LABELS='{"zone":"us-west-1a","instance-type":"m5.large"}'  
ray start --head

Dynamic Labels: Updated at runtime

// From cluster resource data  
struct NodeResources {  
absl::flat\_hash\_map<std::string, std::string> labels;  
bool HasRequiredLabels(const LabelSelector &label\_selector) const;  
bool NodeLabelMatchesConstraint(const LabelConstraint &constraint) const;  
};

// From cluster resource data  
struct NodeResources {  
absl::flat\_hash\_map<std::string, std::string> labels;  
bool HasRequiredLabels(const LabelSelector &label\_selector) const;  
bool NodeLabelMatchesConstraint(const LabelConstraint &constraint) const;  
};

## Locality-Aware Scheduling

### Locality-Aware Lease Policy

Location: src/ray/core\_worker/lease\_policy.cc  
Ray implements data locality-aware scheduling to minimize data movement:

src/ray/core\_worker/lease\_policy.cc

std::pair<rpc::Address, bool> LocalityAwareLeasePolicy::GetBestNodeForTask(  
const TaskSpecification &spec) {  
// Check for explicit scheduling strategies first  
if (spec.IsSpreadSchedulingStrategy() || spec.IsNodeAffinitySchedulingStrategy()) {  
return std::make\_pair(fallback\_rpc\_address\_, false);  
}  
// Pick node based on locality  
if (auto node\_id = GetBestNodeIdForTask(spec)) {  
if (auto addr = node\_addr\_factory\_(node\_id.value())) {  
return std::make\_pair(addr.value(), true);  
}  
}  
return std::make\_pair(fallback\_rpc\_address\_, false);  
}

std::pair<rpc::Address, bool> LocalityAwareLeasePolicy::GetBestNodeForTask(  
const TaskSpecification &spec) {  
// Check for explicit scheduling strategies first  
if (spec.IsSpreadSchedulingStrategy() || spec.IsNodeAffinitySchedulingStrategy()) {  
return std::make\_pair(fallback\_rpc\_address\_, false);  
}  
// Pick node based on locality  
if (auto node\_id = GetBestNodeIdForTask(spec)) {  
if (auto addr = node\_addr\_factory\_(node\_id.value())) {  
return std::make\_pair(addr.value(), true);  
}  
}  
return std::make\_pair(fallback\_rpc\_address\_, false);  
}

### Locality Calculation

Criteria: Node with most object bytes local

std::optional<NodeID> LocalityAwareLeasePolicy::GetBestNodeIdForTask(  
const TaskSpecification &spec) {  
const auto &dependencies = spec.GetDependencies();  
if (dependencies.empty()) {  
return std::nullopt;  
}  
// Calculate locality scores for each node  
absl::flat\_hash\_map<NodeID, int64\_t> locality\_scores;  
for (const auto &obj\_id : dependencies) {  
auto locality\_data = locality\_data\_provider\_.GetLocalityData(obj\_id);  
for (const auto &node\_id : locality\_data.nodes\_containing\_object) {  
locality\_scores[node\_id] += locality\_data.object\_size;  
}  
}  
// Return node with highest locality score  
return GetNodeWithMaxScore(locality\_scores);  
}

std::optional<NodeID> LocalityAwareLeasePolicy::GetBestNodeIdForTask(  
const TaskSpecification &spec) {  
const auto &dependencies = spec.GetDependencies();  
if (dependencies.empty()) {  
return std::nullopt;  
}  
// Calculate locality scores for each node  
absl::flat\_hash\_map<NodeID, int64\_t> locality\_scores;  
for (const auto &obj\_id : dependencies) {  
auto locality\_data = locality\_data\_provider\_.GetLocalityData(obj\_id);  
for (const auto &node\_id : locality\_data.nodes\_containing\_object) {  
locality\_scores[node\_id] += locality\_data.object\_size;  
}  
}  
// Return node with highest locality score  
return GetNodeWithMaxScore(locality\_scores);  
}

### Locality vs Strategy Priority

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Locality Testing

// From src/ray/tests/test\_scheduling.py  
def test\_locality\_aware\_leasing(ray\_start\_cluster):  
@ray.remote(resources={"pin": 1})  
def non\_local():  
return ray.\_private.worker.global\_worker.node.unique\_id  
@ray.remote  
def f(x):  
return ray.\_private.worker.global\_worker.node.unique\_id  
# Test that task f() runs on the same node as non\_local()  
# due to data locality  
assert ray.get(f.remote(non\_local.remote())) == non\_local\_node.unique\_id

// From src/ray/tests/test\_scheduling.py  
def test\_locality\_aware\_leasing(ray\_start\_cluster):  
@ray.remote(resources={"pin": 1})  
def non\_local():  
return ray.\_private.worker.global\_worker.node.unique\_id  
@ray.remote  
def f(x):  
return ray.\_private.worker.global\_worker.node.unique\_id  
# Test that task f() runs on the same node as non\_local()  
# due to data locality  
assert ray.get(f.remote(non\_local.remote())) == non\_local\_node.unique\_id

## Cluster Resource Scheduling

### Cluster Resource Manager

Location: src/ray/raylet/scheduling/cluster\_resource\_manager.h  
Maintains global view of cluster resources:

src/ray/raylet/scheduling/cluster\_resource\_manager.h

class ClusterResourceManager {  
// Add or update node resources  
void AddOrUpdateNode(scheduling::NodeID node\_id,  
const NodeResources &node\_resources);  
// Check resource availability  
bool HasAvailableResources(scheduling::NodeID node\_id,  
const ResourceRequest &resource\_request) const;  
// Resource allocation  
bool SubtractNodeAvailableResources(scheduling::NodeID node\_id,  
const ResourceRequest &resource\_request);  
};

class ClusterResourceManager {  
// Add or update node resources  
void AddOrUpdateNode(scheduling::NodeID node\_id,  
const NodeResources &node\_resources);  
// Check resource availability  
bool HasAvailableResources(scheduling::NodeID node\_id,  
const ResourceRequest &resource\_request) const;  
// Resource allocation  
bool SubtractNodeAvailableResources(scheduling::NodeID node\_id,  
const ResourceRequest &resource\_request);  
};

### Resource Synchronization

📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

### Resource Reporting

Location: src/ray/raylet/scheduling/scheduler\_resource\_reporter.cc

src/ray/raylet/scheduling/scheduler\_resource\_reporter.cc

void SchedulerResourceReporter::FillResourceUsage(rpc::ResourcesData &data) const {  
// Report resource demands by shape  
auto resource\_load\_by\_shape = data.mutable\_resource\_load\_by\_shape();  
for (const auto &[scheduling\_class, task\_queue] : tasks\_to\_schedule\_) {  
const auto &resources = scheduling\_class\_descriptor.resource\_set.GetResourceMap();  
auto by\_shape\_entry = resource\_load\_by\_shape->Add();  
for (const auto &resource : resources) {  
(\*by\_shape\_entry->mutable\_shape())[resource.first] = resource.second;  
}  
by\_shape\_entry->set\_num\_ready\_requests\_queued(task\_queue.size());  
}  
}

void SchedulerResourceReporter::FillResourceUsage(rpc::ResourcesData &data) const {  
// Report resource demands by shape  
auto resource\_load\_by\_shape = data.mutable\_resource\_load\_by\_shape();  
for (const auto &[scheduling\_class, task\_queue] : tasks\_to\_schedule\_) {  
const auto &resources = scheduling\_class\_descriptor.resource\_set.GetResourceMap();  
auto by\_shape\_entry = resource\_load\_by\_shape->Add();  
for (const auto &resource : resources) {  
(\*by\_shape\_entry->mutable\_shape())[resource.first] = resource.second;  
}  
by\_shape\_entry->set\_num\_ready\_requests\_queued(task\_queue.size());  
}  
}

## Autoscaler Integration

### Resource Demand Scheduler

Location: python/ray/autoscaler/v2/scheduler.py  
The autoscaler uses sophisticated scheduling algorithms to determine cluster scaling decisions:

python/ray/autoscaler/v2/scheduler.py

class ResourceDemandScheduler(IResourceScheduler):  
def schedule(self, request: SchedulingRequest) -> SchedulingReply:  
ctx = self.ScheduleContext.from\_schedule\_request(request)  
# 1. Enforce min workers per type  
self.\_enforce\_min\_workers\_per\_type(ctx)  
# 2. Enforce resource constraints  
infeasible\_constraints = self.\_enforce\_resource\_constraints(  
ctx, request.cluster\_resource\_constraints)  
# 3. Schedule gang resource requests  
infeasible\_gang\_requests = self.\_sched\_gang\_resource\_requests(  
ctx, request.gang\_resource\_requests)  
# 4. Schedule regular resource requests  
infeasible\_requests = self.\_sched\_resource\_requests(  
ctx, ResourceRequestUtil.ungroup\_by\_count(request.resource\_requests))  
# 5. Enforce idle termination  
self.\_enforce\_idle\_termination(ctx)  
return SchedulingReply(  
to\_launch=ctx.get\_launch\_requests(),  
to\_terminate=ctx.get\_terminate\_requests(),  
infeasible\_resource\_requests=infeasible\_requests,  
infeasible\_gang\_resource\_requests=infeasible\_gang\_requests,  
infeasible\_cluster\_resource\_constraints=infeasible\_constraints  
)

class ResourceDemandScheduler(IResourceScheduler):  
def schedule(self, request: SchedulingRequest) -> SchedulingReply:  
ctx = self.ScheduleContext.from\_schedule\_request(request)  
# 1. Enforce min workers per type  
self.\_enforce\_min\_workers\_per\_type(ctx)  
# 2. Enforce resource constraints  
infeasible\_constraints = self.\_enforce\_resource\_constraints(  
ctx, request.cluster\_resource\_constraints)  
# 3. Schedule gang resource requests  
infeasible\_gang\_requests = self.\_sched\_gang\_resource\_requests(  
ctx, request.gang\_resource\_requests)  
# 4. Schedule regular resource requests  
infeasible\_requests = self.\_sched\_resource\_requests(  
ctx, ResourceRequestUtil.ungroup\_by\_count(request.resource\_requests))  
# 5. Enforce idle termination  
self.\_enforce\_idle\_termination(ctx)  
return SchedulingReply(  
to\_launch=ctx.get\_launch\_requests(),  
to\_terminate=ctx.get\_terminate\_requests(),  
infeasible\_resource\_requests=infeasible\_requests,  
infeasible\_gang\_resource\_requests=infeasible\_gang\_requests,  
infeasible\_cluster\_resource\_constraints=infeasible\_constraints  
)

### Binpacking Algorithm

def \_try\_schedule(  
ctx: ScheduleContext,  
requests\_to\_sched: List[ResourceRequest],  
resource\_request\_source: ResourceRequestSource,  
) -> Tuple[List[SchedulingNode], List[ResourceRequest]]:  
# Sort requests by complexity for better binpacking  
def \_sort\_resource\_request(req: ResourceRequest) -> Tuple:  
return (  
len(req.placement\_constraints),  
len(req.resources\_bundle.values()),  
sum(req.resources\_bundle.values()),  
sorted(req.resources\_bundle.items()),  
)  
requests\_to\_sched = sorted(  
requests\_to\_sched, key=\_sort\_resource\_request, reverse=True)  
while len(requests\_to\_sched) > 0 and len(existing\_nodes) > 0:  
best\_node, requests\_to\_sched, existing\_nodes = \  
self.\_sched\_best\_node(requests\_to\_sched, existing\_nodes, resource\_request\_source)  
if best\_node is None:  
break  
target\_nodes.append(best\_node)  
for node\_type, num\_available in node\_type\_available.items():  
if num\_available > 0:  
new\_node = SchedulingNode.from\_node\_config(  
ctx.get\_node\_type\_configs()[node\_type],  
status=SchedulingNodeStatus.TO\_LAUNCH)  
# Try to schedule remaining requests on new node

def \_try\_schedule(  
ctx: ScheduleContext,  
requests\_to\_sched: List[ResourceRequest],  
resource\_request\_source: ResourceRequestSource,  
) -> Tuple[List[SchedulingNode], List[ResourceRequest]]:  
# Sort requests by complexity for better binpacking  
def \_sort\_resource\_request(req: ResourceRequest) -> Tuple:  
return (  
len(req.placement\_constraints),  
len(req.resources\_bundle.values()),  
sum(req.resources\_bundle.values()),  
sorted(req.resources\_bundle.items()),  
)  
requests\_to\_sched = sorted(  
requests\_to\_sched, key=\_sort\_resource\_request, reverse=True)  
while len(requests\_to\_sched) > 0 and len(existing\_nodes) > 0:  
best\_node, requests\_to\_sched, existing\_nodes = \  
self.\_sched\_best\_node(requests\_to\_sched, existing\_nodes, resource\_request\_source)  
if best\_node is None:  
break  
target\_nodes.append(best\_node)  
for node\_type, num\_available in node\_type\_available.items():  
if num\_available > 0:  
new\_node = SchedulingNode.from\_node\_config(  
ctx.get\_node\_type\_configs()[node\_type],  
status=SchedulingNodeStatus.TO\_LAUNCH)  
# Try to schedule remaining requests on new node

### Placement Group Autoscaling

def placement\_groups\_to\_resource\_demands(  
pending\_placement\_groups: List[PlacementGroupTableData],  
) -> Tuple[List[ResourceDict], List[List[ResourceDict]]]:  
resource\_demand\_vector = []  
unconverted = []  
for placement\_group in pending\_placement\_groups:  
shapes = [dict(bundle.unit\_resources) for bundle in placement\_group.bundles  
if bundle.node\_id == b""] # Only unplaced bundles  
if placement\_group.strategy == PlacementStrategy.PACK:  
resource\_demand\_vector.extend(shapes)  
elif placement\_group.strategy == PlacementStrategy.STRICT\_PACK:  
# Combine all bundles into single demand  
combined = collections.defaultdict(float)  
for shape in shapes:  
for label, quantity in shape.items():  
combined[label] += quantity  
resource\_demand\_vector.append(combined)  
elif placement\_group.strategy == PlacementStrategy.STRICT\_SPREAD:  
# Cannot be converted - needs special handling  
unconverted.append(shapes)  
return resource\_demand\_vector, unconverted

def placement\_groups\_to\_resource\_demands(  
pending\_placement\_groups: List[PlacementGroupTableData],  
) -> Tuple[List[ResourceDict], List[List[ResourceDict]]]:  
resource\_demand\_vector = []  
unconverted = []  
for placement\_group in pending\_placement\_groups:  
shapes = [dict(bundle.unit\_resources) for bundle in placement\_group.bundles  
if bundle.node\_id == b""] # Only unplaced bundles  
if placement\_group.strategy == PlacementStrategy.PACK:  
resource\_demand\_vector.extend(shapes)  
elif placement\_group.strategy == PlacementStrategy.STRICT\_PACK:  
# Combine all bundles into single demand  
combined = collections.defaultdict(float)  
for shape in shapes:  
for label, quantity in shape.items():  
combined[label] += quantity  
resource\_demand\_vector.append(combined)  
elif placement\_group.strategy == PlacementStrategy.STRICT\_SPREAD:  
# Cannot be converted - needs special handling  
unconverted.append(shapes)  
return resource\_demand\_vector, unconverted

### Autoscaler Configuration

# Example autoscaler configuration  
cluster\_name: ray-cluster  
max\_workers: 100  
upscaling\_speed: 1.0  
idle\_timeout\_minutes: 5  
available\_node\_types:  
ray.head.default:  
min\_workers: 0  
max\_workers: 0  
resources: {"CPU": 4}  
ray.worker.cpu:  
min\_workers: 0  
max\_workers: 50  
resources: {"CPU": 8, "memory": 32000000000}  
ray.worker.gpu:  
min\_workers: 0  
max\_workers: 10  
resources: {"CPU": 16, "GPU": 4, "memory": 64000000000}

# Example autoscaler configuration  
cluster\_name: ray-cluster  
max\_workers: 100  
upscaling\_speed: 1.0  
idle\_timeout\_minutes: 5  
available\_node\_types:  
ray.head.default:  
min\_workers: 0  
max\_workers: 0  
resources: {"CPU": 4}  
ray.worker.cpu:  
min\_workers: 0  
max\_workers: 50  
resources: {"CPU": 8, "memory": 32000000000}  
ray.worker.gpu:  
min\_workers: 0  
max\_workers: 10  
resources: {"CPU": 16, "GPU": 4, "memory": 64000000000}

## Performance Characteristics

### Scheduling Latency

Typical Latencies:  
- Local scheduling: 1-5ms  
- Remote scheduling: 10-50ms  
- Placement group creation: 100-1000ms  
- Autoscaler response: 30-300s

### Scalability Metrics

Cluster Size: Ray scheduling tested up to 1000+ nodes  
Task Throughput:  
- Simple tasks: 100K+ tasks/second  
- Complex scheduling: 10K+ tasks/second  
- Placement groups: 100+ groups/second

### Memory Usage

Scheduler Memory Overhead:

// Per-node overhead in ClusterResourceManager  
struct NodeResources {  
NodeResourceSet total; // ~1KB per node  
NodeResourceSet available; // ~1KB per node  
NodeResourceSet normal\_task\_resources; // ~1KB per node  
absl::flat\_hash\_map<std::string, std::string> labels; // Variable  
};  
// Total: ~3KB + labels per node

// Per-node overhead in ClusterResourceManager  
struct NodeResources {  
NodeResourceSet total; // ~1KB per node  
NodeResourceSet available; // ~1KB per node  
NodeResourceSet normal\_task\_resources; // ~1KB per node  
absl::flat\_hash\_map<std::string, std::string> labels; // Variable  
};  
// Total: ~3KB + labels per node

Task Queue Memory:

// Per-task overhead in scheduling queues  
class Work {  
RayTask task; // ~2KB per task  
TaskResourceInstances allocated; // ~500B per task  
WorkStatus state; // ~100B per task  
};  
// Total: ~2.6KB per queued task

// Per-task overhead in scheduling queues  
class Work {  
RayTask task; // ~2KB per task  
TaskResourceInstances allocated; // ~500B per task  
WorkStatus state; // ~100B per task  
};  
// Total: ~2.6KB per queued task

### Performance Optimization

Top-K Selection: Reduces scheduling complexity from O(N) to O(K)

// Default configuration  
RAY\_scheduler\_top\_k\_fraction = 0.2 // 20% of nodes  
RAY\_scheduler\_top\_k\_absolute = 5 // Minimum 5 nodes

// Default configuration  
RAY\_scheduler\_top\_k\_fraction = 0.2 // 20% of nodes  
RAY\_scheduler\_top\_k\_absolute = 5 // Minimum 5 nodes

Caching: Resource views cached to avoid repeated calculations

class ClusterResourceManager {  
// Cached resource calculations  
mutable absl::flat\_hash\_map<scheduling::NodeID, float> utilization\_cache\_;  
mutable int64\_t cache\_timestamp\_;  
};

class ClusterResourceManager {  
// Cached resource calculations  
mutable absl::flat\_hash\_map<scheduling::NodeID, float> utilization\_cache\_;  
mutable int64\_t cache\_timestamp\_;  
};

## Configuration and Tuning

### Environment Variables

Core Scheduling:

export RAY\_scheduler\_spread\_threshold=0.5  
# Top-k node selection  
export RAY\_scheduler\_top\_k\_fraction=0.2  
export RAY\_scheduler\_top\_k\_absolute=5  
# Worker management  
export RAY\_num\_workers\_soft\_limit=1000  
export RAY\_maximum\_startup\_concurrency=10

export RAY\_scheduler\_spread\_threshold=0.5  
# Top-k node selection  
export RAY\_scheduler\_top\_k\_fraction=0.2  
export RAY\_scheduler\_top\_k\_absolute=5  
# Worker management  
export RAY\_num\_workers\_soft\_limit=1000  
export RAY\_maximum\_startup\_concurrency=10

Resource Management:

export RAY\_object\_store\_memory=1000000000  
# Pull manager configuration  
export RAY\_object\_manager\_pull\_timeout\_ms=10000  
export RAY\_object\_manager\_max\_bytes\_in\_flight=100000000

export RAY\_object\_store\_memory=1000000000  
# Pull manager configuration  
export RAY\_object\_manager\_pull\_timeout\_ms=10000  
export RAY\_object\_manager\_max\_bytes\_in\_flight=100000000

