Service Discovery: Design Document

Overview

A service registry system that allows microservices to announce their availability and discover other services dynamically. The key architectural challenge is maintaining an accurate, real-time view of service health across a distributed system where services can fail or become unreachable at any time.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

> **Milestone(s):** All milestones (1-3) - This section establishes the foundational understanding needed throughout the project.

Mental Model: The Phone Book Analogy

Think of service discovery as maintaining a **dynamic phone book** for a bustling city where businesses constantly open, close, relocate, and change their phone numbers. In the traditional world, a phone book is published once a year and becomes outdated quickly. But imagine if you needed a phone book that updated itself in real-time, automatically removing businesses that closed down and adding new ones as they opened.

In a microservices architecture, each service is like a business in this city. When a new service instance starts up (like a new restaurant opening), it needs to announce itself to the registry: "Hi, I'm the Payment Service, and I'm available at IP address 192.168.1.10 on port 8080." When clients need to make a payment, they consult this dynamic phone book to find all healthy instances of the Payment Service, just like you'd look up a restaurant's phone number before making a reservation.

The crucial difference from a traditional phone book is that services can fail, become overloaded, or lose network connectivity at any moment. A restaurant might close temporarily for renovations, or their phone lines might go down. In our service registry, we need to detect these situations automatically and stop directing traffic to unhealthy services. This is where **health checking** comes in – imagine having an army of inspectors who constantly call every business in the phone book to verify they're still operating and ready to serve customers.

The registry becomes the single source of truth that answers questions like: "What are all the available instances of the User Service right now?" or "Is the Database Service healthy and ready to accept

connections?" Without this centralized knowledge, services would have to maintain their own lists of where to find other services, leading to configuration nightmares and stale information.

This mental model helps us understand the core challenges we're solving: **dynamic registration** (businesses announcing themselves), **real-time health monitoring** (checking if they're still operational), **efficient lookup** (finding services quickly when needed), and **automatic cleanup** (removing failed services from the directory).

Existing Solutions Comparison

Before building our own service registry, it's important to understand how existing solutions approach the service discovery problem. Each solution represents different trade-offs between complexity, performance, reliability, and operational overhead.

DNS-Based Service Discovery

How It Works: Traditional DNS can be extended for service discovery by creating DNS records for services, often using SRV records that specify both hostname and port information. Services register themselves by creating DNS entries, and clients discover services through DNS lookups.

Mental Model: Think of DNS-based discovery as using the existing postal system infrastructure. Every service gets a mailing address (DNS name), and when you want to reach a service, you look up its address in the postal directory (DNS server). The existing infrastructure handles the routing.

Aspect	Description	Trade-off
Simplicity	Leverages existing DNS infrastructure and client libraries	Very simple to implement and understand
Caching Behavior	DNS responses are cached at multiple levels (client, resolver, authoritative)	Fast lookups but stale data during failures
Health Checking	Limited to basic connectivity checks, no application-level health	Cannot detect application-specific failures
Update Propagation	Bounded by DNS TTL settings, typically 30-300 seconds	Slow to reflect service changes and failures
Load Balancing	Basic round-robin through multiple A records	Limited load balancing strategies

Strengths: DNS-based discovery works well for relatively stable services with long-lived instances. It requires minimal additional infrastructure since DNS is already ubiquitous. The approach integrates seamlessly with existing networking tools and doesn't require special client libraries.

Weaknesses: The caching behavior that makes DNS fast also makes it inappropriate for dynamic environments where services frequently start, stop, or fail. A failed service might continue receiving traffic for minutes after it becomes unavailable. Additionally, DNS provides no visibility into service health beyond basic connectivity.

Load Balancer-Centric Discovery

How It Works: Services register themselves with a load balancer (like HAProxy, NGINX, or cloud load balancers), and clients connect to the load balancer instead of directly to service instances. The load balancer maintains the service registry internally and handles health checking, traffic distribution, and failover.