Placement Groups:

# CPU fraction limits  
export RAY\_placement\_group\_max\_cpu\_fraction\_per\_node=0.8  
export RAY\_placement\_group\_bundle\_resource\_timeout\_s=30

# CPU fraction limits  
export RAY\_placement\_group\_max\_cpu\_fraction\_per\_node=0.8  
export RAY\_placement\_group\_bundle\_resource\_timeout\_s=30

### Runtime Configuration

Cluster Resource Constraints:

import ray  
# Set cluster-wide resource constraints  
ray.autoscaler.sdk.request\_resources([  
{"CPU": 100, "GPU": 10}, # Ensure cluster can handle this workload  
{"memory": 1000000000} # Minimum memory requirement  
])

import ray  
# Set cluster-wide resource constraints  
ray.autoscaler.sdk.request\_resources([  
{"CPU": 100, "GPU": 10}, # Ensure cluster can handle this workload  
{"memory": 1000000000} # Minimum memory requirement  
])

Node Type Configuration:

# Configure node types for autoscaling  
node\_config = {  
"ray.worker.cpu": {  
"min\_workers": 2,  
"max\_workers": 20,  
"resources": {"CPU": 8, "memory": 32000000000}  
},  
"ray.worker.gpu": {  
"min\_workers": 0,  
"max\_workers": 5,  
"resources": {"CPU": 16, "GPU": 4, "memory": 64000000000}  
}  
}

# Configure node types for autoscaling  
node\_config = {  
"ray.worker.cpu": {  
"min\_workers": 2,  
"max\_workers": 20,  
"resources": {"CPU": 8, "memory": 32000000000}  
},  
"ray.worker.gpu": {  
"min\_workers": 0,  
"max\_workers": 5,  
"resources": {"CPU": 16, "GPU": 4, "memory": 64000000000}  
}  
}

### Performance Tuning

For High Throughput:

# Increase worker limits  
export RAY\_num\_workers\_soft\_limit=2000  
export RAY\_maximum\_startup\_concurrency=50  
export RAY\_scheduler\_top\_k\_absolute=10  
export RAY\_scheduler\_spread\_threshold=0.3

# Increase worker limits  
export RAY\_num\_workers\_soft\_limit=2000  
export RAY\_maximum\_startup\_concurrency=50  
export RAY\_scheduler\_top\_k\_absolute=10  
export RAY\_scheduler\_spread\_threshold=0.3

For Low Latency:

export RAY\_scheduler\_spread\_threshold=0.8  
export RAY\_scheduler\_top\_k\_fraction=0.1  
# Reduce worker startup time  
export RAY\_worker\_lease\_timeout\_milliseconds=1000

export RAY\_scheduler\_spread\_threshold=0.8  
export RAY\_scheduler\_top\_k\_fraction=0.1  
# Reduce worker startup time  
export RAY\_worker\_lease\_timeout\_milliseconds=1000

For Large Clusters:

# Optimize for scale  
export RAY\_scheduler\_top\_k\_fraction=0.1 # Top 10% of nodes  
export RAY\_raylet\_report\_resources\_period\_milliseconds=1000  
export RAY\_gcs\_resource\_report\_poll\_period\_milliseconds=1000

# Optimize for scale  
export RAY\_scheduler\_top\_k\_fraction=0.1 # Top 10% of nodes  
export RAY\_raylet\_report\_resources\_period\_milliseconds=1000  
export RAY\_gcs\_resource\_report\_poll\_period\_milliseconds=1000

## Best Practices

### Task Scheduling

1. Use Appropriate Scheduling Strategies:

# For embarrassingly parallel workloads  
@ray.remote(scheduling\_strategy="SPREAD")  
def parallel\_task(data):  
return process(data)  
# For data-dependent tasks (default locality-aware)  
@ray.remote  
def dependent\_task(large\_object):  
return analyze(large\_object)  
# For specific hardware requirements  
@ray.remote(scheduling\_strategy=NodeAffinitySchedulingStrategy(  
node\_id=gpu\_node\_id, soft=True))  
def gpu\_task():  
return train\_model()

# For embarrassingly parallel workloads  
@ray.remote(scheduling\_strategy="SPREAD")  
def parallel\_task(data):  
return process(data)  
# For data-dependent tasks (default locality-aware)  
@ray.remote  
def dependent\_task(large\_object):  
return analyze(large\_object)  
# For specific hardware requirements  
@ray.remote(scheduling\_strategy=NodeAffinitySchedulingStrategy(  
node\_id=gpu\_node\_id, soft=True))  
def gpu\_task():  
return train\_model()

2. Resource Specification:

# Be specific about resource requirements  
@ray.remote(num\_cpus=2, num\_gpus=1, memory=4000\*1024\*1024)  
def resource\_intensive\_task():  
return compute()  
# Use custom resources for specialized hardware  
@ray.remote(resources={"accelerator": 1})  
def accelerated\_task():  
return specialized\_compute()

# Be specific about resource requirements  
@ray.remote(num\_cpus=2, num\_gpus=1, memory=4000\*1024\*1024)  
def resource\_intensive\_task():  
return compute()  
# Use custom resources for specialized hardware  
@ray.remote(resources={"accelerator": 1})  
def accelerated\_task():  
return specialized\_compute()

### Actor Placement

1. Consider Resource Lifetime:

# Actors hold resources for their lifetime  
@ray.remote(num\_cpus=4, num\_gpus=1)  
class ModelServer:  
def \_\_init\_\_(self):  
self.model = load\_large\_model()  
def predict(self, data):  
return self.model.predict(data)  
# Create fewer, long-lived actors rather than many short-lived ones  
server = ModelServer.remote()

# Actors hold resources for their lifetime  
@ray.remote(num\_cpus=4, num\_gpus=1)  
class ModelServer:  
def \_\_init\_\_(self):  
self.model = load\_large\_model()  
def predict(self, data):  
return self.model.predict(data)  
# Create fewer, long-lived actors rather than many short-lived ones  
server = ModelServer.remote()

2. Use Placement Groups for Related Actors:

# Group related actors together  
pg = placement\_group([{"CPU": 4}, {"CPU": 4}, {"CPU": 4}], strategy="PACK")  
actors = [  
Actor.options(scheduling\_strategy=PlacementGroupSchedulingStrategy(  
placement\_group=pg, placement\_group\_bundle\_index=i  
)).remote() for i in range(3)  
]

# Group related actors together  
pg = placement\_group([{"CPU": 4}, {"CPU": 4}, {"CPU": 4}], strategy="PACK")  
actors = [  
Actor.options(scheduling\_strategy=PlacementGroupSchedulingStrategy(  
placement\_group=pg, placement\_group\_bundle\_index=i  
)).remote() for i in range(3)  
]

### Placement Group Design

1. Choose Appropriate Strategies:

# For tightly coupled workloads  
pg\_pack = placement\_group([{"CPU": 2, "GPU": 1}] \* 4, strategy="PACK")  
# For fault tolerance  
pg\_spread = placement\_group([{"CPU": 2}] \* 8, strategy="SPREAD")  
# For strict requirements  
pg\_strict = placement\_group([{"CPU": 4}] \* 2, strategy="STRICT\_SPREAD")

# For tightly coupled workloads  
pg\_pack = placement\_group([{"CPU": 2, "GPU": 1}] \* 4, strategy="PACK")  
# For fault tolerance  
pg\_spread = placement\_group([{"CPU": 2}] \* 8, strategy="SPREAD")  
# For strict requirements  
pg\_strict = placement\_group([{"CPU": 4}] \* 2, strategy="STRICT\_SPREAD")

2. Bundle Size Optimization:

# Avoid bundles larger than single node capacity  
# Bad: Bundle requires more than any node has  
bad\_pg = placement\_group([{"CPU": 64, "GPU": 8}]) # If max node has 32 CPU  
# Good: Bundle fits on available nodes  
good\_pg = placement\_group([{"CPU": 16, "GPU": 2}] \* 4)

# Avoid bundles larger than single node capacity  
# Bad: Bundle requires more than any node has  
bad\_pg = placement\_group([{"CPU": 64, "GPU": 8}]) # If max node has 32 CPU  
# Good: Bundle fits on available nodes  
good\_pg = placement\_group([{"CPU": 16, "GPU": 2}] \* 4)

### Autoscaler Optimization

1. Configure Appropriate Limits:

# Set realistic min/max workers  
available\_node\_types:  
ray.worker.default:  
min\_workers: 2 # Always keep some capacity  
max\_workers: 100 # Prevent runaway scaling  
upscaling\_speed: 2.0 # Scale up aggressively

# Set realistic min/max workers  
available\_node\_types:  
ray.worker.default:  
min\_workers: 2 # Always keep some capacity  
max\_workers: 100 # Prevent runaway scaling  
upscaling\_speed: 2.0 # Scale up aggressively

2. Use Resource Constraints:

# Ensure cluster can handle expected workload  
ray.autoscaler.sdk.request\_resources([  
{"CPU": 200, "memory": 500000000000}, # Expected peak usage  
])

# Ensure cluster can handle expected workload  
ray.autoscaler.sdk.request\_resources([  
{"CPU": 200, "memory": 500000000000}, # Expected peak usage  
])

## Troubleshooting

### Common Scheduling Issues

1. Tasks Stuck in Pending State:  
Symptoms: Tasks remain in PENDING\_SCHEDULING state  
Causes:  
- Insufficient cluster resources  
- Infeasible resource requirements  
- Node affinity to unavailable nodes  
Debugging:

# Check cluster resources  
print(ray.cluster\_resources())  
print(ray.available\_resources())  
# Check task resource requirements  
@ray.remote(num\_cpus=1)  
def debug\_task():  
return ray.get\_runtime\_context().get\_assigned\_resources()  
# Check for infeasible tasks  
ray.autoscaler.sdk.request\_resources([{"CPU": 1000}]) # Will show if infeasible

# Check cluster resources  
print(ray.cluster\_resources())  
print(ray.available\_resources())  
# Check task resource requirements  
@ray.remote(num\_cpus=1)  
def debug\_task():  
return ray.get\_runtime\_context().get\_assigned\_resources()  
# Check for infeasible tasks  
ray.autoscaler.sdk.request\_resources([{"CPU": 1000}]) # Will show if infeasible

2. Poor Load Balancing:  
Symptoms: Some nodes overloaded while others idle  
Causes:  
- Inappropriate scheduling strategy  
- Data locality overriding load balancing  
- Sticky worker assignment  
Solutions:

# Use SPREAD strategy for better distribution  
@ray.remote(scheduling\_strategy="SPREAD")  
def distributed\_task():  
return compute()  
# Adjust spread threshold  
import os  
os.environ["RAY\_scheduler\_spread\_threshold"] = "0.3"

# Use SPREAD strategy for better distribution  
@ray.remote(scheduling\_strategy="SPREAD")  
def distributed\_task():  
return compute()  
# Adjust spread threshold  
import os  
os.environ["RAY\_scheduler\_spread\_threshold"] = "0.3"

3. Placement Group Creation Failures:  
Symptoms: Placement groups fail to create or timeout  
Causes:  
- Insufficient cluster capacity  
- Conflicting resource constraints  
- Network partitions  
Debugging:

import ray  
from ray.util.placement\_group import placement\_group  
# Check placement group status  
pg = placement\_group([{"CPU": 2}] \* 4, strategy="STRICT\_SPREAD")  
print(pg.ready()) # False if creation failed  
# Check bundle placement  
print(ray.util.placement\_group\_table())

import ray  
from ray.util.placement\_group import placement\_group  
# Check placement group status  
pg = placement\_group([{"CPU": 2}] \* 4, strategy="STRICT\_SPREAD")  
print(pg.ready()) # False if creation failed  
# Check bundle placement  
print(ray.util.placement\_group\_table())

### Performance Issues

1. High Scheduling Latency:  
Symptoms: Long delays between task submission and execution  
Causes:  
- Large cluster with inefficient node selection  
- Complex placement constraints  
- Resource fragmentation  
Solutions:

# Reduce top-k selection size  
export RAY\_scheduler\_top\_k\_fraction=0.1  
export RAY\_scheduler\_spread\_threshold=0.7

# Reduce top-k selection size  
export RAY\_scheduler\_top\_k\_fraction=0.1  
export RAY\_scheduler\_spread\_threshold=0.7

2. Memory Issues in Scheduler:  
Symptoms: Raylet OOM, high memory usage in scheduling components  
Causes:  
- Large number of queued tasks  
- Memory leaks in scheduling data structures  
- Excessive resource tracking overhead  
Solutions:

# Limit concurrent tasks  
export RAY\_num\_workers\_soft\_limit=500  
# Reduce resource reporting frequency  
export RAY\_raylet\_report\_resources\_period\_milliseconds=5000

# Limit concurrent tasks  
export RAY\_num\_workers\_soft\_limit=500  
# Reduce resource reporting frequency  
export RAY\_raylet\_report\_resources\_period\_milliseconds=5000

### Debugging Tools

1. Ray Status Commands:

# Check cluster state  
ray status  
# Check resource usage  
ray status --verbose  
# Check placement groups  
ray status --placement-groups

# Check cluster state  
ray status  
# Check resource usage  
ray status --verbose  
# Check placement groups  
ray status --placement-groups

2. Programmatic Debugging:

import ray.\_private.state as state  
# Get pending tasks  
pending\_tasks = state.tasks(filters=[("state", "=", "PENDING\_SCHEDULING")])  
# Get resource usage by node  
nodes = state.nodes()  
for node in nodes:  
print(f"Node {node['node\_id']}: {node['resources\_total']}")

import ray.\_private.state as state  
# Get pending tasks  
pending\_tasks = state.tasks(filters=[("state", "=", "PENDING\_SCHEDULING")])  
# Get resource usage by node  
nodes = state.nodes()  
for node in nodes:  
print(f"Node {node['node\_id']}: {node['resources\_total']}")

3. Logging Configuration:

export RAY\_LOG\_LEVEL=DEBUG  
export RAY\_BACKEND\_LOG\_LEVEL=DEBUG  
# Focus on specific components  
export RAY\_LOG\_TO\_STDERR=1  
ray start --head --log-to-driver

export RAY\_LOG\_LEVEL=DEBUG  
export RAY\_BACKEND\_LOG\_LEVEL=DEBUG  
# Focus on specific components  
export RAY\_LOG\_TO\_STDERR=1  
ray start --head --log-to-driver

### Monitoring and Observability

1. Metrics Collection:

import ray  
from ray.util.metrics import Counter, Histogram  
scheduling\_latency = Histogram(  
"ray\_scheduling\_latency\_seconds",  
description="Time from task submission to scheduling",  
boundaries=[0.001, 0.01, 0.1, 1.0, 10.0]  
)  
task\_queue\_size = Counter(  
"ray\_task\_queue\_size",  
description="Number of tasks in scheduling queue"  
)

import ray  
from ray.util.metrics import Counter, Histogram  
scheduling\_latency = Histogram(  
"ray\_scheduling\_latency\_seconds",  
description="Time from task submission to scheduling",  
boundaries=[0.001, 0.01, 0.1, 1.0, 10.0]  
)  
task\_queue\_size = Counter(  
"ray\_task\_queue\_size",  
description="Number of tasks in scheduling queue"  
)

2. Dashboard Integration:  
- Use Ray Dashboard for real-time cluster monitoring  
- Monitor resource utilization trends  
- Track placement group creation success rates  
- Observe task scheduling patterns  
This comprehensive guide covers Ray's distributed scheduling system from architecture to implementation details, providing developers and operators with the knowledge needed to effectively use and optimize Ray's scheduling capabilities in production environments.

# Chapter 10: Autoscaling System

## Table of Contents

Introduction

Autoscaling Architecture Overview

Core Autoscaling Components

Resource Demand Detection

Node Lifecycle Management

Scheduling and Binpacking Algorithms

Cloud Provider Integration

Autoscaler Policies and Strategies

Load Metrics and Monitoring

Placement Group Autoscaling

Resource Constraints and Limits

Multi-Cloud and Hybrid Deployments

Performance Optimization

Configuration and Tuning

Production Deployment

Troubleshooting and Debugging

Best Practices

Advanced Topics

## Introduction

Ray's autoscaling system is like having a smart assistant that watches your computing workload and automatically adjusts your cluster size. When you have more work to do, it adds more machines. When things quiet down, it removes unused machines to save money. Think of it as an intelligent resource manager that ensures you always have just the right amount of computing power for your needs.

### What Makes Ray Autoscaling Special?

Smart Decision Making: Unlike simple autoscalers that just count CPU usage, Ray's autoscaler understands the specific resources your tasks need - CPUs, GPUs, memory, and custom resources. It can predict exactly what type of machines you need before you run out of capacity.  
Lightning Fast: The autoscaler can make scaling decisions in seconds, not minutes. It doesn't wait for machines to become overloaded - it anticipates demand and scales proactively.  
Cost Efficient: By understanding your workload patterns, it minimizes cloud costs by spinning up the cheapest combination of machines that can handle your work.  
Multi-Cloud Ready: Works seamlessly across AWS, GCP, Azure, Kubernetes, and even your local data center.

### Core Features

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships  
- Resource-Aware Scaling: Understands your exact compute needs (CPU, GPU, memory)  
- Placement Group Support: Handles complex multi-node workloads that need specific arrangements  
- Intelligent Binpacking: Finds the most cost-effective way to fit your workload  
- Preemptible Instance Support: Uses cheaper spot/preemptible instances when appropriate  
- Custom Resource Types: Supports specialized hardware like TPUs, FPGAs, or custom accelerators

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

## Autoscaling Architecture Overview

Think of Ray's autoscaling system as a well-orchestrated team where each component has a specific job, but they all work together seamlessly.

### The Big Picture: How It All Works Together

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### What Happens During Autoscaling (In Plain English)

👀 Watching Phase: The system continuously monitors your cluster, tracking how many tasks are waiting, what resources they need, and how busy each machine is.

🤔 Thinking Phase: When it notices unmet demand, the autoscaler calculates the optimal mix of machines to add, considering costs, availability, and your constraints.

🚀 Acting Phase: It launches new machines through cloud APIs, installs Ray software, and integrates them into your cluster.

🧹 Cleanup Phase: When machines sit idle too long, it safely removes them to save costs.

### Multi-Level Decision Making

Ray's autoscaler operates at multiple levels to make optimal decisions:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

## Core Autoscaling Components

Let's dive into the key players that make Ray's autoscaling system work. Think of these as different departments in a company, each with specific responsibilities.