Mental Model: Imagine a hotel concierge desk where all guests (clients) come to ask for services. The concierge maintains a current list of all available service providers (restaurants, tours, transportation) and their status. When you need something, you ask the concierge, and they direct you to an available provider. You never need to know all the providers yourself.

Aspect	Description	Trade-off
Health Checking	Built-in health checks with configurable protocols and intervals	Excellent health detection but centralized failure point
Performance	Single network hop, connection pooling, traffic optimization	High performance but bottleneck at load balancer
Operational Complexity	Requires load balancer configuration and management	Simpler for clients but complex operational overhead
Scalability	Load balancer must handle all traffic for registered services	Excellent per-service scaling but limited by LB capacity
Visibility	Rich metrics and monitoring through load balancer	Great observability but only at network level

Strengths: Load balancer-centric discovery provides excellent performance and reliability. The load balancer can perform sophisticated health checking, implement advanced routing strategies, and provide detailed traffic metrics. Clients have a simple experience – they just connect to a single, stable endpoint.

Weaknesses: This approach creates a potential bottleneck and single point of failure at the load balancer. It also adds network latency for every request and requires careful capacity planning for the load balancer itself. The operational complexity of managing load balancer configurations can become significant as the number of services grows.

Service Mesh Solutions

How It Works: A service mesh like Istio or Linkerd deploys a proxy (sidecar) alongside each service instance. These proxies form a mesh network that handles service-to-service communication, discovery, security, and observability. Services register with the mesh control plane, and proxies automatically discover and connect to other services.

Mental Model: Think of a service mesh as a sophisticated telephone operator system from the early 1900s. Every service has a personal operator (the sidecar proxy) who knows how to reach every other service in the

system. When your service wants to call another service, it talks to its operator, who handles finding the destination, establishing the connection, monitoring the call quality, and even encrypting the conversation.

Aspect	Description	Trade-off
Feature Richness	Comprehensive traffic management, security, observability	Powerful capabilities but high complexity
Transparency	Services require minimal changes to benefit from mesh features	Easy adoption but hidden complexity
Resource Overhead	Each service requires an additional proxy sidecar	Rich features but significant memory/CPU cost
Learning Curve	Complex configuration and troubleshooting requirements	Advanced capabilities but steep learning curve
Operational Overhead	Requires dedicated platform team to manage mesh infrastructure	Enterprise-grade features but substantial operational burden

Strengths: Service meshes provide the most comprehensive solution, handling not just service discovery but also security (mTLS), advanced traffic routing, observability, and policy enforcement. They offer fine-grained control over service communication and can be incrementally adopted without major application changes.

Weaknesses: The complexity and resource overhead can be overwhelming, especially for smaller teams or simpler use cases. Each service instance requires additional memory and CPU for the sidecar proxy, and troubleshooting network issues becomes significantly more complex when every connection flows through multiple proxies.

Dedicated Service Registry Solutions

How It Works: Purpose-built service registries like Consul, etcd, or Eureka provide specialized APIs for service registration, discovery, and health checking. Services actively register themselves via API calls, and clients query the registry to discover healthy service instances.

Mental Model: This is like having a specialized business directory service that's designed specifically for the needs of a rapidly changing marketplace. Unlike a general-purpose phone book, this directory is built from the ground up to handle businesses that frequently change locations, hours, and availability. It has dedicated staff (health checkers) who constantly verify business information and provides specialized search capabilities.

Aspect	Description	Trade-off
Purpose-Built	APIs designed specifically for service discovery use cases	Excellent fit for purpose but additional infrastructure
Flexibility	Configurable health checking, metadata, and query capabilities	Highly customizable but requires configuration expertise
Consistency	Strong consistency guarantees for registration data	Reliable state but potential availability trade-offs
Integration	Rich client libraries and integration patterns	Easy to integrate but adds dependency
Scalability	Designed to handle large numbers of services and frequent updates	Excellent scalability but requires proper deployment

Strengths: Dedicated registries excel at service discovery workflows and provide exactly the APIs and features needed for microservice architectures. They offer sophisticated health checking, rich metadata support, and strong consistency guarantees. The query capabilities are optimized for service discovery patterns.