### 1. StandardAutoscaler - The Main Controller

Location: python/ray/autoscaler/\_private/autoscaler.py  
This is the "CEO" of the autoscaling system - it coordinates everything and makes the final decisions.

python/ray/autoscaler/\_private/autoscaler.py

class StandardAutoscaler:  
def \_\_init\_\_(self, config\_reader, load\_metrics, gcs\_client, ...):  
# The brain of the operation  
self.provider = self.\_get\_node\_provider(provider\_config, cluster\_name)  
self.resource\_demand\_scheduler = ResourceDemandScheduler(...)  
self.load\_metrics = load\_metrics  
# Key configuration settings  
self.max\_workers = config.get("max\_workers", 0)  
self.upscaling\_speed = config.get("upscaling\_speed", 1.0)  
self.idle\_timeout\_minutes = config.get("idle\_timeout\_minutes", 5)

class StandardAutoscaler:  
def \_\_init\_\_(self, config\_reader, load\_metrics, gcs\_client, ...):  
# The brain of the operation  
self.provider = self.\_get\_node\_provider(provider\_config, cluster\_name)  
self.resource\_demand\_scheduler = ResourceDemandScheduler(...)  
self.load\_metrics = load\_metrics  
# Key configuration settings  
self.max\_workers = config.get("max\_workers", 0)  
self.upscaling\_speed = config.get("upscaling\_speed", 1.0)  
self.idle\_timeout\_minutes = config.get("idle\_timeout\_minutes", 5)

What It Does (In Simple Terms):  
- Wakes up every few seconds to check if the cluster needs changes  
- Decides when to add new machines (scale up)  
- Decides when to remove idle machines (scale down)  
- Ensures the cluster never exceeds your budget or size limits  
Key Responsibilities:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### 2. ResourceDemandScheduler - The Smart Planner

Location: python/ray/autoscaler/\_private/resource\_demand\_scheduler.py  
This component is like a smart logistics coordinator that figures out the most efficient way to arrange your computing resources.

python/ray/autoscaler/\_private/resource\_demand\_scheduler.py

class ResourceDemandScheduler:  
def get\_nodes\_to\_launch(self,  
resource\_demands, # What you need  
unused\_resources\_by\_ip, # What's available  
pending\_placement\_groups, # Complex arrangements  
max\_resources\_by\_ip): # Machine capacities  
# Step 1: Understand current cluster state  
node\_resources, node\_type\_counts = self.calculate\_node\_resources(...)  
# Step 2: Respect minimum worker requirements  
adjusted\_min\_workers = self.\_add\_min\_workers\_nodes(...)  
# Step 3: Handle placement groups (complex workloads)  
spread\_pg\_nodes = self.reserve\_and\_allocate\_spread(...)  
# Step 4: Use "bin packing" to find optimal machine mix  
nodes\_to\_add, unfulfilled = get\_nodes\_for(...)  
return total\_nodes\_to\_add, final\_unfulfilled

class ResourceDemandScheduler:  
def get\_nodes\_to\_launch(self,  
resource\_demands, # What you need  
unused\_resources\_by\_ip, # What's available  
pending\_placement\_groups, # Complex arrangements  
max\_resources\_by\_ip): # Machine capacities  
# Step 1: Understand current cluster state  
node\_resources, node\_type\_counts = self.calculate\_node\_resources(...)  
# Step 2: Respect minimum worker requirements  
adjusted\_min\_workers = self.\_add\_min\_workers\_nodes(...)  
# Step 3: Handle placement groups (complex workloads)  
spread\_pg\_nodes = self.reserve\_and\_allocate\_spread(...)  
# Step 4: Use "bin packing" to find optimal machine mix  
nodes\_to\_add, unfulfilled = get\_nodes\_for(...)  
return total\_nodes\_to\_add, final\_unfulfilled

The Bin Packing Magic: Think of this like playing Tetris with cloud machines. You have different shaped "resource blocks" (your tasks) and different sized "containers" (machine types). The scheduler finds the combination that wastes the least space and costs the least money.  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### 3. LoadMetrics - The Cluster Monitor

Location: python/ray/autoscaler/\_private/load\_metrics.py  
This is like having a health monitor attached to your cluster that constantly reports vital signs.

python/ray/autoscaler/\_private/load\_metrics.py

class LoadMetrics:  
def \_\_init\_\_(self):  
# Tracks what resources each machine has  
self.static\_resources\_by\_ip = {} # Total capacity  
self.dynamic\_resources\_by\_ip = {} # Currently available  
# Tracks what work is waiting  
self.pending\_resource\_requests = [] # Individual tasks  
self.pending\_placement\_groups = [] # Complex arrangements  
# Tracks cluster health  
self.last\_heartbeat\_time\_by\_ip = {} # When we last heard from nodes  
self.last\_heartbeat\_failed = {} # Which nodes are unresponsive

class LoadMetrics:  
def \_\_init\_\_(self):  
# Tracks what resources each machine has  
self.static\_resources\_by\_ip = {} # Total capacity  
self.dynamic\_resources\_by\_ip = {} # Currently available  
# Tracks what work is waiting  
self.pending\_resource\_requests = [] # Individual tasks  
self.pending\_placement\_groups = [] # Complex arrangements  
# Tracks cluster health  
self.last\_heartbeat\_time\_by\_ip = {} # When we last heard from nodes  
self.last\_heartbeat\_failed = {} # Which nodes are unresponsive

What It Monitors:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### 4. Node Providers - The Cloud Connectors

Location: python/ray/autoscaler/\_private/providers.py  
These are like specialized translators that know how to talk to different cloud providers. Each provider speaks its own "language" (API), but Ray abstracts this complexity.

python/ray/autoscaler/\_private/providers.py

# AWS Provider  
class AWSNodeProvider(NodeProvider):  
def create\_node(self, node\_config, tags, count):  
# Launches EC2 instances using AWS API  
response = self.ec2.run\_instances(  
ImageId=node\_config["ImageId"],  
InstanceType=node\_config["InstanceType"],  
MinCount=count, MaxCount=count,  
SubnetId=node\_config["SubnetId"]  
)  
return [instance.id for instance in response["Instances"]]  
# GCP Provider  
class GCPNodeProvider(NodeProvider):  
def create\_node(self, node\_config, tags, count):  
# Launches Compute Engine instances using GCP API  
operation = self.compute.instances().insert(  
project=self.project\_id,  
zone=self.zone,  
body=instance\_config  
).execute()  
return operation["targetId"]

# AWS Provider  
class AWSNodeProvider(NodeProvider):  
def create\_node(self, node\_config, tags, count):  
# Launches EC2 instances using AWS API  
response = self.ec2.run\_instances(  
ImageId=node\_config["ImageId"],  
InstanceType=node\_config["InstanceType"],  
MinCount=count, MaxCount=count,  
SubnetId=node\_config["SubnetId"]  
)  
return [instance.id for instance in response["Instances"]]  
# GCP Provider  
class GCPNodeProvider(NodeProvider):  
def create\_node(self, node\_config, tags, count):  
# Launches Compute Engine instances using GCP API  
operation = self.compute.instances().insert(  
project=self.project\_id,  
zone=self.zone,  
body=instance\_config  
).execute()  
return operation["targetId"]

Supported Cloud Providers:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### 5. GCS Autoscaler State Manager - The Central Coordinator

Location: src/ray/gcs/gcs\_server/gcs\_autoscaler\_state\_manager.cc  
This component runs inside Ray's Global Control Service (GCS) and acts as the central hub for all autoscaling information.

src/ray/gcs/gcs\_server/gcs\_autoscaler\_state\_manager.cc

class GcsAutoscalerStateManager {  
void UpdateResourceLoadAndUsage(rpc::ResourcesData data) {  
// Receives resource reports from all nodes  
NodeID node\_id = NodeID::FromBinary(data.node\_id());  
node\_resource\_info\_[node\_id] = std::move(data);  
}  
void GetPendingResourceRequests(rpc::autoscaler::ClusterResourceState \*state) {  
// Aggregates demand from all nodes  
auto aggregate\_load = GetAggregatedResourceLoad();  
for (const auto &[shape, demand] : aggregate\_load) {  
if (demand.num\_ready\_requests\_queued() > 0) {  
// Add to autoscaling demand  
auto pending\_req = state->add\_pending\_resource\_requests();  
pending\_req->set\_count(demand.num\_ready\_requests\_queued());  
}  
}  
}  
};

class GcsAutoscalerStateManager {  
void UpdateResourceLoadAndUsage(rpc::ResourcesData data) {  
// Receives resource reports from all nodes  
NodeID node\_id = NodeID::FromBinary(data.node\_id());  
node\_resource\_info\_[node\_id] = std::move(data);  
}  
void GetPendingResourceRequests(rpc::autoscaler::ClusterResourceState \*state) {  
// Aggregates demand from all nodes  
auto aggregate\_load = GetAggregatedResourceLoad();  
for (const auto &[shape, demand] : aggregate\_load) {  
if (demand.num\_ready\_requests\_queued() > 0) {  
// Add to autoscaling demand  
auto pending\_req = state->add\_pending\_resource\_requests();  
pending\_req->set\_count(demand.num\_ready\_requests\_queued());  
}  
}  
}  
};

Role in the System:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

## Resource Demand Detection

Understanding how Ray detects and measures resource demand is crucial because this drives all autoscaling decisions. Think of it like a restaurant that needs to predict how many customers will arrive and what they'll order.

### How Ray Sees Resource Demand

Ray tracks demand at multiple levels, each providing different insights:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Resource Demand Aggregation Process

Here's how Ray collects and processes demand information:

# From python/ray/autoscaler/\_private/load\_metrics.py  
class LoadMetrics:  
def summary(self) -> LoadMetricsSummary:  
# Step 1: Collect demand from each node's queued tasks  
aggregate\_load = {}  
for node\_ip, resource\_data in self.resource\_usage\_by\_ip.items():  
for resource\_shape, demand in resource\_data.items():  
total\_demand = (demand.num\_ready\_requests\_queued() +  
demand.num\_infeasible\_requests\_queued() +  
demand.backlog\_size())  
if total\_demand > 0:  
aggregate\_load[resource\_shape] = total\_demand  
# Step 2: Add placement group demands  
pg\_demands = self.\_get\_placement\_group\_demands()  
# Step 3: Add explicit resource requests  
explicit\_requests = self.resource\_requests or []  
return LoadMetricsSummary(  
resource\_demand=aggregate\_load,  
pg\_demand=pg\_demands,  
request\_demand=explicit\_requests  
)

# From python/ray/autoscaler/\_private/load\_metrics.py  
class LoadMetrics:  
def summary(self) -> LoadMetricsSummary:  
# Step 1: Collect demand from each node's queued tasks  
aggregate\_load = {}  
for node\_ip, resource\_data in self.resource\_usage\_by\_ip.items():  
for resource\_shape, demand in resource\_data.items():  
total\_demand = (demand.num\_ready\_requests\_queued() +  
demand.num\_infeasible\_requests\_queued() +  
demand.backlog\_size())  
if total\_demand > 0:  
aggregate\_load[resource\_shape] = total\_demand  
# Step 2: Add placement group demands  
pg\_demands = self.\_get\_placement\_group\_demands()  
# Step 3: Add explicit resource requests  
explicit\_requests = self.resource\_requests or []  
return LoadMetricsSummary(  
resource\_demand=aggregate\_load,  
pg\_demand=pg\_demands,  
request\_demand=explicit\_requests  
)

### Types of Resource Shapes

Ray thinks about resources in "shapes" - specific combinations of resources that tasks need:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Real-Time Demand Tracking

The GCS continuously receives updates from all cluster nodes about their resource usage and pending work:

// From src/ray/gcs/gcs\_server/gcs\_autoscaler\_state\_manager.cc  
void GcsAutoscalerStateManager::UpdateResourceLoadAndUsage(rpc::ResourcesData data) {  
NodeID node\_id = NodeID::FromBinary(data.node\_id());  
// Update this node's resource information  
auto &node\_info = node\_resource\_info\_[node\_id];  
node\_info.second = std::move(data);  
node\_info.first = absl::Now(); // Last update time  
// The data includes:  
// - Total resources on this node  
// - Currently available resources  
// - Resource demands by shape (queued tasks)  
// - Object store memory usage  
// - Placement group demands  
}

// From src/ray/gcs/gcs\_server/gcs\_autoscaler\_state\_manager.cc  
void GcsAutoscalerStateManager::UpdateResourceLoadAndUsage(rpc::ResourcesData data) {  
NodeID node\_id = NodeID::FromBinary(data.node\_id());  
// Update this node's resource information  
auto &node\_info = node\_resource\_info\_[node\_id];  
node\_info.second = std::move(data);  
node\_info.first = absl::Now(); // Last update time  
// The data includes:  
// - Total resources on this node  
// - Currently available resources  
// - Resource demands by shape (queued tasks)  
// - Object store memory usage  
// - Placement group demands  
}

### Demand Processing Pipeline

Here's the complete flow of how demand information travels through the system:  
📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

### Intelligent Demand Prediction

Ray doesn't just react to current demand - it predicts future needs:

# Proactive scaling based on trends  
def \_should\_scale\_up\_preemptively(self, load\_metrics):  
# Look at demand growth rate  
current\_demand = len(load\_metrics.pending\_tasks)  
demand\_growth\_rate = (current\_demand - self.last\_demand) / self.update\_interval  
# If demand is growing quickly, scale up before we run out  
if demand\_growth\_rate > self.preemptive\_threshold:  
return True  
# Look at placement group patterns  
pending\_pgs = load\_metrics.pending\_placement\_groups  
if len(pending\_pgs) > 0:  
# Placement groups often come in batches  
return True  
return False

# Proactive scaling based on trends  
def \_should\_scale\_up\_preemptively(self, load\_metrics):  
# Look at demand growth rate  
current\_demand = len(load\_metrics.pending\_tasks)  
demand\_growth\_rate = (current\_demand - self.last\_demand) / self.update\_interval  
# If demand is growing quickly, scale up before we run out  
if demand\_growth\_rate > self.preemptive\_threshold:  
return True  
# Look at placement group patterns  
pending\_pgs = load\_metrics.pending\_placement\_groups  
if len(pending\_pgs) > 0:  
# Placement groups often come in batches  
return True  
return False

# Chapter 11: High Availability and Fault Tolerance

## Table of Contents

Introduction

Architecture Overview

Core HA Components

GCS Fault Tolerance

Node Failure Handling

Actor Fault Tolerance

Object Fault Tolerance

Network Partition Recovery

Health Monitoring

Recovery Mechanisms

Performance Impact

Implementation Details

Configuration Guidelines

## Introduction

Ray's High Availability (HA) system provides comprehensive fault tolerance across all layers of the distributed system. It ensures that Ray clusters can survive and recover from various types of failures including node crashes, network partitions, process failures, and storage outages. The HA system is designed to minimize downtime and maintain service continuity while preserving data consistency and system reliability.

### Key Principles

Layered Fault Tolerance: Different components have specialized recovery mechanisms

Automatic Recovery: Most failures are handled automatically without manual intervention

Graceful Degradation: System continues operating with reduced capacity during failures

State Preservation: Critical state is persisted to enable recovery after failures

Minimal Performance Impact: HA mechanisms are optimized for production workloads

### Failure Types Handled

Head Node Failures: GCS server crashes, head node hardware failures

Worker Node Failures: Raylet crashes, worker node hardware failures

Process Failures: Actor crashes, task failures, worker process exits

Network Partitions: Network splits, connectivity issues

Storage Failures: Redis outages, disk failures, I/O errors

Resource Exhaustion: Memory pressure, CPU saturation, disk space

## Architecture Overview

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### HA Design Philosophy

Failure Isolation: Failures in one component don't cascade to others  
Fast Recovery: Minimize time between failure detection and recovery completion  
Consistency Preservation: Maintain data consistency during recovery operations  
Observability: Comprehensive monitoring and alerting for failure scenarios

## Core HA Components

The Ray HA system consists of several interconnected components working together to provide comprehensive fault tolerance.

### Component Interaction Model

📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

### HA Component Responsibilities

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

python/ray/tests/test\_gcs\_fault\_tolerance.py:45-100

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

// From test configuration  
struct GCSRecoveryConfig {  
int64\_t gcs\_rpc\_server\_reconnect\_timeout\_s = 60; // Reconnection timeout  
int64\_t gcs\_server\_request\_timeout\_seconds = 10; // Request timeout  
int64\_t redis\_db\_connect\_retries = 50; // Redis retry attempts  
bool enable\_external\_redis = true; // Use persistent Redis  
};

// From test configuration  
struct GCSRecoveryConfig {  
int64\_t gcs\_rpc\_server\_reconnect\_timeout\_s = 60; // Reconnection timeout  
int64\_t gcs\_server\_request\_timeout\_seconds = 10; // Request timeout  
int64\_t redis\_db\_connect\_retries = 50; // Redis retry attempts  
bool enable\_external\_redis = true; // Use persistent Redis  
};

### Critical State Preserved

Node Registry: All active and failed nodes

Actor Information: Actor metadata and placement

Job State: Running and completed jobs

Resource Allocation: Cluster resource assignments

Placement Groups: Group configurations and status

## Node Failure Handling

Ray implements sophisticated node failure detection and recovery mechanisms to maintain cluster health.

### Node State Transitions

🔧 TECHNICAL DIAGRAM: System Architecture

[DIAGRAM: 🔧 TECHNICAL DIAGRAM: System Architecture]

### Health Check Protocol

From src/ray/gcs/gcs\_server/gcs\_health\_check\_manager.h:40-60:

src/ray/gcs/gcs\_server/gcs\_health\_check\_manager.h:40-60

class GcsHealthCheckManager {  
// Health check configuration  
int64\_t initial\_delay\_ms\_; // Delay before first check  
int64\_t timeout\_ms\_; // Timeout per health check  
int64\_t period\_ms\_; // Interval between checks  
int64\_t failure\_threshold\_; // Failures before marking dead  
// Health check process  
void StartHealthCheck() {  
// Send gRPC health check to node  
stub\_->Check(request\_, &response\_, [this](Status status) {  
if (status.ok()) {  
health\_check\_remaining\_ = failure\_threshold\_; // Reset counter  
ScheduleNextCheck();  
} else {  
health\_check\_remaining\_--;  
if (health\_check\_remaining\_ <= 0) {  
manager\_->FailNode(node\_id\_); // Mark node as failed  
} else {  
ScheduleNextCheck(); // Retry after delay  
}  
}  
});  
}  
};

class GcsHealthCheckManager {  
// Health check configuration  
int64\_t initial\_delay\_ms\_; // Delay before first check  
int64\_t timeout\_ms\_; // Timeout per health check  
int64\_t period\_ms\_; // Interval between checks  
int64\_t failure\_threshold\_; // Failures before marking dead  
// Health check process  
void StartHealthCheck() {  
// Send gRPC health check to node  
stub\_->Check(request\_, &response\_, [this](Status status) {  
if (status.ok()) {  
health\_check\_remaining\_ = failure\_threshold\_; // Reset counter  
ScheduleNextCheck();  
} else {  
health\_check\_remaining\_--;  
if (health\_check\_remaining\_ <= 0) {  
manager\_->FailNode(node\_id\_); // Mark node as failed  
} else {  
ScheduleNextCheck(); // Retry after delay  
}  
}  
});  
}  
};

### Node Failure Impact and Recovery

Immediate Effects:  
- All running tasks on the node are terminated  
- Actors hosted on the node become unavailable  
- Objects stored locally are marked as lost  
- Resource allocations are freed  
Recovery Actions:  
- Failed tasks are automatically retried on healthy nodes  
- Actors with max\_restarts > 0 are restarted elsewhere  
- Lost objects are reconstructed via lineage if possible  
- Resource scheduling excludes the failed node

max\_restarts > 0

## Actor Fault Tolerance

Ray actors can automatically recover from failures through configurable restart policies and state management.

### Actor Restart Mechanisms

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Actor Restart Configuration

From doc/source/ray-core/doc\_code/actor\_restart.py:8-15:

doc/source/ray-core/doc\_code/actor\_restart.py:8-15

@ray.remote(max\_restarts=4, max\_task\_retries=-1)  
class FaultTolerantActor:  
def \_\_init\_\_(self):  
self.counter = 0  
# Actor state is reconstructed by re-running constructor  
def increment\_and\_possibly\_fail(self):  
if self.counter == 10:  
os.\_exit(0) # Simulate actor failure  
self.counter += 1  
return self.counter

@ray.remote(max\_restarts=4, max\_task\_retries=-1)  
class FaultTolerantActor:  
def \_\_init\_\_(self):  
self.counter = 0  
# Actor state is reconstructed by re-running constructor  
def increment\_and\_possibly\_fail(self):  
if self.counter == 10:  
os.\_exit(0) # Simulate actor failure  
self.counter += 1  
return self.counter

Restart Policy Parameters:  
| Parameter | Default | Description | Effect |  
|-----------|---------|-------------|---------|  
| max\_restarts | 0 | Maximum actor restarts | Controls restart attempts |  
| max\_task\_retries | 0 | Task retry attempts | Enables at-least-once semantics |  
| max\_pending\_calls | -1 | Queue size limit | Prevents memory overflow |

max\_restarts

max\_task\_retries

max\_pending\_calls

### Actor Lifecycle During Failures

📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

## Object Fault Tolerance

Ray provides automatic object recovery through lineage reconstruction and data replication.