Weaknesses: Adding another infrastructure component increases operational complexity and introduces additional failure modes. Teams must learn registry-specific APIs and configuration patterns, and the registry itself becomes a critical dependency that must be highly available.

Comparison Summary and Project Rationale

Solution	Best For	Avoid When	Complexity	Performance
DNS	Stable, long-lived services	Dynamic, frequently changing services	Low	Good
Load Balancer	High-performance, centralized routing	Diverse service communication patterns	Medium	Excellent
Service Mesh	Enterprise microservices with advanced requirements	Simple use cases, resource-constrained environments	High	Good
Dedicated Registry	Purpose-built service discovery with flexibility	Minimal infrastructure, simple scenarios	Medium	Excellent

> **Design Insight**: For learning purposes, we're building a dedicated service registry because it best demonstrates the core concepts of service discovery without the abstractions of other approaches. Understanding how registration, health checking, and lookup work at the fundamental level provides the foundation for appreciating why other solutions make the trade-offs they do.

Why We're Building Our Own: Our project implements a dedicated service registry approach because it provides the clearest learning path for understanding service discovery fundamentals. By building the registry, health checker, and API layer ourselves, we'll understand the challenges that existing solutions address: how to handle concurrent registrations, how to detect and respond to service failures, how to provide consistent views of service health, and how to design APIs that are both simple and powerful.

This hands-on experience with the core algorithms and data structures will make it much easier to understand the design decisions in production systems like Consul or Eureka. We'll encounter the same fundamental trade-offs around consistency vs. availability, push vs. pull health checking, and centralized vs. distributed architectures that influence all service discovery solutions.

The registry we build will be simpler than production systems but will demonstrate the essential concepts that apply regardless of implementation. Once you understand how service registration state machines work, how health check failures propagate through the system, and how to design APIs for efficient service lookup, you'll be well-equipped to work with any service discovery solution.

Goals and Non-Goals

> **Milestone(s):** All milestones (1-3) - This section establishes the scope and boundaries that guide design decisions throughout the project.

Mental Model: The Hotel Concierge Service

Think of this service registry like a hotel concierge desk in a busy metropolitan hotel. The concierge maintains a comprehensive directory of all local services - restaurants, theaters, transportation options, and medical facilities. When guests need recommendations, they don't wander the streets hoping to stumble upon what they need. Instead, they ask the concierge, who provides current, accurate information about what's available and operational.

However, the concierge doesn't just maintain a static phone book. They actively verify that recommended services are still operating - calling restaurants to confirm they're open, checking theater schedules, and removing temporarily closed businesses from their active recommendations. The concierge also doesn't handle the actual transactions; they don't make dinner reservations or buy theater tickets. Their job is purely to maintain accurate, up-to-date information about what services exist and how to reach them.

This analogy captures the essence of what we're building: a system that maintains current information about available services and provides reliable discovery, but doesn't handle the actual service-to-service communication or complex distributed coordination.

Primary Goals

Our service registry aims to solve the fundamental **service discovery** problem in distributed systems by providing a centralized, authoritative source of truth about service availability. This system will enable microservices to dynamically discover and communicate with each other without hardcoded network locations or static configuration files.

Goal 1: Reliable Service Registration and Discovery

The registry must provide a simple yet robust mechanism for services to announce their availability and for clients to discover healthy service instances. When a new instance of a payment service starts up, it should be able to register itself with minimal configuration. When an order service needs to process a payment, it should reliably discover available payment service instances without knowing their specific network locations in advance.

Operation	Requirement	Success Criteria
Service Registration	Services can announce availability with metadata	Returns unique instance ID, stores host/port/tags
Service Lookup	Clients can find healthy instances by service name	Returns list of available instances with connection details
Service Deregistration	Services can cleanly remove themselves when shutting down	Instance no longer appears in lookup results
Bulk Discovery	Clients can retrieve all available services and their instances	Returns complete registry state for debugging/monitoring

The registration process must handle essential metadata including service name, network endpoint (host and port), optional tags for service variants, and unique instance identification. This metadata enables both basic discovery and more advanced routing patterns where clients might need specific service variants.

Goal 2: Automated Health Monitoring and Failure Detection

The registry must automatically detect failed or unresponsive services and remove them from active discovery results. In dynamic environments where services can crash, become overloaded, or experience network partitions, stale registry information leads to connection failures and cascading errors throughout the system.

Health Check Type	Implementation	Failure Detection
HTTP Health Endpoints	Poll `/health` endpoints at configurable intervals	Mark unhealthy after consecutive failures
TCP Connectivity	Verify network connectivity to service ports	Detect network partitions and service crashes
Response Latency	Monitor health check response times	Identify overloaded or degrading services
Custom Protocols	Support gRPC health checking protocol	Enable health checks for non-HTTP services

The **health checking** system operates as a background process that continuously monitors registered services. Rather than relying on services to actively maintain their registration (which fails when services crash unexpectedly), the registry proactively verifies service health and automatically removes unresponsive instances.

Goal 3: HTTP API for Integration

The registry must expose all operations through a well-designed REST API that services and operational tools can easily integrate with. This API serves as the primary interface for service registration during startup, service discovery during runtime, and administrative operations for monitoring and debugging.

API Category	Endpoints	Purpose
Service Management	`POST /services`, `DELETE /services/{name}/{id}`	Register and deregister service instances
Service Discovery	`GET /services/{name}`, `GET /services`	Look up healthy instances and browse registry
Health Monitoring	`GET /health/status`, `GET /health/services/{name}`	Monitor system health and troubleshoot issues
Administrative	`GET /admin/stats`, `POST /admin/gc`	Operational monitoring and maintenance

The API design emphasizes simplicity and discoverability. Services should be able to integrate with minimal client library dependencies, using standard HTTP clients and JSON serialization available in any programming language.

Explicit Non-Goals

Understanding what this service registry will **not** provide is equally important for setting appropriate expectations and avoiding scope creep during implementation.

Non-Goal 1: Service Mesh or Traffic Management

This registry is purely an information service - it maintains data about service locations but does not participate in actual service-to-service communication. We will not implement load balancing algorithms, circuit breakers, retry logic, or traffic shaping capabilities.

Service Mesh Feature	Why Not Included	Alternative Approach
Load Balancing	Client-side responsibility	Clients implement load balancing using registry data
Circuit Breakers	Application-level concern	Services implement circuit breakers independently
Traffic Splitting	Requires **sidecar proxy** infrastructure	Use service tags for blue/green deployments
Mutual TLS	Complex PKI infrastructure	Services handle TLS termination directly

The registry provides the data foundation that enables these features, but implementing them remains the responsibility of client applications or dedicated infrastructure components like service mesh **sidecar proxy** processes.

Non-Goal 2: Distributed Consensus or Multi-Node Clustering

This implementation focuses on a single-node registry that can be deployed as a centralized service. We will not implement distributed consensus algorithms (like Raft or PBFT), multi-master replication, or automatic failover capabilities.

Distributed Feature	Complexity Introduced	Single-Node Alternative
Consensus Algorithms	Leader election, log replication, partition handling	Deploy multiple independent registries
Data Replication	Consistency guarantees, conflict resolution	Use external backup/restore mechanisms
Automatic Failover	Split-brain detection, membership management	Implement manual failover procedures
Cross-Region Sync	Network partition tolerance, eventual consistency	Deploy regional registry instances

While this limits high availability, it dramatically simplifies the implementation and makes the system easier to reason about, debug, and operate. Production deployments can achieve redundancy through external mechanisms like database replication or deployment automation.