### Object Recovery Architecture

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Object Recovery Algorithm

From src/ray/core\_worker/object\_recovery\_manager.h:70-90:

src/ray/core\_worker/object\_recovery\_manager.h:70-90

// Object recovery algorithm steps:  
bool RecoverObject(const ObjectID &object\_id) {  
// 1. Check object ownership and missing status  
if (!IsObjectMissing(object\_id) || !IsObjectOwned(object\_id)) {  
return false; // Cannot recover  
}  
// 2. Look for existing copies on other nodes  
auto locations = GetObjectLocations(object\_id);  
if (!locations.empty()) {  
return PinObjectFromLocation(object\_id, locations);  
}  
// 3. Attempt lineage reconstruction  
auto task\_spec = GetCreationTaskSpec(object\_id);  
if (task\_spec.has\_value()) {  
return ResubmitTask(task\_spec.value());  
}  
return false; // Object not recoverable  
}

// Object recovery algorithm steps:  
bool RecoverObject(const ObjectID &object\_id) {  
// 1. Check object ownership and missing status  
if (!IsObjectMissing(object\_id) || !IsObjectOwned(object\_id)) {  
return false; // Cannot recover  
}  
// 2. Look for existing copies on other nodes  
auto locations = GetObjectLocations(object\_id);  
if (!locations.empty()) {  
return PinObjectFromLocation(object\_id, locations);  
}  
// 3. Attempt lineage reconstruction  
auto task\_spec = GetCreationTaskSpec(object\_id);  
if (task\_spec.has\_value()) {  
return ResubmitTask(task\_spec.value());  
}  
return false; // Object not recoverable  
}

### Object Recovery Limitations

Recoverable Objects:  
- Objects created by deterministic tasks  
- Objects with living owners  
- Objects with available lineage information  
Non-Recoverable Objects:  
- Objects created by ray.put() (no lineage)  
- Objects with dead owners  
- Objects from non-deterministic tasks  
- Objects exceeding retry limits

ray.put()

## Health Monitoring

Ray implements comprehensive health monitoring across all cluster components.

### Multi-Layer Health Monitoring

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Health Check Implementation

GCS Health Check Manager Configuration:

// Health check parameters  
struct HealthCheckConfig {  
int64\_t initial\_delay\_ms = 5000; // Delay before first check  
int64\_t timeout\_ms = 10000; // Timeout per check  
int64\_t period\_ms = 30000; // Check interval  
int64\_t failure\_threshold = 3; // Failures before marking dead  
};  
// Health check process  
class HealthCheckContext {  
void StartHealthCheck() {  
auto deadline = std::chrono::steady\_clock::now() +  
std::chrono::milliseconds(timeout\_ms\_);  
stub\_->async()->Check(&context\_, &request\_, &response\_,  
[this](grpc::Status status) {  
if (status.ok()) {  
ResetFailureCount();  
ScheduleNextCheck();  
} else {  
IncrementFailureCount();  
if (failure\_count\_ >= failure\_threshold\_) {  
ReportNodeFailure();  
} else {  
ScheduleNextCheck();  
}  
}  
});  
}  
};

// Health check parameters  
struct HealthCheckConfig {  
int64\_t initial\_delay\_ms = 5000; // Delay before first check  
int64\_t timeout\_ms = 10000; // Timeout per check  
int64\_t period\_ms = 30000; // Check interval  
int64\_t failure\_threshold = 3; // Failures before marking dead  
};  
// Health check process  
class HealthCheckContext {  
void StartHealthCheck() {  
auto deadline = std::chrono::steady\_clock::now() +  
std::chrono::milliseconds(timeout\_ms\_);  
stub\_->async()->Check(&context\_, &request\_, &response\_,  
[this](grpc::Status status) {  
if (status.ok()) {  
ResetFailureCount();  
ScheduleNextCheck();  
} else {  
IncrementFailureCount();  
if (failure\_count\_ >= failure\_threshold\_) {  
ReportNodeFailure();  
} else {  
ScheduleNextCheck();  
}  
}  
});  
}  
};

## Recovery Mechanisms

Ray implements several coordinated recovery mechanisms to handle different failure scenarios.

### Recovery Strategy Selection

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Recovery Coordination Protocol

📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

### Performance Impact Analysis

Recovery Time Objectives:  
| Component | Detection Time | Recovery Time | Availability Target |  
|-----------|---------------|---------------|-------------------|  
| Node failure | 30-90 seconds | 2-5 minutes | 99.9% |  
| Actor failure | 1-10 seconds | 5-30 seconds | 99.95% |  
| Object loss | Near-instant | 10-60 seconds | 99.99% |  
| GCS failure | 10-30 seconds | 30-120 seconds | 99.9% |  
Throughput Impact During Recovery:  
- Node Failure: 10-30% throughput reduction during task migration  
- Actor Restart: Minimal impact on other actors  
- Object Reconstruction: Temporary latency increase for dependent tasks  
- Network Partition: Proportional to partition size

## Implementation Details

### Critical Recovery Code Paths

Node Failure Handler:

// From GcsNodeManager::OnNodeFailure  
void GcsNodeManager::OnNodeFailure(const NodeID &node\_id,  
const StatusCallback &callback) {  
auto node = GetAliveNode(node\_id);  
if (!node) return; // Node already marked dead  
// Remove from alive nodes and mark as dead  
auto death\_info = InferDeathInfo(node\_id);  
auto dead\_node = RemoveNode(node\_id, death\_info);  
// Notify all listeners (resource manager, actor manager, etc.)  
for (auto &listener : node\_removed\_listeners\_) {  
listener(dead\_node);  
}  
// Persist state change  
RAY\_CHECK\_OK(gcs\_table\_storage\_->NodeTable().Put(  
node\_id, \*dead\_node, callback));  
}

// From GcsNodeManager::OnNodeFailure  
void GcsNodeManager::OnNodeFailure(const NodeID &node\_id,  
const StatusCallback &callback) {  
auto node = GetAliveNode(node\_id);  
if (!node) return; // Node already marked dead  
// Remove from alive nodes and mark as dead  
auto death\_info = InferDeathInfo(node\_id);  
auto dead\_node = RemoveNode(node\_id, death\_info);  
// Notify all listeners (resource manager, actor manager, etc.)  
for (auto &listener : node\_removed\_listeners\_) {  
listener(dead\_node);  
}  
// Persist state change  
RAY\_CHECK\_OK(gcs\_table\_storage\_->NodeTable().Put(  
node\_id, \*dead\_node, callback));  
}

Actor Restart Logic:

// Actor restart decision process  
bool ShouldRestartActor(const ActorID &actor\_id) {  
auto actor\_info = GetActorInfo(actor\_id);  
if (!actor\_info) return false;  
int current\_restarts = actor\_info->num\_restarts();  
int max\_restarts = actor\_info->max\_restarts();  
// Check restart policy  
if (max\_restarts == 0) return false; // No restarts allowed  
if (max\_restarts == -1) return true; // Infinite restarts  
return current\_restarts < max\_restarts; // Within limit  
}

// Actor restart decision process  
bool ShouldRestartActor(const ActorID &actor\_id) {  
auto actor\_info = GetActorInfo(actor\_id);  
if (!actor\_info) return false;  
int current\_restarts = actor\_info->num\_restarts();  
int max\_restarts = actor\_info->max\_restarts();  
// Check restart policy  
if (max\_restarts == 0) return false; // No restarts allowed  
if (max\_restarts == -1) return true; // Infinite restarts  
return current\_restarts < max\_restarts; // Within limit  
}

### Error Handling Patterns

Graceful Degradation Example:

Status HandleObjectRecovery(const ObjectID &object\_id) {  
// Try multiple recovery strategies in order  
if (auto status = TryPinFromOtherNodes(object\_id); status.ok()) {  
return status;  
}  
if (auto status = TryLineageReconstruction(object\_id); status.ok()) {  
return status;  
}  
if (auto status = TrySpillRecovery(object\_id); status.ok()) {  
return status;  
}  
// All recovery methods failed  
return Status::ObjectLost("Object cannot be recovered");  
}

Status HandleObjectRecovery(const ObjectID &object\_id) {  
// Try multiple recovery strategies in order  
if (auto status = TryPinFromOtherNodes(object\_id); status.ok()) {  
return status;  
}  
if (auto status = TryLineageReconstruction(object\_id); status.ok()) {  
return status;  
}  
if (auto status = TrySpillRecovery(object\_id); status.ok()) {  
return status;  
}  
// All recovery methods failed  
return Status::ObjectLost("Object cannot be recovered");  
}

## Configuration Guidelines

### Ray Cluster Configuration

// From ray/core/src/ray/ray\_config.h  
struct RayConfig {  
int64\_t gcs\_rpc\_server\_reconnect\_timeout\_s = 60; // Reconnection timeout  
int64\_t gcs\_server\_request\_timeout\_seconds = 10; // Request timeout  
int64\_t redis\_db\_connect\_retries = 50; // Redis retry attempts  
bool enable\_external\_redis = true; // Use persistent Redis  
};

// From ray/core/src/ray/ray\_config.h  
struct RayConfig {  
int64\_t gcs\_rpc\_server\_reconnect\_timeout\_s = 60; // Reconnection timeout  
int64\_t gcs\_server\_request\_timeout\_seconds = 10; // Request timeout  
int64\_t redis\_db\_connect\_retries = 50; // Redis retry attempts  
bool enable\_external\_redis = true; // Use persistent Redis  
};

### Health Monitoring Configuration

// From ray/core/src/ray/ray\_config.h  
struct HealthCheckConfig {  
int64\_t initial\_delay\_ms = 5000; // Delay before first check  
int64\_t timeout\_ms = 10000; // Timeout per check  
int64\_t period\_ms = 30000; // Check interval  
int64\_t failure\_threshold = 3; // Failures before marking dead  
};

// From ray/core/src/ray/ray\_config.h  
struct HealthCheckConfig {  
int64\_t initial\_delay\_ms = 5000; // Delay before first check  
int64\_t timeout\_ms = 10000; // Timeout per check  
int64\_t period\_ms = 30000; // Check interval  
int64\_t failure\_threshold = 3; // Failures before marking dead  
};

### Recovery Configuration

// From ray/core/src/ray/ray\_config.h  
struct GCSRecoveryConfig {  
int64\_t gcs\_rpc\_server\_reconnect\_timeout\_s = 60; // Reconnection timeout  
int64\_t gcs\_server\_request\_timeout\_seconds = 10; // Request timeout  
int64\_t redis\_db\_connect\_retries = 50; // Redis retry attempts  
bool enable\_external\_redis = true; // Use persistent Redis  
};

// From ray/core/src/ray/ray\_config.h  
struct GCSRecoveryConfig {  
int64\_t gcs\_rpc\_server\_reconnect\_timeout\_s = 60; // Reconnection timeout  
int64\_t gcs\_server\_request\_timeout\_seconds = 10; // Request timeout  
int64\_t redis\_db\_connect\_retries = 50; // Redis retry attempts  
bool enable\_external\_redis = true; // Use persistent Redis  
};

## Network Partition Recovery

Ray handles network partitions through timeout-based detection and coordinated recovery.

### Partition Detection and Isolation

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Split-Brain Prevention

Quorum-Based Decision Making:

// Partition handling logic  
class PartitionDetector {  
bool ShouldShutdownOnPartition() {  
size\_t visible\_nodes = GetVisibleNodeCount();  
size\_t total\_nodes = GetTotalNodeCount();  
// Require majority quorum to continue operation  
return visible\_nodes <= total\_nodes / 2;  
}  
void HandleNetworkPartition() {  
if (ShouldShutdownOnPartition()) {  
RAY\_LOG(WARNING) << "Node in minority partition, shutting down";  
InitiateGracefulShutdown();  
} else {  
RAY\_LOG(INFO) << "Node in majority partition, continuing operation";  
MarkMinorityNodesAsFailed();  
}  
}  
};

// Partition handling logic  
class PartitionDetector {  
bool ShouldShutdownOnPartition() {  
size\_t visible\_nodes = GetVisibleNodeCount();  
size\_t total\_nodes = GetTotalNodeCount();  
// Require majority quorum to continue operation  
return visible\_nodes <= total\_nodes / 2;  
}  
void HandleNetworkPartition() {  
if (ShouldShutdownOnPartition()) {  
RAY\_LOG(WARNING) << "Node in minority partition, shutting down";  
InitiateGracefulShutdown();  
} else {  
RAY\_LOG(INFO) << "Node in majority partition, continuing operation";  
MarkMinorityNodesAsFailed();  
}  
}  
};

## Production Deployment Best Practices

### Redis High Availability Setup

Redis Cluster Configuration:

# Redis HA configuration for GCS persistence  
apiVersion: v1  
kind: ConfigMap  
metadata:  
name: redis-config  
data:  
redis.conf: |  
save 900 1 # Save if at least 1 key changed in 900 seconds  
save 300 10 # Save if at least 10 keys changed in 300 seconds  
save 60 10000 # Save if at least 10000 keys changed in 60 seconds  
# Replication settings  
replica-read-only yes  
replica-serve-stale-data yes  
# Persistence settings  
appendonly yes  
appendfsync everysec  
# Memory management  
maxmemory-policy allkeys-lru  
# Network settings  
timeout 300  
tcp-keepalive 300

# Redis HA configuration for GCS persistence  
apiVersion: v1  
kind: ConfigMap  
metadata:  
name: redis-config  
data:  
redis.conf: |  
save 900 1 # Save if at least 1 key changed in 900 seconds  
save 300 10 # Save if at least 10 keys changed in 300 seconds  
save 60 10000 # Save if at least 10000 keys changed in 60 seconds  
# Replication settings  
replica-read-only yes  
replica-serve-stale-data yes  
# Persistence settings  
appendonly yes  
appendfsync everysec  
# Memory management  
maxmemory-policy allkeys-lru  
# Network settings  
timeout 300  
tcp-keepalive 300

### KubeRay HA Configuration

RayService with GCS Fault Tolerance:

apiVersion: ray.io/v1alpha1  
kind: RayService  
metadata:  
name: rayservice-ha  
spec:  
serviceUnhealthySecondThreshold: 900  
deploymentUnhealthySecondThreshold: 300  
rayClusterConfig:  
headGroupSpec:  
template:  
spec:  
containers:  
- name: ray-head  
image: rayproject/ray:2.8.0  
env:  
- name: RAY\_external\_storage\_namespace  
value: "ray-cluster"  
- name: RAY\_redis\_address  
value: "redis-master:6379"  
- name: RAY\_gcs\_rpc\_server\_reconnect\_timeout\_s  
value: "60"  
- name: RAY\_gcs\_server\_request\_timeout\_seconds  
value: "10"  
- name: RAY\_redis\_db\_connect\_retries  
value: "50"  
resources:  
limits:  
cpu: "2"  
memory: "4Gi"  
requests:  
cpu: "1"  
memory: "2Gi"  
workerGroupSpecs:  
- replicas: 3  
minReplicas: 1  
maxReplicas: 10  
groupName: worker-group  
template:  
spec:  
containers:  
- name: ray-worker  
image: rayproject/ray:2.8.0  
resources:  
limits:  
cpu: "4"  
memory: "8Gi"  
requests:  
cpu: "2"  
memory: "4Gi"

apiVersion: ray.io/v1alpha1  
kind: RayService  
metadata:  
name: rayservice-ha  
spec:  
serviceUnhealthySecondThreshold: 900  
deploymentUnhealthySecondThreshold: 300  
rayClusterConfig:  
headGroupSpec:  
template:  
spec:  
containers:  
- name: ray-head  
image: rayproject/ray:2.8.0  
env:  
- name: RAY\_external\_storage\_namespace  
value: "ray-cluster"  
- name: RAY\_redis\_address  
value: "redis-master:6379"  
- name: RAY\_gcs\_rpc\_server\_reconnect\_timeout\_s  
value: "60"  
- name: RAY\_gcs\_server\_request\_timeout\_seconds  
value: "10"  
- name: RAY\_redis\_db\_connect\_retries  
value: "50"  
resources:  
limits:  
cpu: "2"  
memory: "4Gi"  
requests:  
cpu: "1"  
memory: "2Gi"  
workerGroupSpecs:  
- replicas: 3  
minReplicas: 1  
maxReplicas: 10  
groupName: worker-group  
template:  
spec:  
containers:  
- name: ray-worker  
image: rayproject/ray:2.8.0  
resources:  
limits:  
cpu: "4"  
memory: "8Gi"  
requests:  
cpu: "2"  
memory: "4Gi"

### Health Check Configuration

Comprehensive Health Monitoring:

# Application-level health monitoring  
import ray  
import time  
import logging  
@ray.remote  
class HealthMonitor:  
def \_\_init\_\_(self):  
self.start\_time = time.time()  
self.check\_interval = 30 # seconds  
def check\_cluster\_health(self):  
"""Comprehensive cluster health check"""  
health\_status = {  
'timestamp': time.time(),  
'uptime': time.time() - self.start\_time,  
'nodes': {},  
'actors': {},  
'objects': {}  
}  
# Check node health  
nodes = ray.nodes()  
for node in nodes:  
health\_status['nodes'][node['NodeID']] = {  
'alive': node['Alive'],  
'resources': node['Resources'],  
'cpu\_usage': node.get('cpu', 0),  
'memory\_usage': node.get('memory', 0)  
}  
# Check actor health  
try:  
actors = ray.util.state.list\_actors()  
for actor in actors:  
health\_status['actors'][actor['actor\_id']] = {  
'state': actor['state'],  
'pid': actor.get('pid'),  
'node\_id': actor.get('node\_id')  
}  
except Exception as e:  
logging.warning(f"Failed to get actor status: {e}")  
return health\_status  
def monitor\_continuously(self):  
"""Continuous health monitoring loop"""  
while True:  
try:  
health = self.check\_cluster\_health()  
# Log unhealthy components  
dead\_nodes = [nid for nid, info in health['nodes'].items()  
if not info['alive']]  
if dead\_nodes:  
logging.warning(f"Dead nodes detected: {dead\_nodes}")  
failed\_actors = [aid for aid, info in health['actors'].items()  
if info['state'] == 'FAILED']  
if failed\_actors:  
logging.warning(f"Failed actors detected: {failed\_actors}")  
except Exception as e:  
logging.error(f"Health check failed: {e}")  
time.sleep(self.check\_interval)

# Application-level health monitoring  
import ray  
import time  
import logging  
@ray.remote  
class HealthMonitor:  
def \_\_init\_\_(self):  
self.start\_time = time.time()  
self.check\_interval = 30 # seconds  
def check\_cluster\_health(self):  
"""Comprehensive cluster health check"""  
health\_status = {  
'timestamp': time.time(),  
'uptime': time.time() - self.start\_time,  
'nodes': {},  
'actors': {},  
'objects': {}  
}  
# Check node health  
nodes = ray.nodes()  
for node in nodes:  
health\_status['nodes'][node['NodeID']] = {  
'alive': node['Alive'],  
'resources': node['Resources'],  
'cpu\_usage': node.get('cpu', 0),  
'memory\_usage': node.get('memory', 0)  
}  
# Check actor health  
try:  
actors = ray.util.state.list\_actors()  
for actor in actors:  
health\_status['actors'][actor['actor\_id']] = {  
'state': actor['state'],  
'pid': actor.get('pid'),  
'node\_id': actor.get('node\_id')  
}  
except Exception as e:  
logging.warning(f"Failed to get actor status: {e}")  
return health\_status  
def monitor\_continuously(self):  
"""Continuous health monitoring loop"""  
while True:  
try:  
health = self.check\_cluster\_health()  
# Log unhealthy components  
dead\_nodes = [nid for nid, info in health['nodes'].items()  
if not info['alive']]  
if dead\_nodes:  
logging.warning(f"Dead nodes detected: {dead\_nodes}")  
failed\_actors = [aid for aid, info in health['actors'].items()  
if info['state'] == 'FAILED']  
if failed\_actors:  
logging.warning(f"Failed actors detected: {failed\_actors}")  
except Exception as e:  
logging.error(f"Health check failed: {e}")  
time.sleep(self.check\_interval)

## Testing and Validation

### Chaos Engineering for HA Testing

Node Failure Simulation:

import ray  
import psutil  
import random  
import time  
@ray.remote  
class ChaosAgent:  
"""Simulates various failure scenarios for HA testing"""  
def simulate\_node\_failure(self, duration\_seconds=60):  
"""Simulate node failure by stopping raylet process"""  
try:  
# Find raylet process  
for proc in psutil.process\_iter(['pid', 'name']):  
if 'raylet' in proc.info['name']:  
proc.terminate()  
break  
time.sleep(duration\_seconds)  
# Raylet should be restarted by process manager  
return "Node failure simulation completed"  
except Exception as e:  
return f"Simulation failed: {e}"  
def simulate\_memory\_pressure(self, allocation\_mb=1000):  
"""Simulate memory pressure"""  
data = []  
try:  
# Allocate memory to create pressure  
for \_ in range(allocation\_mb):  
data.append(b'x' \* 1024 \* 1024) # 1MB chunks  
time.sleep(30) # Hold memory for 30 seconds  
return "Memory pressure simulation completed"  
except MemoryError:  
return "Memory exhausted as expected"  
finally:  
del data # Release memory  
def simulate\_network\_partition(self, target\_nodes, duration\_seconds=60):  
"""Simulate network partition using iptables rules"""  
import subprocess  
try:  
# Block traffic to/from target nodes  
for node in target\_nodes:  
subprocess.run(['iptables', '-A', 'INPUT', '-s', node, '-j', 'DROP'])  
subprocess.run(['iptables', '-A', 'OUTPUT', '-d', node, '-j', 'DROP'])  
time.sleep(duration\_seconds)  
# Restore connectivity  
for node in target\_nodes:  
subprocess.run(['iptables', '-D', 'INPUT', '-s', node, '-j', 'DROP'])  
subprocess.run(['iptables', '-D', 'OUTPUT', '-d', node, '-j', 'DROP'])  
return "Network partition simulation completed"  
except Exception as e:  
return f"Network simulation failed: {e}"

import ray  
import psutil  
import random  
import time  
@ray.remote  
class ChaosAgent:  
"""Simulates various failure scenarios for HA testing"""  
def simulate\_node\_failure(self, duration\_seconds=60):  
"""Simulate node failure by stopping raylet process"""  
try:  
# Find raylet process  
for proc in psutil.process\_iter(['pid', 'name']):  
if 'raylet' in proc.info['name']:  
proc.terminate()  
break  
time.sleep(duration\_seconds)  
# Raylet should be restarted by process manager  
return "Node failure simulation completed"  
except Exception as e:  
return f"Simulation failed: {e}"  
def simulate\_memory\_pressure(self, allocation\_mb=1000):  
"""Simulate memory pressure"""  
data = []  
try:  
# Allocate memory to create pressure  
for \_ in range(allocation\_mb):  
data.append(b'x' \* 1024 \* 1024) # 1MB chunks  
time.sleep(30) # Hold memory for 30 seconds  
return "Memory pressure simulation completed"  
except MemoryError:  
return "Memory exhausted as expected"  
finally:  
del data # Release memory  
def simulate\_network\_partition(self, target\_nodes, duration\_seconds=60):  
"""Simulate network partition using iptables rules"""  
import subprocess  
try:  
# Block traffic to/from target nodes  
for node in target\_nodes:  
subprocess.run(['iptables', '-A', 'INPUT', '-s', node, '-j', 'DROP'])  
subprocess.run(['iptables', '-A', 'OUTPUT', '-d', node, '-j', 'DROP'])  
time.sleep(duration\_seconds)  
# Restore connectivity  
for node in target\_nodes:  
subprocess.run(['iptables', '-D', 'INPUT', '-s', node, '-j', 'DROP'])  
subprocess.run(['iptables', '-D', 'OUTPUT', '-d', node, '-j', 'DROP'])  
return "Network partition simulation completed"  
except Exception as e:  
return f"Network simulation failed: {e}"

HA Test Suite:

import pytest  
import ray  
import time  
class TestRayHighAvailability:  
def setup\_method(self):  
"""Setup test cluster"""  
ray.init(address='ray://localhost:10001')  
def teardown\_method(self):  
"""Cleanup after test"""  
ray.shutdown()  
def test\_actor\_restart\_on\_failure(self):  
"""Test actor automatic restart after failure"""  
@ray.remote(max\_restarts=3)  
class TestActor:  
def \_\_init\_\_(self):  
self.counter = 0  
def increment(self):  
self.counter += 1  
if self.counter == 5:  
import os  
os.\_exit(1) # Simulate crash  
return self.counter  
actor = TestActor.remote()  
# Should succeed for first 4 calls  
for i in range(4):  
result = ray.get(actor.increment.remote())  
assert result == i + 1  
# 5th call causes crash, but actor should restart  
with pytest.raises(ray.exceptions.RayActorError):  
ray.get(actor.increment.remote())  
time.sleep(5) # Wait for restart  
result = ray.get(actor.increment.remote())  
assert result == 1 # Counter reset after restart  
def test\_object\_reconstruction(self):  
"""Test object reconstruction after data loss"""  
@ray.remote  
def create\_data(size\_mb):  
return b'x' \* (size\_mb \* 1024 \* 1024)  
# Create object  
obj\_ref = create\_data.remote(10)  
original\_data = ray.get(obj\_ref)  
# Simulate object loss (this is hard to do directly)  
# In practice, you'd kill the node storing the object  
# Object should be reconstructible  
reconstructed\_data = ray.get(obj\_ref)  
assert original\_data == reconstructed\_data  
def test\_gcs\_recovery(self):  
"""Test GCS server recovery (requires external Redis)"""  
@ray.remote  
class PersistentActor:  
def get\_pid(self):  
import os  
return os.getpid()  
actors = [PersistentActor.remote() for \_ in range(5)]  
pids\_before = ray.get([actor.get\_pid.remote() for actor in actors])  
# Kill GCS server (in real test, you'd restart GCS process)  
# This requires external coordination  
# Verify actors survive GCS restart  
time.sleep(10) # Wait for GCS recovery  
pids\_after = ray.get([actor.get\_pid.remote() for actor in actors])  
# Actor PIDs should be unchanged (actors survived)  
assert pids\_before == pids\_after

import pytest  
import ray  
import time  
class TestRayHighAvailability:  
def setup\_method(self):  
"""Setup test cluster"""  
ray.init(address='ray://localhost:10001')  
def teardown\_method(self):  
"""Cleanup after test"""  
ray.shutdown()  
def test\_actor\_restart\_on\_failure(self):  
"""Test actor automatic restart after failure"""  
@ray.remote(max\_restarts=3)  
class TestActor:  
def \_\_init\_\_(self):  
self.counter = 0  
def increment(self):  
self.counter += 1  
if self.counter == 5:  
import os  
os.\_exit(1) # Simulate crash  
return self.counter  
actor = TestActor.remote()  
# Should succeed for first 4 calls  
for i in range(4):  
result = ray.get(actor.increment.remote())  
assert result == i + 1  
# 5th call causes crash, but actor should restart  
with pytest.raises(ray.exceptions.RayActorError):  
ray.get(actor.increment.remote())  
time.sleep(5) # Wait for restart  
result = ray.get(actor.increment.remote())  
assert result == 1 # Counter reset after restart  
def test\_object\_reconstruction(self):  
"""Test object reconstruction after data loss"""  
@ray.remote  
def create\_data(size\_mb):  
return b'x' \* (size\_mb \* 1024 \* 1024)  
# Create object  
obj\_ref = create\_data.remote(10)  
original\_data = ray.get(obj\_ref)  
# Simulate object loss (this is hard to do directly)  
# In practice, you'd kill the node storing the object  
# Object should be reconstructible  
reconstructed\_data = ray.get(obj\_ref)  
assert original\_data == reconstructed\_data  
def test\_gcs\_recovery(self):  
"""Test GCS server recovery (requires external Redis)"""  
@ray.remote  
class PersistentActor:  
def get\_pid(self):  
import os  
return os.getpid()  
actors = [PersistentActor.remote() for \_ in range(5)]  
pids\_before = ray.get([actor.get\_pid.remote() for actor in actors])  
# Kill GCS server (in real test, you'd restart GCS process)  
# This requires external coordination  
# Verify actors survive GCS restart  
time.sleep(10) # Wait for GCS recovery  
pids\_after = ray.get([actor.get\_pid.remote() for actor in actors])  
# Actor PIDs should be unchanged (actors survived)  
assert pids\_before == pids\_after

### Performance Benchmarking

HA Overhead Measurement:

import ray  
import time  
import statistics  
def benchmark\_ha\_overhead():  
"""Measure performance overhead of HA features"""  
# Baseline: No HA features  
ray.init(address='ray://localhost:10001')  
@ray.remote  
class BaselineActor:  
def compute(self, data):  
return sum(data)  
# Benchmark baseline  
actor = BaselineActor.remote()  
data = list(range(10000))  
start\_time = time.time()  
futures = [actor.compute.remote(data) for \_ in range(100)]  
results = ray.get(futures)  
baseline\_time = time.time() - start\_time  
ray.shutdown()  
ray.init(address='ray://localhost:10001')  
@ray.remote(max\_restarts=3, max\_task\_retries=2)  
class HAEnabledActor:  
def compute(self, data):  
return sum(data)  
# Benchmark with HA  
actor = HAEnabledActor.remote()  
start\_time = time.time()  
futures = [actor.compute.remote(data) for \_ in range(100)]  
results = ray.get(futures)  
ha\_time = time.time() - start\_time  
overhead\_percent = ((ha\_time - baseline\_time) / baseline\_time) \* 100  
print(f"Baseline time: {baseline\_time:.2f}s")  
print(f"HA enabled time: {ha\_time:.2f}s")  
print(f"HA overhead: {overhead\_percent:.2f}%")  
return overhead\_percent  
if \_\_name\_\_ == "\_\_main\_\_":  
overhead = benchmark\_ha\_overhead()  
assert overhead < 10, f"HA overhead too high: {overhead}%"

import ray  
import time  
import statistics  
def benchmark\_ha\_overhead():  
"""Measure performance overhead of HA features"""  
# Baseline: No HA features  
ray.init(address='ray://localhost:10001')  
@ray.remote  
class BaselineActor:  
def compute(self, data):  
return sum(data)  
# Benchmark baseline  
actor = BaselineActor.remote()  
data = list(range(10000))  
start\_time = time.time()  
futures = [actor.compute.remote(data) for \_ in range(100)]  
results = ray.get(futures)  
baseline\_time = time.time() - start\_time  
ray.shutdown()  
ray.init(address='ray://localhost:10001')  
@ray.remote(max\_restarts=3, max\_task\_retries=2)  
class HAEnabledActor:  
def compute(self, data):  
return sum(data)  
# Benchmark with HA  
actor = HAEnabledActor.remote()  
start\_time = time.time()  
futures = [actor.compute.remote(data) for \_ in range(100)]  
results = ray.get(futures)  
ha\_time = time.time() - start\_time  
overhead\_percent = ((ha\_time - baseline\_time) / baseline\_time) \* 100  
print(f"Baseline time: {baseline\_time:.2f}s")  
print(f"HA enabled time: {ha\_time:.2f}s")  
print(f"HA overhead: {overhead\_percent:.2f}%")  
return overhead\_percent  
if \_\_name\_\_ == "\_\_main\_\_":  
overhead = benchmark\_ha\_overhead()  
assert overhead < 10, f"HA overhead too high: {overhead}%"

## Best Practices and Recommendations

### Production Deployment Checklist

Infrastructure Setup:  
- [ ] Deploy Redis cluster with replication and persistence  
- [ ] Configure external storage for object spilling  
- [ ] Set up monitoring and alerting systems  
- [ ] Implement automated backup procedures  
- [ ] Configure network policies and firewalls  
Ray Configuration:  
- [ ] Enable GCS fault tolerance with external Redis  
- [ ] Configure appropriate health check intervals  
- [ ] Set reasonable retry limits for tasks and actors  
- [ ] Tune memory and resource allocation  
- [ ] Enable comprehensive logging and metrics  
Application Design:  
- [ ] Design actors with restart capabilities  
- [ ] Implement idempotent task functions  
- [ ] Avoid storing critical state only in memory  
- [ ] Use placement groups for co-location requirements  
- [ ] Handle exceptions and failures gracefully

### Common Pitfalls and Solutions

# Essential HA metrics  
ha\_metrics = {  
'node\_failures\_per\_hour': 'Rate of node failures',  
'actor\_restart\_rate': 'Actor restart frequency',  
'object\_reconstruction\_time': 'Time to reconstruct lost objects',  
'gcs\_recovery\_time': 'GCS server recovery duration',  
'network\_partition\_events': 'Network split occurrences',  
'health\_check\_failures': 'Health check failure rate',  
'storage\_backend\_availability': 'Redis/storage uptime',  
'cluster\_resource\_utilization': 'Resource usage efficiency'  
}  
# Alert thresholds  
alert\_thresholds = {  
'node\_failure\_rate': 5, # More than 5 failures per hour  
'actor\_restart\_rate': 10, # More than 10 restarts per minute  
'gcs\_recovery\_time': 300, # More than 5 minutes  
'health\_check\_failure\_rate': 20, # More than 20% failure rate  
'storage\_availability': 99.9 # Less than 99.9% uptime  
}

# Essential HA metrics  
ha\_metrics = {  
'node\_failures\_per\_hour': 'Rate of node failures',  
'actor\_restart\_rate': 'Actor restart frequency',  
'object\_reconstruction\_time': 'Time to reconstruct lost objects',  
'gcs\_recovery\_time': 'GCS server recovery duration',  
'network\_partition\_events': 'Network split occurrences',  
'health\_check\_failures': 'Health check failure rate',  
'storage\_backend\_availability': 'Redis/storage uptime',  
'cluster\_resource\_utilization': 'Resource usage efficiency'  
}  
# Alert thresholds  
alert\_thresholds = {  
'node\_failure\_rate': 5, # More than 5 failures per hour  
'actor\_restart\_rate': 10, # More than 10 restarts per minute  
'gcs\_recovery\_time': 300, # More than 5 minutes  
'health\_check\_failure\_rate': 20, # More than 20% failure rate  
'storage\_availability': 99.9 # Less than 99.9% uptime  
}

This comprehensive guide covers Ray's High Availability features, implementation details, and production deployment best practices. For the most current implementation details, refer to the source files in the Ray repository, particularly src/ray/gcs/gcs\_server/, src/ray/core\_worker/, and the fault tolerance documentation in doc/source/ray-core/fault\_tolerance/.

src/ray/gcs/gcs\_server/

src/ray/core\_worker/

doc/source/ray-core/fault\_tolerance/

# Chapter 12: Network Communication and Protocols

# Ray's Custom Protocol Over Unix Domain Sockets: A Deep Technical Dive

## Table of Contents

Introduction

Protocol Architecture Overview

Wire Protocol Format

Why Not gRPC Over UDS?

Message Types and Structure

Connection Establishment

Communication Patterns

Performance Characteristics

Comparison with Other Systems

Implementation Details

Advantages and Trade-offs

Conclusion

## Introduction

Ray uses a custom binary protocol over Unix Domain Sockets (UDS) for high-frequency, low-latency communication between workers and the local raylet. This is fundamentally different from the gRPC-over-TCP approach used for inter-node communication.

### Why a Custom Protocol?

Ray's design prioritizes performance for the critical path - the frequent interactions between workers and their local raylet. These include:  
- Task submission and completion notifications  
- Object dependency resolution  
- Worker lifecycle events  
- Resource allocation requests  
The custom protocol achieves microsecond-level latency compared to gRPC's millisecond overhead for these frequent, simple operations.

## Protocol Architecture Overview

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Key Components

Unix Domain Sockets: IPC transport mechanism

FlatBuffers: Zero-copy serialization format

Custom Message Protocol: Ray-specific message framing

Connection Management: Per-worker persistent connections

## Wire Protocol Format

Ray's wire protocol is elegantly simple, optimized for both performance and correctness:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Message Header Structure

From src/ray/common/client\_connection.cc:217-250:

src/ray/common/client\_connection.cc:217-250

Status ServerConnection::WriteMessage(int64\_t type, int64\_t length, const uint8\_t \*message) {  
auto write\_cookie = RayConfig::instance().ray\_cookie();  
return WriteBuffer({  
boost::asio::buffer(&write\_cookie, sizeof(write\_cookie)), // 8 bytes  
boost::asio::buffer(&type, sizeof(type)), // 8 bytes  
boost::asio::buffer(&length, sizeof(length)), // 8 bytes  
boost::asio::buffer(message, length), // variable  
});  
}

Status ServerConnection::WriteMessage(int64\_t type, int64\_t length, const uint8\_t \*message) {  
auto write\_cookie = RayConfig::instance().ray\_cookie();  
return WriteBuffer({  
boost::asio::buffer(&write\_cookie, sizeof(write\_cookie)), // 8 bytes  
boost::asio::buffer(&type, sizeof(type)), // 8 bytes  
boost::asio::buffer(&length, sizeof(length)), // 8 bytes  
boost::asio::buffer(message, length), // variable  
});  
}

Header Breakdown:  
- Ray Cookie (8 bytes): Protocol identifier and version check  
- Message Type (8 bytes): Identifies the FlatBuffer schema to use  
- Payload Length (8 bytes): Size of the FlatBuffer payload  
- Payload (variable): The actual FlatBuffer-serialized message

## Why Not gRPC Over UDS?

You correctly noted that gRPC can run over Unix Domain Sockets. Here's why Ray chose a custom approach:

### 1. Performance Requirements

Ray's Latency Requirements:  
- Task submission: < 10 microseconds  
- Object dependency checks: < 5 microseconds  
- Worker lifecycle events: < 1 microsecond  
gRPC Overhead (even over UDS):  
- HTTP/2 framing: ~20-50 microseconds  
- Protobuf serialization: ~10-30 microseconds  
- Connection state management: ~5-15 microseconds  
- Total gRPC overhead: 35-95 microseconds

### 2. Message Pattern Optimization

Ray's communication patterns are very specific:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships  
Ray's optimization:  
- 90% of messages are tiny (< 50 bytes)  
- These only need 24-byte headers + minimal payload  
- No need for HTTP/2 features (multiplexing, flow control, etc.)