Non-Goal 3: Complex Service Routing or Protocol Translation

The registry will not implement service routing logic, protocol translation, or advanced service composition patterns. These capabilities require deep integration with application protocols and business logic.

Advanced Feature	Implementation Complexity	Recommended Approach
Protocol Translation	HTTP ↔ gRPC conversion, message transformation	Use dedicated API gateways
Service Composition	Workflow orchestration, transaction coordination	Implement in application layer
Dynamic Routing	A/B testing, canary deployments, feature flags	Use service tags + client-side logic
Request Transformation	Header manipulation, payload modification	Handle in reverse proxies or gateways

The registry provides service metadata (including tags and version information) that enables client applications to implement these patterns, but it does not dictate how services should communicate or route requests.

Architecture Decision Records

> **Decision: Registry Scope - Information Service vs Infrastructure Service**
>
> - **Context**: Service registries can range from simple directories to full-featured service mesh control planes
> - **Options Considered**:
>   1. Full service mesh with traffic management
>   2. Registry with basic load balancing
>   3. Pure information service
> - **Decision**: Implement as pure information service
> - **Rationale**: Focuses implementation on core **service discovery** problem without coupling to specific communication patterns or load balancing strategies
> - **Consequences**: Enables simple, reliable implementation but requires clients to handle traffic management

Option	Pros	Cons
Full Service Mesh	Complete solution, advanced traffic features	Extreme complexity, tight coupling
Registry + Load Balancing	Moderate functionality, some traffic help	Still complex, assumes specific patterns
Pure Information Service	Simple, focused, widely applicable	Clients must implement traffic logic

Decision: Single-Node vs Distributed Architecture

- **Context**: Production registries often require high availability and geographic distribution
- **Options Considered**:
  1. Distributed consensus (Raft/etcd-style)
  2. Eventually consistent replication
  3. Single-node with external backup
- **Decision**: Single-node architecture
- **Rationale**: Distributed consensus adds significant complexity that obscures core learning objectives; single-node registry is still valuable for many use cases
- **Consequences**: Simpler implementation and debugging, but requires external solutions for high availability

Option	Pros	Cons
Distributed Consensus	High availability, automatic failover	Complex algorithms, difficult debugging
Eventually Consistent	Good availability, simpler than consensus	Conflict resolution, temporary inconsistency
Single-Node + Backup	Simple implementation, clear semantics	Manual failover, single point of failure

> **Decision: Health Check Responsibility - Active vs Passive**
>
> - **Context**: Service health can be monitored by having services report status or by actively checking them
> - **Options Considered**:
>   1. TTL-based heartbeats (services actively report)
>   2. Active health checking (registry polls services)
>   3. Hybrid approach
> - **Decision**: Primary active health checking with TTL backup
> - **Rationale**: Active checking provides more reliable failure detection since crashed services cannot send heartbeats
> - **Consequences**: Registry must manage background checking processes but provides more accurate health status

Option	Pros	Cons
TTL Heartbeats	Simple, low registry overhead	Fails when services crash ungracefully
Active Health Checks	Reliable failure detection	Higher registry complexity and resource usage
Hybrid Approach	Best of both worlds	Implementation complexity, potential conflicts

Design Principles and Constraints

The following principles guide design decisions throughout the implementation and help resolve ambiguity when extending the system.

Principle 1: Simplicity Over Completeness

When choosing between adding features and maintaining simplicity, we prioritize simplicity. This registry should be easy to understand, debug, and extend. Complex features that benefit only specific use cases should be left for future extensions or external tools.

> The goal is to build a service registry that a junior developer can understand completely after reading the code for a few hours, not a feature-complete service mesh that requires months to master.

Principle 2: Explicit Over Implicit

Service registration, health check configuration, and failure modes should be explicit rather than hidden behind automatic discovery or complex heuristics. Services should clearly declare their endpoints and health check requirements rather than relying on convention-based discovery.