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### 3. Custom Requirements

Ray needs specific features that gRPC doesn't optimize for:  
Synchronous Object Dependencies:  
- Worker blocks until objects are available  
- Need immediate notification when dependencies resolve  
- gRPC's async model adds unnecessary complexity  
Zero-Copy Object Access:  
- FlatBuffers allow direct buffer access  
- No need to deserialize into objects  
- Critical for high-frequency, small messages  
Predictable Performance:  
- Custom protocol has deterministic behavior  
- No hidden complexity from HTTP/2 state machine  
- Easier to profile and optimize

## Message Types and Structure

Ray defines comprehensive message types from src/ray/raylet/format/node\_manager.fbs:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

src/ray/raylet/format/node\_manager.fbs

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Core Message Categories

1. Connection Lifecycle  
- RegisterClientRequest/Reply: Worker registration and capabilities  
- DisconnectClientRequest/Reply: Graceful worker shutdown  
- AnnounceWorkerPort/Reply: gRPC port setup for remote communication  
2. Task Management  
- SubmitTask: Submit task for execution  
- ExecuteTask: Assign task to worker  
- ActorCreationTaskDone: Actor initialization complete  
3. Object Dependency Management  
- FetchOrReconstruct: Request object availability  
- WaitRequest/Reply: Wait for object dependencies  
- NotifyUnblocked: Signal dependency resolution

RegisterClientRequest/Reply

DisconnectClientRequest/Reply

AnnounceWorkerPort/Reply

SubmitTask

ExecuteTask

ActorCreationTaskDone

FetchOrReconstruct

WaitRequest/Reply

NotifyUnblocked

## Connection Establishment

The connection establishment follows a specific handshake protocol:  
📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

### Registration Details

From the FlatBuffer schema:

table RegisterClientRequest {  
worker\_type: int; // Worker, Driver, etc.  
worker\_id: string; // Unique worker identifier  
worker\_pid: long; // Process ID  
startup\_token: long; // Security token  
job\_id: string; // Job association  
runtime\_env\_hash: int; // Environment fingerprint  
language: int; // Python, Java, C++, etc.  
ip\_address: string; // Network address  
port: int; // gRPC listening port  
serialized\_job\_config: string; // Job configuration  
}

table RegisterClientRequest {  
worker\_type: int; // Worker, Driver, etc.  
worker\_id: string; // Unique worker identifier  
worker\_pid: long; // Process ID  
startup\_token: long; // Security token  
job\_id: string; // Job association  
runtime\_env\_hash: int; // Environment fingerprint  
language: int; // Python, Java, C++, etc.  
ip\_address: string; // Network address  
port: int; // gRPC listening port  
serialized\_job\_config: string; // Job configuration  
}

## Communication Patterns

Ray uses different communication patterns optimized for specific use cases:

### 1. Fire-and-Forget Pattern

For non-critical notifications that don't require responses:  
📊 SEQUENCE DIAGRAM: Process Flow and Interactions  
Implementation:

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

Status RayletClient::ActorCreationTaskDone() {  
return conn\_->WriteMessage(MessageType::ActorCreationTaskDone);  
}

Status RayletClient::ActorCreationTaskDone() {  
return conn\_->WriteMessage(MessageType::ActorCreationTaskDone);  
}

### 2. Request-Reply Pattern

For operations requiring confirmation or data return:  
📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

### 3. Asynchronous Notification Pattern

For events that may arrive at any time:  
📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

## Performance Characteristics

### Latency Analysis

Ray's custom protocol achieves significant performance advantages:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Throughput Characteristics

Message Size Efficiency:  
| Message Type | Ray Protocol | gRPC Equivalent | Savings |  
|-------------|-------------|-----------------|---------|  
| ActorCreationTaskDone | 24 bytes | ~200 bytes | 88% |  
| NotifyUnblocked | 48 bytes | ~250 bytes | 81% |  
| RegisterClient | ~300 bytes | ~500 bytes | 40% |  
Connection Overhead:  
| Aspect | Ray Protocol | gRPC |  
|--------|-------------|------|  
| Connection setup | ~100μs | ~2ms |  
| Per-message overhead | 24 bytes | 50-100 bytes |  
| Memory per connection | ~8KB | ~32KB |

ActorCreationTaskDone

NotifyUnblocked

RegisterClient

## Comparison with Other Systems

### ScyllaDB Similarity Analysis

Based on the provided ScyllaDB documentation, there are interesting parallels:  
Similarities:  
1. Custom Protocol Focus: Both Ray and ScyllaDB choose custom protocols for performance-critical paths  
2. Memory Management: Both systems carefully manage memory allocation and use semaphores for resource control  
3. Chunked Processing: Both handle large requests by breaking them into chunks  
Key Differences:  
| Aspect | Ray | ScyllaDB |  
|--------|-----|----------|  
| Transport | Unix Domain Sockets | TCP/Network |  
| Serialization | FlatBuffers | Custom binary format |  
| Use Case | Local IPC only | Network communication |  
| Memory Strategy | Zero-copy when possible | Pre-reservation with expansion |

### Ray vs. gRPC Design Philosophy

Ray's Approach:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships  
gRPC's Approach:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

## Implementation Details

### FlatBuffers Integration

Ray chose FlatBuffers over Protocol Buffers for several reasons:  
🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

### Connection Management

Each worker maintains a persistent connection to the raylet:

class RayletConnection {  
private:  
std::shared\_ptr<ServerConnection> conn\_; // UDS connection  
std::mutex mutex\_; // Thread safety  
std::mutex write\_mutex\_; // Write synchronization  
public:  
Status WriteMessage(MessageType type, flatbuffers::FlatBufferBuilder \*fbb);  
Status AtomicRequestReply(MessageType request\_type, MessageType reply\_type,  
std::vector<uint8\_t> \*reply, flatbuffers::FlatBufferBuilder \*fbb);  
};

class RayletConnection {  
private:  
std::shared\_ptr<ServerConnection> conn\_; // UDS connection  
std::mutex mutex\_; // Thread safety  
std::mutex write\_mutex\_; // Write synchronization  
public:  
Status WriteMessage(MessageType type, flatbuffers::FlatBufferBuilder \*fbb);  
Status AtomicRequestReply(MessageType request\_type, MessageType reply\_type,  
std::vector<uint8\_t> \*reply, flatbuffers::FlatBufferBuilder \*fbb);  
};

### Error Handling and Recovery

Ray's protocol includes robust error handling:  
1. Connection-Level Errors:

void RayletConnection::ShutdownIfLocalRayletDisconnected(const Status &status) {  
if (!status.ok() && IsRayletFailed(RayConfig::instance().RAYLET\_PID())) {  
RAY\_LOG(WARNING) << "Local raylet died. Terminating process.";  
QuickExit(); // Fast process termination  
}  
}

void RayletConnection::ShutdownIfLocalRayletDisconnected(const Status &status) {  
if (!status.ok() && IsRayletFailed(RayConfig::instance().RAYLET\_PID())) {  
RAY\_LOG(WARNING) << "Local raylet died. Terminating process.";  
QuickExit(); // Fast process termination  
}  
}

2. Protocol-Level Validation:

// Cookie validation for message integrity  
if (read\_cookie != RayConfig::instance().ray\_cookie()) {  
return Status::IOError("Ray cookie mismatch - protocol corruption detected");  
}  
// Message type validation  
if (expected\_type != read\_type) {  
return Status::IOError("Message type mismatch - connection corrupted");  
}

// Cookie validation for message integrity  
if (read\_cookie != RayConfig::instance().ray\_cookie()) {  
return Status::IOError("Ray cookie mismatch - protocol corruption detected");  
}  
// Message type validation  
if (expected\_type != read\_type) {  
return Status::IOError("Message type mismatch - connection corrupted");  
}

3. Graceful Shutdown:  
📊 SEQUENCE DIAGRAM: Process Flow and Interactions

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

## Advantages and Trade-offs

### Advantages

1. Performance Benefits:  
- Ultra-low latency: 1-10μs vs 50-200μs for gRPC  
- High throughput: Minimal serialization overhead  
- Zero-copy operations: Direct buffer access where possible  
- Reduced memory footprint: ~8KB vs ~32KB per connection  
2. Simplicity Benefits:  
- Minimal dependencies: No complex gRPC stack  
- Deterministic behavior: Simple protocol, predictable performance  
- Easy debugging: Human-readable message types and simple framing  
3. Optimization Benefits:  
- Custom tuning: Protocol optimized for Ray's specific use cases  
- Efficient batching: Can batch multiple small messages  
- Direct integration: Tight coupling with Ray's object model

### Trade-offs

1. Development Overhead:  
- Custom protocol maintenance: Need to maintain protocol evolution  
- Limited tooling: Fewer debugging tools compared to gRPC  
- Documentation burden: Need to document protocol thoroughly  
2. Feature Limitations:  
- No built-in features: No automatic compression, authentication, etc.  
- Local-only: Cannot be used for network communication  
- Platform-specific: Unix Domain Sockets are not available on all platforms  
3. Ecosystem Integration:  
- Non-standard: Harder for external tools to integrate  
- Learning curve: Developers need to understand custom protocol  
- Testing complexity: Need custom testing infrastructure

### When This Approach Makes Sense

Ray's custom protocol is justified because:  
1. High-frequency, low-latency requirements: Worker-raylet communication is extremely frequent  
2. Simple message patterns: Most messages are small and follow predictable patterns  
3. Local-only communication: No need for network features like load balancing  
4. Performance-critical path: This communication is on the critical path for task execution  
5. Controlled environment: Ray controls both ends of the communication

## Conclusion

Ray's custom protocol over Unix Domain Sockets represents a performance-first design decision that prioritizes the critical path of distributed computing. The choice demonstrates that there's no one-size-fits-all solution in distributed systems design.  
Key Takeaways:  
1. When performance matters most, custom protocols can provide significant advantages over general-purpose solutions  
2. Protocol simplicity can be a feature - Ray's 24-byte header and FlatBuffer payload are easy to understand and debug  
3. Hybrid approaches work well - Ray uses custom protocols for local communication and gRPC for remote communication  
4. Context matters - What works for Ray's local IPC may not work for other use cases like network communication  
This approach is similar to ScyllaDB's philosophy of optimizing the critical path, but differs in implementation details based on the specific requirements of each system.

This analysis is based on Ray's source code, particularly files in src/ray/raylet\_client/, src/ray/common/client\_connection.cc, and src/ray/raylet/format/node\_manager.fbs.

src/ray/raylet\_client/

src/ray/common/client\_connection.cc

src/ray/raylet/format/node\_manager.fbs

# Chapter 13: Port Assignment and Management

## Overview

This document provides a comprehensive explanation of how Ray allocates and manages ports for actors and tasks. Understanding this mechanism is crucial for configuring Ray clusters properly, especially in environments with strict firewall rules or limited port availability.

## Key Concepts

### 1. Single Port Pool Architecture

Ray uses a unified port pool managed by the WorkerPool class for both actors and tasks. This is not separate pools - it's one shared resource.  
Code Reference: src/ray/raylet/worker\_pool.h:834

WorkerPool

src/ray/raylet/worker\_pool.h:834

/// Keeps track of unused ports that newly-created workers can bind on.  
/// If null, workers will not be passed ports and will choose them randomly.  
std::unique\_ptr<std::queue<int>> free\_ports\_;

/// Keeps track of unused ports that newly-created workers can bind on.  
/// If null, workers will not be passed ports and will choose them randomly.  
std::unique\_ptr<std::queue<int>> free\_ports\_;

### 2. Port Allocation Model

One port per worker (regardless of CPU usage)

Both actors and tasks use the same pool

Ports are assigned when workers register with the raylet

Ports are returned to the pool when workers terminate

## Port Pool Creation

### Port Pool Initialization

The port pool is created during WorkerPool construction with ports from either:  
1. Port Range (min\_worker\_port to max\_worker\_port)  
2. Explicit Port List (worker\_port\_list)  
Code Reference: src/ray/raylet/worker\_pool.cc:148-161

WorkerPool

src/ray/raylet/worker\_pool.cc:148-161

// Initialize free ports list with all ports in the specified range.  
if (!worker\_ports.empty()) {  
free\_ports\_ = std::make\_unique<std::queue<int>>();  
for (int port : worker\_ports) {  
free\_ports\_->push(port);  
}  
} else if (min\_worker\_port != 0 && max\_worker\_port != 0) {  
free\_ports\_ = std::make\_unique<std::queue<int>>();  
if (max\_worker\_port == 0) {  
max\_worker\_port = 65535; // Maximum valid port number  
}  
for (int port = min\_worker\_port; port <= max\_worker\_port; port++) {  
free\_ports\_->push(port);  
}  
}

// Initialize free ports list with all ports in the specified range.  
if (!worker\_ports.empty()) {  
free\_ports\_ = std::make\_unique<std::queue<int>>();  
for (int port : worker\_ports) {  
free\_ports\_->push(port);  
}  
} else if (min\_worker\_port != 0 && max\_worker\_port != 0) {  
free\_ports\_ = std::make\_unique<std::queue<int>>();  
if (max\_worker\_port == 0) {  
max\_worker\_port = 65535; // Maximum valid port number  
}  
for (int port = min\_worker\_port; port <= max\_worker\_port; port++) {  
free\_ports\_->push(port);  
}  
}

### Configuration Options

#### Method 1: Port Range

ray start --min-worker-port=10000 --max-worker-port=10100  
# Python API  
ray.init(min\_worker\_port=10000, max\_worker\_port=10100)

ray start --min-worker-port=10000 --max-worker-port=10100  
# Python API  
ray.init(min\_worker\_port=10000, max\_worker\_port=10100)

#### Method 2: Explicit Port List

ray start --worker-port-list="10000,10001,10002,10003"  
# Python API  
ray.init(worker\_port\_list=[10000, 10001, 10002, 10003])

ray start --worker-port-list="10000,10001,10002,10003"  
# Python API  
ray.init(worker\_port\_list=[10000, 10001, 10002, 10003])

Code Reference: src/ray/raylet/main.cc:55-60

src/ray/raylet/main.cc:55-60

DEFINE\_int32(min\_worker\_port, 0, "The lowest port that workers' gRPC servers will bind on.");  
DEFINE\_int32(max\_worker\_port, 0, "The highest port that workers' gRPC servers will bind on.");  
DEFINE\_string(worker\_port\_list, "", "An explicit list of ports that workers' gRPC servers will bind on.");

DEFINE\_int32(min\_worker\_port, 0, "The lowest port that workers' gRPC servers will bind on.");  
DEFINE\_int32(max\_worker\_port, 0, "The highest port that workers' gRPC servers will bind on.");  
DEFINE\_string(worker\_port\_list, "", "An explicit list of ports that workers' gRPC servers will bind on.");

## Port Assignment Process

### Worker Registration and Port Assignment

When any worker (task or actor) starts, it follows this exact process:  
Code Reference: src/ray/raylet/worker\_pool.cc:796-812

src/ray/raylet/worker\_pool.cc:796-812

// The port that this worker's gRPC server should listen on  
int port = 0;  
Status status = GetNextFreePort(&port);  
if (!status.ok()) {  
return PopWorkerStatus::Failed;  
}  
worker->SetAssignedPort(port);

// The port that this worker's gRPC server should listen on  
int port = 0;  
Status status = GetNextFreePort(&port);  
if (!status.ok()) {  
return PopWorkerStatus::Failed;  
}  
worker->SetAssignedPort(port);

### Port Allocation Function

Code Reference: src/ray/raylet/worker\_pool.cc:683-701

src/ray/raylet/worker\_pool.cc:683-701

Status WorkerPool::GetNextFreePort(int \*port) {  
if (free\_ports\_ == nullptr || free\_ports\_->empty()) {  
return Status::Invalid(  
"No available ports. Please specify a wider port range using --min-worker-port and "  
"--max-worker-port.");  
}  
// Try up to the current number of ports.  
int current\_size = free\_ports\_->size();  
for (int i = 0; i < current\_size; i++) {  
\*port = free\_ports\_->front();  
free\_ports\_->pop();  
if (IsPortAvailable(\*port)) {  
return Status::OK();  
} else {  
// Port is occupied, try next one  
free\_ports\_->push(\*port);  
}  
}  
return Status::Invalid(  
"No available ports. Please specify a wider port range using --min-worker-port and "  
"--max-worker-port.");  
}

Status WorkerPool::GetNextFreePort(int \*port) {  
if (free\_ports\_ == nullptr || free\_ports\_->empty()) {  
return Status::Invalid(  
"No available ports. Please specify a wider port range using --min-worker-port and "  
"--max-worker-port.");  
}  
// Try up to the current number of ports.  
int current\_size = free\_ports\_->size();  
for (int i = 0; i < current\_size; i++) {  
\*port = free\_ports\_->front();  
free\_ports\_->pop();  
if (IsPortAvailable(\*port)) {  
return Status::OK();  
} else {  
// Port is occupied, try next one  
free\_ports\_->push(\*port);  
}  
}  
return Status::Invalid(  
"No available ports. Please specify a wider port range using --min-worker-port and "  
"--max-worker-port.");  
}

## Actor vs Task Port Usage

### Actors: Long-lived Port Dedication

@ray.remote  
class MyActor:  
def method(self):  
return "Hello"  
actor = MyActor.remote()

@ray.remote  
class MyActor:  
def method(self):  
return "Hello"  
actor = MyActor.remote()

Characteristics:  
- Dedicated Port: Each actor gets its own port  
- Long-lived: Port is held until actor terminates/dies  
- Persistent: Same port for all method calls on the actor  
- gRPC Server: Actor runs a gRPC server on its assigned port

### Tasks: Short-lived Port Usage

@ray.remote  
def my\_task():  
return "Hello"  
future = my\_task.remote()

@ray.remote  
def my\_task():  
return "Hello"  
future = my\_task.remote()

Characteristics:  
- Temporary Port: Task gets port from pool when worker is assigned  
- Short-lived: Port returned to pool when task completes  
- Worker Reuse: Same worker (and port) can execute multiple sequential tasks  
- Pooled Workers: Tasks share a pool of workers

## Worker Pool Size Limits

### The num\_workers\_soft\_limit Configuration

num\_workers\_soft\_limit

This is the critical parameter that controls maximum port usage.  
Code Reference: src/ray/raylet/node\_manager.cc:130-150

src/ray/raylet/node\_manager.cc:130-150

[this, config]() {  
// Callback to determine the maximum number of idle workers to keep around.  
if (config.num\_workers\_soft\_limit >= 0) {  
return config.num\_workers\_soft\_limit;  
}  
// If no limit is provided, use the available number of CPUs,  
// assuming that each incoming task will likely require 1 CPU.  
return static\_cast<int64\_t>(  
cluster\_resource\_scheduler\_->GetLocalResourceManager()  
.GetLocalAvailableCpus());  
}

[this, config]() {  
// Callback to determine the maximum number of idle workers to keep around.  
if (config.num\_workers\_soft\_limit >= 0) {  
return config.num\_workers\_soft\_limit;  
}  
// If no limit is provided, use the available number of CPUs,  
// assuming that each incoming task will likely require 1 CPU.  
return static\_cast<int64\_t>(  
cluster\_resource\_scheduler\_->GetLocalResourceManager()  
.GetLocalAvailableCpus());  
}

Default Behavior: num\_workers\_soft\_limit = -1 → defaults to CPU count  
Code Reference: src/ray/common/ray\_config\_def.h:617-624

num\_workers\_soft\_limit = -1

src/ray/common/ray\_config\_def.h:617-624

/// The soft limit of the number of workers to keep around.  
/// We apply this limit to the idle workers instead of total workers,  
/// because the total number of workers used depends on the  
/// application. -1 means using the available number of CPUs.  
RAY\_CONFIG(int64\_t, num\_workers\_soft\_limit, -1)

/// The soft limit of the number of workers to keep around.  
/// We apply this limit to the idle workers instead of total workers,  
/// because the total number of workers used depends on the  
/// application. -1 means using the available number of CPUs.  
RAY\_CONFIG(int64\_t, num\_workers\_soft\_limit, -1)

### Configuration Examples

ray start --num-workers-soft-limit=50  
# Python API  
ray.init(num\_workers\_soft\_limit=50)

ray start --num-workers-soft-limit=50  
# Python API  
ray.init(num\_workers\_soft\_limit=50)

## Port Exhaustion Scenarios

### When Do You Run Out of Ports?

#### Scenario 1: Too Many Concurrent Actors

# Problem: Creating 100 long-lived actors  
actors = [MyActor.remote() for \_ in range(100)] # ❌ FAIL after 16

# Problem: Creating 100 long-lived actors  
actors = [MyActor.remote() for \_ in range(100)] # ❌ FAIL after 16

#### Scenario 2: Fractional CPU Tasks

# Problem: Tasks with fractional CPU requirements  
@ray.remote(num\_cpus=0.1) # Only 0.1 CPU per task  
def light\_task():  
return "done"  
# Can theoretically run 160 concurrent tasks (16 CPUs / 0.1)  
futures = [light\_task.remote() for \_ in range(160)] # ❌ FAIL after 16

# Problem: Tasks with fractional CPU requirements  
@ray.remote(num\_cpus=0.1) # Only 0.1 CPU per task  
def light\_task():  
return "done"  
# Can theoretically run 160 concurrent tasks (16 CPUs / 0.1)  
futures = [light\_task.remote() for \_ in range(160)] # ❌ FAIL after 16

### Error Messages

Code Reference: src/ray/raylet/worker\_pool.cc:693-701

src/ray/raylet/worker\_pool.cc:693-701

return Status::Invalid(  
"No available ports. Please specify a wider port range using --min-worker-port and "  
"--max-worker-port.");

return Status::Invalid(  
"No available ports. Please specify a wider port range using --min-worker-port and "  
"--max-worker-port.");

## Best Practices & Solutions

### 1. Calculate Required Ports

Required Ports = Max Concurrent Workers  
= Max(Long-lived Actors + Peak Concurrent Tasks)

Required Ports = Max Concurrent Workers  
= Max(Long-lived Actors + Peak Concurrent Tasks)

### 2. Configure Appropriate Port Range

# For 1000 concurrent workers  
ray start --min-worker-port=10000 --max-worker-port=11000 --num-workers-soft-limit=1000

# For 1000 concurrent workers  
ray start --min-worker-port=10000 --max-worker-port=11000 --num-workers-soft-limit=1000

### 3. Use Explicit Port Lists for Control

ray start --worker-port-list="10000,10001,10002,10003,10004"

ray start --worker-port-list="10000,10001,10002,10003,10004"

### 4. Monitor Port Usage

# Check cluster resources  
print(ray.cluster\_resources())  
# Check current worker count  
import ray.\_private.worker  
print(len(ray.\_private.worker.global\_worker.core\_worker.get\_all\_reference\_counts()))

# Check cluster resources  
print(ray.cluster\_resources())  
# Check current worker count  
import ray.\_private.worker  
print(len(ray.\_private.worker.global\_worker.core\_worker.get\_all\_reference\_counts()))

## Advanced Configuration Examples

### Large Cluster Setup (1000 nodes)

# Head node  
ray start --head \  
--port=6379 \  
--min-worker-port=20000 \  
--max-worker-port=25000 \  
--num-workers-soft-limit=5000  
# Worker nodes  
ray start --address=head\_ip:6379 \  
--min-worker-port=20000 \  
--max-worker-port=25000 \  
--num-workers-soft-limit=5000

# Head node  
ray start --head \  
--port=6379 \  
--min-worker-port=20000 \  
--max-worker-port=25000 \  
--num-workers-soft-limit=5000  
# Worker nodes  
ray start --address=head\_ip:6379 \  
--min-worker-port=20000 \  
--max-worker-port=25000 \  
--num-workers-soft-limit=5000

### Actor-Heavy Workload

# For 500 concurrent actors per node  
ray start --min-worker-port=30000 --max-worker-port=30500 --num-workers-soft-limit=500

# For 500 concurrent actors per node  
ray start --min-worker-port=30000 --max-worker-port=30500 --num-workers-soft-limit=500

### Mixed Workload (Actors + Tasks)

# 100 actors + 400 peak concurrent tasks = 500 total  
ray start --min-worker-port=40000 --max-worker-port=40500 --num-workers-soft-limit=500

# 100 actors + 400 peak concurrent tasks = 500 total  
ray start --min-worker-port=40000 --max-worker-port=40500 --num-workers-soft-limit=500

## Port Usage Summary

Total Ports Per Node = Core Ray Ports + Worker Ports  
Core Ray Ports = 7 (fixed)  
- Node Manager: 1  
- Object Manager: 1  
- Metrics Agent: 1  
- Runtime Env Agent: 1  
- Dashboard Agent: 1  
- Metrics Export: 1  
- Ray Client Server: 1 (head only)  
Worker Ports = num\_workers\_soft\_limit (configurable)  
- Default: CPU count  
- Configurable: --num-workers-soft-limit  
Example for 16-CPU node:  
Total = 7 + 16 = 23 ports minimum

Total Ports Per Node = Core Ray Ports + Worker Ports  
Core Ray Ports = 7 (fixed)  
- Node Manager: 1  
- Object Manager: 1  
- Metrics Agent: 1  
- Runtime Env Agent: 1  
- Dashboard Agent: 1  
- Metrics Export: 1  
- Ray Client Server: 1 (head only)  
Worker Ports = num\_workers\_soft\_limit (configurable)  
- Default: CPU count  
- Configurable: --num-workers-soft-limit  
Example for 16-CPU node:  
Total = 7 + 16 = 23 ports minimum

## Common Issues and Solutions

### Issue 1: Port Exhaustion with Fractional CPU Tasks

Problem: num\_workers\_soft\_limit defaults to CPU count, but fractional CPU tasks can exceed this.  
Solution: Increase num\_workers\_soft\_limit and port range:

num\_workers\_soft\_limit

num\_workers\_soft\_limit

ray start --num-workers-soft\_limit=100 --min-worker-port=20000 --max-worker-port=20100

ray start --num-workers-soft\_limit=100 --min-worker-port=20000 --max-worker-port=20100

### Issue 2: Firewall Restrictions

Problem: Need to specify exact ports for firewall rules.  
Solution: Use explicit port lists:

ray start --worker-port-list="10000,10001,10002,10003"

ray start --worker-port-list="10000,10001,10002,10003"

### Issue 3: Actor Port Leakage

Problem: Dead actors not releasing ports properly.  
Solution: Ensure proper actor cleanup:

# Explicit cleanup  
ray.kill(actor)  
del actor  
# Or use context managers for automatic cleanup

# Explicit cleanup  
ray.kill(actor)  
del actor  
# Or use context managers for automatic cleanup

## Code References Summary

src/ray/raylet/worker\_pool.cc

GetNextFreePort()

PopWorker()

src/ray/raylet/main.cc

src/ray/raylet/node\_manager.cc

num\_workers\_soft\_limit

src/ray/raylet/worker\_pool.h

free\_ports\_

num\_workers\_soft\_limit

ray.get()

ray.get()

// Code Reference: src/ray/raylet/local\_task\_manager.cc  
bool LocalTaskManager::ReleaseCpuResourcesFromBlockedWorker(  
std::shared\_ptr<WorkerInterface> worker) {  
// CPU resources are released back to the scheduler  
}

// Code Reference: src/ray/raylet/local\_task\_manager.cc  
bool LocalTaskManager::ReleaseCpuResourcesFromBlockedWorker(  
std::shared\_ptr<WorkerInterface> worker) {  
// CPU resources are released back to the scheduler  
}

The worker's CPU allocation is returned to the resource pool

Other tasks can use those CPU resources

This prevents deadlocks in resource-constrained environments

Port Resource Management:

// Code Reference: src/ray/raylet/worker.h  
/// Whether the worker is blocked. Workers become blocked in a `ray.get`  
bool blocked\_;

// Code Reference: src/ray/raylet/worker.h  
/// Whether the worker is blocked. Workers become blocked in a `ray.get`  
bool blocked\_;

The worker keeps its gRPC server port open

Port remains allocated until task completely finishes

This is necessary for receiving results and maintaining communication  
Why Ports Stay Open:

The worker's gRPC server must remain accessible to receive the result

Communication channels with raylet must stay active

The worker process itself continues running (just blocked)

### Q2: Who assigns tasks to raylet and via which port?

Answer: GCS (Global Control Service) assigns tasks to raylets via the Node Manager Port  
Complete Task Assignment Flow:

1. Task Submission:  
Worker/Driver → GCS (via GCS Port ~6379)  
2. Task Scheduling:  
GCS → Raylet (via Node Manager Port ~10001)  
3. Worker Assignment:  
Raylet → Worker (via Worker's gRPC Port from pool)  
4. Result Return:  
Worker → Raylet → GCS → Requester

1. Task Submission:  
Worker/Driver → GCS (via GCS Port ~6379)  
2. Task Scheduling:  
GCS → Raylet (via Node Manager Port ~10001)  
3. Worker Assignment:  
Raylet → Worker (via Worker's gRPC Port from pool)  
4. Result Return:  
Worker → Raylet → GCS → Requester

Code References:

// Node Manager Port Configuration  
// src/ray/raylet/main.cc:48  
DEFINE\_int32(node\_manager\_port, -1, "The port of node manager.");  
// GCS to Raylet Communication  
// Tasks are assigned via gRPC calls to the Node Manager service  
// The raylet listens on node\_manager\_port for task assignments

// Node Manager Port Configuration  
// src/ray/raylet/main.cc:48  
DEFINE\_int32(node\_manager\_port, -1, "The port of node manager.");  
// GCS to Raylet Communication  
// Tasks are assigned via gRPC calls to the Node Manager service  
// The raylet listens on node\_manager\_port for task assignments

Port Usage:  
- GCS Port: For initial task submission and cluster coordination  
- Node Manager Port: For task assignment from GCS to raylet  
- Worker Ports: For task execution and inter-task communication

### Q3: What is Ray communication for tasks on the same node?

Answer: Tasks on the same node communicate directly via worker ports, bypassing raylet for task-to-task calls.  
Same-Node Communication Flow:

Task A (Port 10000) → Direct gRPC → Task B (Port 10001)  
↑  
(No raylet involvement)

Task A (Port 10000) → Direct gRPC → Task B (Port 10001)  
↑  
(No raylet involvement)

Cross-Node Communication Flow:

Task A (Node 1, Port 10000) → Raylet 1 → Network → Raylet 2 → Task B (Node 2, Port 10001)

Task A (Node 1, Port 10000) → Raylet 1 → Network → Raylet 2 → Task B (Node 2, Port 10001)

Why Direct Communication:  
- Performance: Eliminates raylet as middleman  
- Efficiency: Reduces network hops and latency  
- Scalability: Reduces load on raylet for local communication

### Q4: Can ray.get() cause port starvation?

YES! This is a critical production consideration.  
Scenario:  
- Available ports: 64 (typical small range)  
- Running tasks: 60 (all blocked on ray.get())  
- New task requests: 10  
Result:  
- All 64 ports occupied by blocked workers  
- New tasks cannot start → Port starvation  
- Cluster appears "hung" despite available CPU  
Solutions:  
1. Increase Port Range:

ray.get()

ray start --min-worker-port=10000 --max-worker-port=20000 # 10K ports

ray start --min-worker-port=10000 --max-worker-port=20000 # 10K ports

Tune Worker Pool:

ray start --num-workers-soft\_limit=1000 # Allow more concurrent workers

ray start --num-workers-soft\_limit=1000 # Allow more concurrent workers

Application Design:

# Instead of blocking many workers  
futures = [task.remote() for \_ in range(1000)]  
results = ray.get(futures) # Single blocking point  
# Better: Batch processing  
batch\_size = 50  
for batch in chunks(futures, batch\_size):  
ray.get(batch) # Process in smaller batches

# Instead of blocking many workers  
futures = [task.remote() for \_ in range(1000)]  
results = ray.get(futures) # Single blocking point  
# Better: Batch processing  
batch\_size = 50  
for batch in chunks(futures, batch\_size):  
ray.get(batch) # Process in smaller batches

### Q5: Port allocation for different worker types

All worker types use the same port pool:  
| Worker Type | Port Source | Port Lifetime | Notes |  
|-------------|-------------|---------------|--------|  
| Actor Workers | Worker port pool | Until actor dies | Dedicated, long-lived |  
| Task Workers | Worker port pool | Until task completes | Shared, short-lived |  
| Driver Workers | Worker port pool | Until driver exits | Dedicated, session-lived |  
Code Reference:

// src/ray/raylet/worker\_pool.cc:683-700  
Status WorkerPool::GetNextFreePort(int \*port) {  
// Same pool used for ALL worker types  
if (free\_ports\_->empty()) {  
return Status::Invalid("No available ports...");  
}  
\*port = free\_ports\_->front();  
free\_ports\_->pop();  
return Status::OK();  
}

// src/ray/raylet/worker\_pool.cc:683-700  
Status WorkerPool::GetNextFreePort(int \*port) {  
// Same pool used for ALL worker types  
if (free\_ports\_->empty()) {  
return Status::Invalid("No available ports...");  
}  
\*port = free\_ports\_->front();  
free\_ports\_->pop();  
return Status::OK();  
}

### Q6: Maximum theoretical port usage

Calculation:

Max Ports = min(  
max\_worker\_port - min\_worker\_port + 1, // Port range size  
num\_workers\_soft\_limit, // Worker pool limit  
System file descriptor limit // OS limit  
)

Max Ports = min(  
max\_worker\_port - min\_worker\_port + 1, // Port range size  
num\_workers\_soft\_limit, // Worker pool limit  
System file descriptor limit // OS limit  
)

Example:

Node: 16 CPUs  
Port Range: 10000-65535 (55,536 ports)  
Worker Limit: Default = 16 (CPU count)  
Actual Max: 16 ports (limited by worker pool)

Node: 16 CPUs  
Port Range: 10000-65535 (55,536 ports)  
Worker Limit: Default = 16 (CPU count)  
Actual Max: 16 ports (limited by worker pool)

To Use More Ports:

# Increase worker pool beyond CPU count  
ray start --num\_workers\_soft\_limit=1000 --min-worker-port=10000 --max-worker-port=11000

# Increase worker pool beyond CPU count  
ray start --num\_workers\_soft\_limit=1000 --min-worker-port=10000 --max-worker-port=11000

## Production Recommendations

Based on the above Q&A, here are production recommendations:

### Port Planning:

Calculate realistic port needs: (Expected concurrent tasks + actors) \* 1.5

(Expected concurrent tasks + actors) \* 1.5

Set generous ranges: Better to over-provision than under-provision

Monitor port usage: Track free\_ports\_ queue size

free\_ports\_

### Application Design:

Minimize blocking: Reduce ray.get() calls in tight loops

ray.get()

Batch operations: Process results in batches, not individually

Use futures wisely: Collect futures first, then ray.get() in batches

ray.get()

### Configuration:

Explicit port lists for controlled environments

Wide port ranges for dynamic workloads

Monitor worker pool metrics in production  
This comprehensive understanding of Ray's port management will help you design robust, scalable Ray applications that avoid common port-related pitfalls in production environments.

## Sequence Diagrams and Flow Charts

This section provides visual representations of Ray's port allocation and communication flows to help understand the system architecture.

### 1. Port Pool Initialization Flow

📊 SEQUENCE DIAGRAM: Process Flow and Interactions  
  
📁 Text-based diagram (backup)

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

┌─────────────────┐ ┌──────────────────┐ ┌─────────────────┐  
│ Raylet Start │ │ WorkerPool │ │ Port Queue │  
│ │ │ Constructor │ │ │  
└─────────┬───────┘ └─────────┬────────┘ └─────────┬───────┘  
│ │ │  
│ 1. Initialize │ │  
├─────────────────────→│ │  
│ │ │  
│ │ 2. Create free\_ports\_ │  
│ ├──────────────────────→│  
│ │ │  
│ │ 3. Parse port range │  
│ │ or explicit list │  
│ │ │  
│ │ 4. Push ports to queue│  
│ │ (10000→10100) │  
│ ├──────────────────────→│  
│ │ │  
│ 5. Pool Ready │ │  
│←─────────────────────┤ │  
│ │ │  
Port Range: --min-worker-port=10000 --max-worker-port=10100  
Result: 101 ports in queue [10000, 10001, 10002, ..., 10100]

┌─────────────────┐ ┌──────────────────┐ ┌─────────────────┐  
│ Raylet Start │ │ WorkerPool │ │ Port Queue │  
│ │ │ Constructor │ │ │  
└─────────┬───────┘ └─────────┬────────┘ └─────────┬───────┘  
│ │ │  
│ 1. Initialize │ │  
├─────────────────────→│ │  
│ │ │  
│ │ 2. Create free\_ports\_ │  
│ ├──────────────────────→│  
│ │ │  
│ │ 3. Parse port range │  
│ │ or explicit list │  
│ │ │  
│ │ 4. Push ports to queue│  
│ │ (10000→10100) │  
│ ├──────────────────────→│  
│ │ │  
│ 5. Pool Ready │ │  
│←─────────────────────┤ │  
│ │ │  
Port Range: --min-worker-port=10000 --max-worker-port=10100  
Result: 101 ports in queue [10000, 10001, 10002, ..., 10100]

### 2. Worker Registration and Port Assignment Sequence

📊 SEQUENCE DIAGRAM: Process Flow and Interactions  
  
📁 Text-based diagram (backup)

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

┌───────────┐ ┌─────────────┐ ┌─────────────┐ ┌─────────────┐  
│ Worker │ │ Raylet │ │ WorkerPool │ │ Port Queue │  
│ Process │ │ │ │ │ │ │  
└─────┬─────┘ └──────┬──────┘ └──────┬──────┘ └──────┬──────┘  
│ │ │ │  
│ 1. Register │ │ │  
├──────────────→│ │ │  
│ │ │ │  
│ │ 2. PopWorker() │ │  
│ ├───────────────→│ │  
│ │ │ │  
│ │ │ 3. GetNextFreePort()  
│ │ ├───────────────→│  
│ │ │ │  
│ │ │ 4. port=10005 │  
│ │ │←───────────────┤  
│ │ │ │  
│ │ 5. SetAssignedPort(10005) │  
│ │ │ │  
│ 6. Port: 10005│ │ │  
│←──────────────┤ │ │  
│ │ │ │  
│ 7. Start gRPC │ │ │  
│ Server on │ │ │  
│ port 10005 │ │ │  
│ │ │ │  
Result: Worker now has dedicated port 10005 for its gRPC server

┌───────────┐ ┌─────────────┐ ┌─────────────┐ ┌─────────────┐  
│ Worker │ │ Raylet │ │ WorkerPool │ │ Port Queue │  
│ Process │ │ │ │ │ │ │  
└─────┬─────┘ └──────┬──────┘ └──────┬──────┘ └──────┬──────┘  
│ │ │ │  
│ 1. Register │ │ │  
├──────────────→│ │ │  
│ │ │ │  
│ │ 2. PopWorker() │ │  
│ ├───────────────→│ │  
│ │ │ │  
│ │ │ 3. GetNextFreePort()  
│ │ ├───────────────→│  
│ │ │ │  
│ │ │ 4. port=10005 │  
│ │ │←───────────────┤  
│ │ │ │  
│ │ 5. SetAssignedPort(10005) │  
│ │ │ │  
│ 6. Port: 10005│ │ │  
│←──────────────┤ │ │  
│ │ │ │  
│ 7. Start gRPC │ │ │  
│ Server on │ │ │  
│ port 10005 │ │ │  
│ │ │ │  
Result: Worker now has dedicated port 10005 for its gRPC server

### 3. Task Assignment Flow Diagram

📊 SEQUENCE DIAGRAM: Process Flow and Interactions  
  
📁 Text-based diagram (backup)

[DIAGRAM: 📊 SEQUENCE DIAGRAM: Process Flow and Interactions]

┌─────────────┐ ┌─────────────┐ ┌─────────────┐ ┌─────────────┐  
│ Driver/ │ │ GCS │ │ Raylet │ │ Worker │  
│ Client │ │ │ │ │ │ │  
└──────┬──────┘ └──────┬──────┘ └──────┬──────┘ └──────┬──────┘  
│ │ │ │  
│ 1. task.remote() │ │ │  
├─────────────────→│ │ │  
│ │ │ │  
│ │ 2. Schedule Task │ │  
│ │ (find node) │ │  
│ │ │ │  
│ │ 3. RequestWorker │ │  
│ │ Lease │ │  
│ ├─────────────────→│ │  
│ │ │ │  
│ │ │ 4. PopWorker() │  
│ │ │ (assign port) │  
│ │ │ │  
│ │ 5. WorkerLease │ │  
│ │ (port info) │ │  
│ │←─────────────────┤ │  
│ │ │ │  
│ │ 6. SubmitTask │ │  
│ │ (to worker) │ │  
│ ├─────────────────────────────────────→│  
│ │ │ │  
│ │ │ │ 7. Execute  
│ │ │ │ Task  
│ │ │ │  
│ │ 8. Task Result │ │  
│ │←─────────────────────────────────────┤  
│ │ │ │  
│ 9. ray.get() │ │ │  
│ result │ │ │  
│←─────────────────┤ │ │  
Ports Used:  
- GCS Port: ~6379 (Driver → GCS)  
- Node Manager Port: ~10001 (GCS → Raylet)  
- Worker Port: from pool, e.g., 10005 (Task execution)

┌─────────────┐ ┌─────────────┐ ┌─────────────┐ ┌─────────────┐  
│ Driver/ │ │ GCS │ │ Raylet │ │ Worker │  
│ Client │ │ │ │ │ │ │  
└──────┬──────┘ └──────┬──────┘ └──────┬──────┘ └──────┬──────┘  
│ │ │ │  
│ 1. task.remote() │ │ │  
├─────────────────→│ │ │  
│ │ │ │  
│ │ 2. Schedule Task │ │  
│ │ (find node) │ │  
│ │ │ │  
│ │ 3. RequestWorker │ │  
│ │ Lease │ │  
│ ├─────────────────→│ │  
│ │ │ │  
│ │ │ 4. PopWorker() │  
│ │ │ (assign port) │  
│ │ │ │  
│ │ 5. WorkerLease │ │  
│ │ (port info) │ │  
│ │←─────────────────┤ │  
│ │ │ │  
│ │ 6. SubmitTask │ │  
│ │ (to worker) │ │  
│ ├─────────────────────────────────────→│  
│ │ │ │  
│ │ │ │ 7. Execute  
│ │ │ │ Task  
│ │ │ │  
│ │ 8. Task Result │ │  
│ │←─────────────────────────────────────┤  
│ │ │ │  
│ 9. ray.get() │ │ │  
│ result │ │ │  
│←─────────────────┤ │ │  
Ports Used:  
- GCS Port: ~6379 (Driver → GCS)  
- Node Manager Port: ~10001 (GCS → Raylet)  
- Worker Port: from pool, e.g., 10005 (Task execution)