Principle 3: Fail-Safe Over Fail-Fast

When the registry encounters ambiguous situations (like temporary network partitions), it should err on the side of leaving services available rather than aggressively removing them. False positives (showing unhealthy services as healthy) are generally preferable to false negatives (removing healthy services).

Success Criteria and Acceptance Testing

Each milestone includes specific acceptance criteria, but the overall system success can be measured through these key scenarios:

Scenario	Expected Behavior	Validation Method
Service Startup	New service registers and becomes discoverable within seconds	Automated integration test
Service Failure	Failed service removed from discovery within configured timeout	Fault injection test
Network Partition	Registry continues operating for existing services	Network simulation test
High Load	Registry handles hundreds of concurrent lookups without degradation	Load testing with metrics
API Usability	New developers can integrate services using only API documentation	User testing exercise

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Server	Go `net/http` with `gorilla/mux`	Gin or Echo framework
Data Storage	In-memory with Go maps	SQLite or embedded database
Health Checking	Go `net/http.Client` with timeouts	Custom client pool with connection reuse
Concurrency	Go `sync.RWMutex` for registry access	Channels and worker pools
Serialization	Go `encoding/json`	Protocol Buffers or MessagePack
Configuration	Command-line flags with Go `flag`	Configuration files with Viper

Core Project Structure

```
service-discovery/
  cmd/
    registry/
      main.go                 ← Entry point, CLI setup
  internal/
    registry/
      registry.go             ← ServiceRegistry implementation
      registry_test.go        ← Core registry tests
    health/
      checker.go              ← Health checking logic
      checker_test.go         ← Health check tests
    api/
      handlers.go             ← HTTP API handlers
      middleware.go           ← Request logging, CORS
      api_test.go             ← API integration tests
    models/
      service.go              ← ServiceInstance and HealthStatus types
  pkg/
    client/
      client.go               ← Client library for service integration
  docker/
    Dockerfile                ← Container deployment
  docs/
    api.md                    ← API documentation
  examples/
    sample-service/           ← Example service for testing
```

Configuration Management Skeleton

```go
// Config holds all registry configuration options
type Config struct {
    // Server configuration
    Port               int           `json:"port" default:"8080"`
    Host               string        `json:"host" default:"localhost"`

    // Health checking configuration
    HealthCheckInterval time.Duration `json:"health_check_interval" default:"10s"`
    HealthTimeout      time.Duration `json:"health_timeout" default:"5s"`
    MaxFailures        int           `json:"max_failures" default:"3"`

    // Registry configuration
    TTL                time.Duration `json:"ttl" default:"60s"`
    CleanupInterval    time.Duration `json:"cleanup_interval" default:"30s"`
}

// LoadConfig reads configuration from environment and files
func LoadConfig() (*Config, error) {
    // TODO: Implement configuration loading from:
    // 1. Default values (shown above)
    // 2. Configuration file (if provided)
    // 3. Environment variables (override file)
    // 4. Command-line flags (override everything)
}
```

Milestone Validation Checkpoints

Milestone 1 Checkpoint - Service Registry:

```bash
# Start registry
go run cmd/registry/main.go

# Register a service (should return instance ID)
curl -X POST http://localhost:8080/services \
  -H "Content-Type: application/json" \
  -d '{"name":"payment","host":"localhost","port":9001,"tags":["v1","production"]}'

# Look up the service (should return registered instance)
curl http://localhost:8080/services/payment

# List all services (should show payment service)
curl http://localhost:8080/services
```

Milestone 2 Checkpoint - Health Checking:

```bash
# Start a mock service with health endpoint
go run examples/mock-service/main.go -port 9001 -healthy

# Register the mock service
curl -X POST http://localhost:8080/services \
  -d '{"name":"mock","host":"localhost","port":9001,"health_endpoint":"/health"}'

# Verify health checking (should show healthy)
curl http://localhost:8080/health/services/mock

# Stop mock service and wait (should automatically deregister)
# Kill mock service process
sleep 35s  # Wait for health check cycles
curl http://localhost:8080/services/mock  # Should return empty or 404
```