### 4. Actor vs Task Port Usage Lifecycle

🔧 TECHNICAL DIAGRAM: System Architecture  
🔧 TECHNICAL DIAGRAM: System Architecture  
  
📁 Text-based diagram (backup)

[DIAGRAM: 🔧 TECHNICAL DIAGRAM: System Architecture]

[DIAGRAM: 🔧 TECHNICAL DIAGRAM: System Architecture]

ACTOR LIFECYCLE:  
┌─────────────────────────────────────────────────────────────────┐  
│ Actor Lifetime │  
├─────────────────────────────────────────────────────────────────┤  
│ Create → Get Port 10005 → Keep Port → Method Calls → Die │  
│ ↓ ↓ ↓ ↓ ↓ │  
│ Start Dedicated Port Held Same Port Return Port │  
│ Port Throughout Used to Pool │  
└─────────────────────────────────────────────────────────────────┘  
TASK LIFECYCLE:  
┌───────────┐ ┌───────────┐ ┌───────────┐ ┌───────────┐ ┌───────────┐  
│ Task A │ │ Task B │ │ Task C │ │ Task D │ │ Task E │  
├───────────┤ ├───────────┤ ├───────────┤ ├───────────┤ ├───────────┤  
│Port: 10005│ │Port: 10005│ │Port: 10006│ │Port: 10005│ │Port: 10007│  
│Worker: W1 │ │Worker: W1 │ │Worker: W2 │ │Worker: W1 │ │Worker: W3 │  
└───────────┘ └───────────┘ └───────────┘ └───────────┘ └───────────┘  
↓ ↓ ↓ ↓ ↓  
Finish Reuse New Port Reuse New Port  
Same Worker (W1 busy) Same Worker (W1,W2 busy)  
Key Difference:  
- Actors: 1 Actor = 1 Dedicated Port (Long-term)  
- Tasks: 1 Worker = 1 Port, Multiple Tasks Share Worker (Short-term)

ACTOR LIFECYCLE:  
┌─────────────────────────────────────────────────────────────────┐  
│ Actor Lifetime │  
├─────────────────────────────────────────────────────────────────┤  
│ Create → Get Port 10005 → Keep Port → Method Calls → Die │  
│ ↓ ↓ ↓ ↓ ↓ │  
│ Start Dedicated Port Held Same Port Return Port │  
│ Port Throughout Used to Pool │  
└─────────────────────────────────────────────────────────────────┘  
TASK LIFECYCLE:  
┌───────────┐ ┌───────────┐ ┌───────────┐ ┌───────────┐ ┌───────────┐  
│ Task A │ │ Task B │ │ Task C │ │ Task D │ │ Task E │  
├───────────┤ ├───────────┤ ├───────────┤ ├───────────┤ ├───────────┤  
│Port: 10005│ │Port: 10005│ │Port: 10006│ │Port: 10005│ │Port: 10007│  
│Worker: W1 │ │Worker: W1 │ │Worker: W2 │ │Worker: W1 │ │Worker: W3 │  
└───────────┘ └───────────┘ └───────────┘ └───────────┘ └───────────┘  
↓ ↓ ↓ ↓ ↓  
Finish Reuse New Port Reuse New Port  
Same Worker (W1 busy) Same Worker (W1,W2 busy)  
Key Difference:  
- Actors: 1 Actor = 1 Dedicated Port (Long-term)  
- Tasks: 1 Worker = 1 Port, Multiple Tasks Share Worker (Short-term)

### 5. Same-Node vs Cross-Node Communication Flow

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships  
  
📁 Text-based diagram (backup)

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

SAME NODE COMMUNICATION (Direct):  
┌─────────────┐ ┌─────────────┐  
│ Task A │ Direct gRPC Call │ Task B │  
│ (Port 10005)│──────────────────────────→│ (Port 10006)│  
│ Worker 1 │ │ Worker 2 │  
└─────────────┘ └─────────────┘  
↑ ↑  
└─────────── Same Raylet ──────────────────┘  
Benefits: Low latency, No raylet overhead, High throughput  
CROSS NODE COMMUNICATION (Via Raylet):  
┌─────────────┐ ┌─────────────┐ ┌─────────────┐ ┌─────────────┐  
│ Task A │ │ Raylet 1 │ │ Raylet 2 │ │ Task B │  
│ (Port 10005)│───→│(Node Mgr │───→│(Node Mgr │───→│ (Port 10006)│  
│ Node 1 │ │ Port 10001) │ │ Port 10001) │ │ Node 2 │  
└─────────────┘ └─────────────┘ └─────────────┘ └─────────────┘  
Benefits: Network routing, Load balancing, Fault tolerance

SAME NODE COMMUNICATION (Direct):  
┌─────────────┐ ┌─────────────┐  
│ Task A │ Direct gRPC Call │ Task B │  
│ (Port 10005)│──────────────────────────→│ (Port 10006)│  
│ Worker 1 │ │ Worker 2 │  
└─────────────┘ └─────────────┘  
↑ ↑  
└─────────── Same Raylet ──────────────────┘  
Benefits: Low latency, No raylet overhead, High throughput  
CROSS NODE COMMUNICATION (Via Raylet):  
┌─────────────┐ ┌─────────────┐ ┌─────────────┐ ┌─────────────┐  
│ Task A │ │ Raylet 1 │ │ Raylet 2 │ │ Task B │  
│ (Port 10005)│───→│(Node Mgr │───→│(Node Mgr │───→│ (Port 10006)│  
│ Node 1 │ │ Port 10001) │ │ Port 10001) │ │ Node 2 │  
└─────────────┘ └─────────────┘ └─────────────┘ └─────────────┘  
Benefits: Network routing, Load balancing, Fault tolerance

### 6. Port Exhaustion Scenario Diagram

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships  
  
📁 Text-based diagram (backup)

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

NORMAL OPERATION:  
Port Pool: [10000, 10001, 10002, 10003, 10004] (5 ports available)  
Active Workers: 2  
Available Ports: 3  
Status: ✅ HEALTHY  
┌─────────────┐ ┌─────────────┐ ┌─────────────┐  
│ Worker 1 │ │ Worker 2 │ │ Pool │  
│ Port: 10000 │ │ Port: 10001 │ │ [10002, │  
│ Status: BUSY│ │ Status: BUSY│ │ 10003, │  
└─────────────┘ └─────────────┘ │ 10004] │  
└─────────────┘  
PORT EXHAUSTION:  
Port Pool: [] (0 ports available)  
Active Workers: 5 (all blocked on ray.get())  
Available Ports: 0  
Status: ❌ STARVED  
┌─────────────┐ ┌─────────────┐ ┌─────────────┐ ┌─────────────┐ ┌─────────────┐  
│ Worker 1 │ │ Worker 2 │ │ Worker 3 │ │ Worker 4 │ │ Worker 5 │  
│ Port: 10000 │ │ Port: 10001 │ │ Port: 10002 │ │ Port: 10003 │ │ Port: 10004 │  
│BLOCKED: │ │BLOCKED: │ │BLOCKED: │ │BLOCKED: │ │BLOCKED: │  
│ray.get() │ │ray.get() │ │ray.get() │ │ray.get() │ │ray.get() │  
└─────────────┘ └─────────────┘ └─────────────┘ └─────────────┘ └─────────────┘  
New Task Request → ❌ FAIL: "No available ports"

NORMAL OPERATION:  
Port Pool: [10000, 10001, 10002, 10003, 10004] (5 ports available)  
Active Workers: 2  
Available Ports: 3  
Status: ✅ HEALTHY  
┌─────────────┐ ┌─────────────┐ ┌─────────────┐  
│ Worker 1 │ │ Worker 2 │ │ Pool │  
│ Port: 10000 │ │ Port: 10001 │ │ [10002, │  
│ Status: BUSY│ │ Status: BUSY│ │ 10003, │  
└─────────────┘ └─────────────┘ │ 10004] │  
└─────────────┘  
PORT EXHAUSTION:  
Port Pool: [] (0 ports available)  
Active Workers: 5 (all blocked on ray.get())  
Available Ports: 0  
Status: ❌ STARVED  
┌─────────────┐ ┌─────────────┐ ┌─────────────┐ ┌─────────────┐ ┌─────────────┐  
│ Worker 1 │ │ Worker 2 │ │ Worker 3 │ │ Worker 4 │ │ Worker 5 │  
│ Port: 10000 │ │ Port: 10001 │ │ Port: 10002 │ │ Port: 10003 │ │ Port: 10004 │  
│BLOCKED: │ │BLOCKED: │ │BLOCKED: │ │BLOCKED: │ │BLOCKED: │  
│ray.get() │ │ray.get() │ │ray.get() │ │ray.get() │ │ray.get() │  
└─────────────┘ └─────────────┘ └─────────────┘ └─────────────┘ └─────────────┘  
New Task Request → ❌ FAIL: "No available ports"

### 7. Worker Pool Size vs Port Range Decision Tree

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships  
  
📁 Text-based diagram (backup)

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

┌─ START: Configure Ray Worker Ports ─┐  
│ │  
▼ │  
┌─ What's your workload? ─┐ │  
│ │ │  
▼ ▼ │  
┌─ Many Actors ─┐ ┌─ Many Tasks ─┐ │  
│ (Long-lived) │ │ (Short-lived) │ │  
└───────┬───────┘ └───────┬───────┘ │  
│ │ │  
▼ ▼ │  
┌─ Port Need = ─┐ ┌─ Port Need = ─┐ │  
│ Actor Count │ │ Peak Concurrent│ │  
│ Example: 500 │ │ Tasks: 200 │ │  
└───────┬───────┘ └───────┬───────┘ │  
│ │ │  
└────── Combine ──────────┘ │  
│ │  
▼ │  
┌─ Total Port Need ─┐ │  
│ = 500 + 200 = 700 │ │  
└─────────┬─────────┘ │  
│ │  
▼ │  
┌─ Configure num\_workers\_soft\_limit = 700 ─┐ │  
│ Configure port range = 10000-10700 │ │  
└─────────────────┬─────────────────────────┘ │  
│ │  
▼ │  
┌─ RESULT: 700 concurrent workers ─┐ │  
│ Each with dedicated port │ │  
└──────────────────────────────────┘ │

┌─ START: Configure Ray Worker Ports ─┐  
│ │  
▼ │  
┌─ What's your workload? ─┐ │  
│ │ │  
▼ ▼ │  
┌─ Many Actors ─┐ ┌─ Many Tasks ─┐ │  
│ (Long-lived) │ │ (Short-lived) │ │  
└───────┬───────┘ └───────┬───────┘ │  
│ │ │  
▼ ▼ │  
┌─ Port Need = ─┐ ┌─ Port Need = ─┐ │  
│ Actor Count │ │ Peak Concurrent│ │  
│ Example: 500 │ │ Tasks: 200 │ │  
└───────┬───────┘ └───────┬───────┘ │  
│ │ │  
└────── Combine ──────────┘ │  
│ │  
▼ │  
┌─ Total Port Need ─┐ │  
│ = 500 + 200 = 700 │ │  
└─────────┬─────────┘ │  
│ │  
▼ │  
┌─ Configure num\_workers\_soft\_limit = 700 ─┐ │  
│ Configure port range = 10000-10700 │ │  
└─────────────────┬─────────────────────────┘ │  
│ │  
▼ │  
┌─ RESULT: 700 concurrent workers ─┐ │  
│ Each with dedicated port │ │  
└──────────────────────────────────┘ │

### 8. Complete Ray Cluster Port Architecture

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships  
  
📁 Text-based diagram (backup)

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

RAY CLUSTER PORT LAYOUT:  
HEAD NODE:  
┌─────────────────────────────────────────────────────────────┐  
│ HEAD NODE │  
├─────────────────────────────────────────────────────────────┤  
│ GCS Server: Port 6379 │  
│ Dashboard: Port 8265 │  
│ Ray Client Server: Port 10001 │  
│ Node Manager: Port 10002 │  
│ Object Manager: Port 10003 │  
│ Metrics Agent: Port 10004 │  
│ Runtime Env Agent: Port 10005 │  
│ │  
│ Worker Pool: Ports 20000-20100 (100 ports) │  
│ ├─ Actor 1: Port 20000 │  
│ ├─ Actor 2: Port 20001 │  
│ ├─ Task Worker 1: Port 20002 │  
│ └─ Task Worker 2: Port 20003 │  
└─────────────────────────────────────────────────────────────┘  
WORKER NODE 1:  
┌─────────────────────────────────────────────────────────────┐  
│ WORKER NODE 1 │  
├─────────────────────────────────────────────────────────────┤  
│ Node Manager: Port 10002 │  
│ Object Manager: Port 10003 │  
│ Metrics Agent: Port 10004 │  
│ Runtime Env Agent: Port 10005 │  
│ │  
│ Worker Pool: Ports 20000-20100 (100 ports) │  
│ ├─ Actor 3: Port 20000 │  
│ ├─ Actor 4: Port 20001 │  
│ ├─ Task Worker 3: Port 20002 │  
│ └─ Task Worker 4: Port 20003 │  
└─────────────────────────────────────────────────────────────┘  
COMMUNICATION FLOWS:  
Driver ──(6379)──→ GCS ──(10002)──→ Node Manager ──(20000+)──→ Workers  
↑ ↓  
└──── Cluster State ───┘  
Worker ──(20000+)──→ Worker (Same Node: Direct)  
Worker ──(10003)───→ Object Manager ──(Network)──→ Object Manager ──(20000+)──→ Worker

RAY CLUSTER PORT LAYOUT:  
HEAD NODE:  
┌─────────────────────────────────────────────────────────────┐  
│ HEAD NODE │  
├─────────────────────────────────────────────────────────────┤  
│ GCS Server: Port 6379 │  
│ Dashboard: Port 8265 │  
│ Ray Client Server: Port 10001 │  
│ Node Manager: Port 10002 │  
│ Object Manager: Port 10003 │  
│ Metrics Agent: Port 10004 │  
│ Runtime Env Agent: Port 10005 │  
│ │  
│ Worker Pool: Ports 20000-20100 (100 ports) │  
│ ├─ Actor 1: Port 20000 │  
│ ├─ Actor 2: Port 20001 │  
│ ├─ Task Worker 1: Port 20002 │  
│ └─ Task Worker 2: Port 20003 │  
└─────────────────────────────────────────────────────────────┘  
WORKER NODE 1:  
┌─────────────────────────────────────────────────────────────┐  
│ WORKER NODE 1 │  
├─────────────────────────────────────────────────────────────┤  
│ Node Manager: Port 10002 │  
│ Object Manager: Port 10003 │  
│ Metrics Agent: Port 10004 │  
│ Runtime Env Agent: Port 10005 │  
│ │  
│ Worker Pool: Ports 20000-20100 (100 ports) │  
│ ├─ Actor 3: Port 20000 │  
│ ├─ Actor 4: Port 20001 │  
│ ├─ Task Worker 3: Port 20002 │  
│ └─ Task Worker 4: Port 20003 │  
└─────────────────────────────────────────────────────────────┘  
COMMUNICATION FLOWS:  
Driver ──(6379)──→ GCS ──(10002)──→ Node Manager ──(20000+)──→ Workers  
↑ ↓  
└──── Cluster State ───┘  
Worker ──(20000+)──→ Worker (Same Node: Direct)  
Worker ──(10003)───→ Object Manager ──(Network)──→ Object Manager ──(20000+)──→ Worker

### 9. Visual Summary: Port Types and Usage Patterns

🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships  
  
📁 Text-based diagram (backup)

[DIAGRAM: 🏗️ ARCHITECTURE DIAGRAM: Component Flow and Relationships]

┌─────────────────────────────────────────────────────────────────────────────┐  
│ RAY PORT CATEGORIES │  
├─────────────────────────────────────────────────────────────────────────────┤  
│ │  
│ 1. INFRASTRUCTURE PORTS (Fixed, 1 per node) │  
│ ┌─────────────────────────────────────────────────────────────────┐ │  
│ │ • GCS Port (6379) • Node Manager (10002) │ │  
│ │ • Dashboard (8265) • Object Manager (10003) │ │  
│ │ • Metrics Agent (10004) • Runtime Env Agent (10005) │ │  
│ └─────────────────────────────────────────────────────────────────┘ │  
│ │  
│ 2. WORKER PORTS (Dynamic, from shared pool) │  
│ ┌─────────────────────────────────────────────────────────────────┐ │  
│ │ SHARED PORT POOL │ │  
│ │ [20000, 20001, 20002, 20003, ..., 20100] │ │  
│ │ │ │ │  
│ │ ┌─────────────┴─────────────┐ │ │  
│ │ ▼ ▼ │ │  
│ │ ┌─ ACTORS ─┐ ┌─ TASKS ─┐ │ │  
│ │ │ Port: 1:1 │ │Port: N:1 │ │ │  
│ │ │ Lifetime: │ │Lifetime: │ │ │  
│ │ │ Long │ │ Short │ │ │  
│ │ └───────────┘ └──────────┘ │ │  
│ └─────────────────────────────────────────────────────────────────┘ │  
│ │  
│ 3. PORT LIMITS │  
│ ┌─────────────────────────────────────────────────────────────────┐ │  
│ │ Max Concurrent Ports = min( │ │  
│ │ Port Range Size, // e.g., 100 │ │  
│ │ num\_workers\_soft\_limit, // e.g., 50 │ │  
│ │ System FD Limit // e.g., 1024 │ │  
│ │ ) │ │  
│ │ Result: 50 concurrent workers maximum │ │  
│ └─────────────────────────────────────────────────────────────────┘ │  
└─────────────────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────────────────┐  
│ RAY PORT CATEGORIES │  
├─────────────────────────────────────────────────────────────────────────────┤  
│ │  
│ 1. INFRASTRUCTURE PORTS (Fixed, 1 per node) │  
│ ┌─────────────────────────────────────────────────────────────────┐ │  
│ │ • GCS Port (6379) • Node Manager (10002) │ │  
│ │ • Dashboard (8265) • Object Manager (10003) │ │  
│ │ • Metrics Agent (10004) • Runtime Env Agent (10005) │ │  
│ └─────────────────────────────────────────────────────────────────┘ │  
│ │  
│ 2. WORKER PORTS (Dynamic, from shared pool) │  
│ ┌─────────────────────────────────────────────────────────────────┐ │  
│ │ SHARED PORT POOL │ │  
│ │ [20000, 20001, 20002, 20003, ..., 20100] │ │  
│ │ │ │ │  
│ │ ┌─────────────┴─────────────┐ │ │  
│ │ ▼ ▼ │ │  
│ │ ┌─ ACTORS ─┐ ┌─ TASKS ─┐ │ │  
│ │ │ Port: 1:1 │ │Port: N:1 │ │ │  
│ │ │ Lifetime: │ │Lifetime: │ │ │  
│ │ │ Long │ │ Short │ │ │  
│ │ └───────────┘ └──────────┘ │ │  
│ └─────────────────────────────────────────────────────────────────┘ │  
│ │  
│ 3. PORT LIMITS │  
│ ┌─────────────────────────────────────────────────────────────────┐ │  
│ │ Max Concurrent Ports = min( │ │  
│ │ Port Range Size, // e.g., 100 │ │  
│ │ num\_workers\_soft\_limit, // e.g., 50 │ │  
│ │ System FD Limit // e.g., 1024 │ │  
│ │ ) │ │  
│ │ Result: 50 concurrent workers maximum │ │  
│ └─────────────────────────────────────────────────────────────────┘ │  
└─────────────────────────────────────────────────────────────────────────────┘

These Mermaid diagrams provide a modern, professional visualization of Ray's port allocation system while maintaining backward compatibility with the text-based versions. The diagrams will render beautifully in GitHub, GitLab, and most modern documentation platforms, while the collapsed text versions ensure the documentation works everywhere.

# About This Guide

This comprehensive guide represents the complete technical documentation of Ray's internal architecture. All 14 chapters have been properly formatted and cleaned of artifacts.

## Document Statistics

Chapters Processed: 14

Format: Clean HTML with proper styling

Generated: June 06, 2025 at 02:35 PM

Status: All formatting issues resolved