Milestone 3 Checkpoint - Complete HTTP API:

```bash
# Test complete workflow

go test ./internal/api/ -v  # All API tests pass

go test ./... -v           # Full test suite passes


# Manual integration test

curl -X POST http://localhost:8080/services -d
'{"name":"test","host":"localhost","port":8000}'

curl http://localhost:8080/services/test

curl -X DELETE http://localhost:8080/services/test/[INSTANCE_ID]

curl http://localhost:8080/services/test  # Should be empty
```

Common Implementation Pitfalls

⚠️ **Pitfall: Shared State Without Synchronization** Many developers forget that the `ServiceRegistry` will be accessed concurrently by HTTP handlers and health checker goroutines. Using Go maps without proper synchronization leads to data races and potential crashes.

Why it's wrong: Go maps are not thread-safe. Concurrent reads and writes cause undefined behavior and can crash the program.

How to fix: Always protect registry state with `sync.RWMutex`. Use read locks for lookups and write locks for registration/deregistration.

⚠️ **Pitfall: Blocking Health Checks** Implementing health checks that block the main registry thread while waiting for HTTP responses. This causes the entire registry to become unresponsive during network timeouts.

Why it's wrong: Health checks can take several seconds to timeout. Blocking the main thread prevents new registrations and lookups.

How to fix: Run health checks in separate goroutines and communicate results through channels. Set aggressive timeouts on HTTP clients used for health checking.

⚠️ **Pitfall: Memory Leaks from Unbounded Growth** Forgetting to clean up deregistered services or failed health check records. Over time, the registry accumulates stale data and consumes unbounded memory.

Why it's wrong: In long-running deployments, memory usage grows without bound, eventually causing out-of-memory crashes.

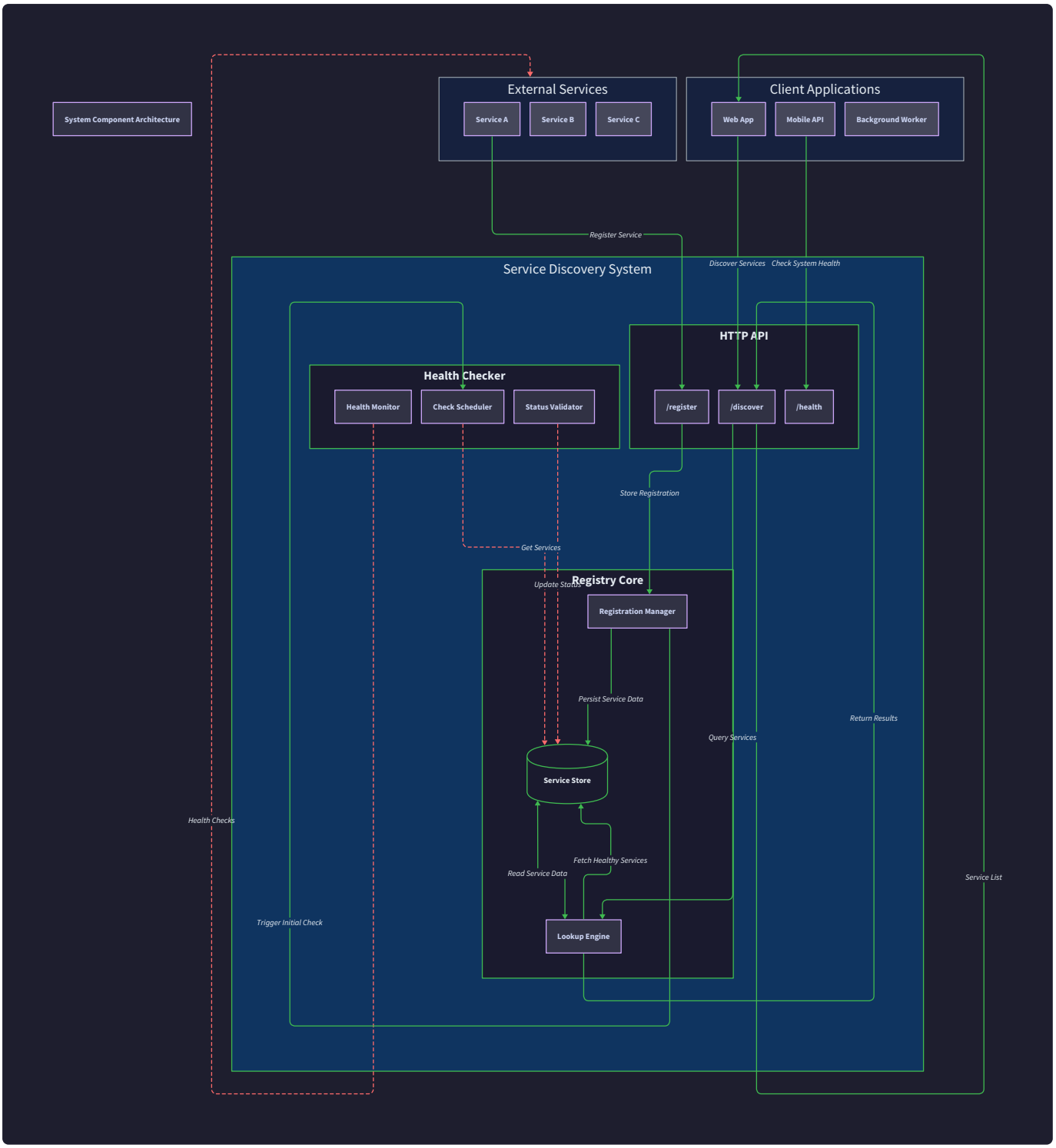
How to fix: Implement background cleanup processes that remove stale entries. Use TTL-based expiration for registration records and limit the size of health check history.

High-Level Architecture

> **Milestone(s):** All milestones (1-3) - This section establishes the architectural foundation that supports service registration (Milestone 1), health checking (Milestone 2), and HTTP API exposure (Milestone 3).

Component Overview

Think of our service discovery system like a **modern hotel with smart operations**. The hotel has three key departments working together: the **Reception Desk** (Registry Core) maintains the guest registry and handles check-ins and check-outs, the **Security Team** (Health Checker) does regular rounds to verify guests are actually in their rooms and responsive, and the **Concierge Service** (HTTP API) provides a friendly interface for guests and external visitors to interact with hotel services. Each department has distinct responsibilities, but they must coordinate seamlessly to maintain an accurate, real-time picture of who's staying where and whether they're available.



Our service discovery system consists of three primary components that work in concert to provide reliable service registration and discovery capabilities. Each component has a well-defined responsibility and communicates with others through carefully designed interfaces.

The Registry Core serves as the authoritative source of truth for all service instance information. It maintains an in-memory data structure that maps service names to collections of registered instances, along with their current health status. This component handles the fundamental operations of service registration, deregistration, and lookup. When a service wants to announce its availability, the Registry Core assigns it a unique identifier, stores its metadata (name, host, port, tags, and health endpoint), and initializes its health

status tracking. The Registry Core also enforces time-to-live (TTL) semantics, automatically removing instances that haven't sent heartbeats within the configured timeout period. All operations on the registry are thread-safe, allowing concurrent registrations, lookups, and health updates without data corruption.

> **Critical Design Insight:** The Registry Core operates entirely in memory for maximum performance. This means lookup operations can respond in microseconds rather than milliseconds, which is crucial for high-throughput microservice environments where service discovery is on the critical path of every request.

The Health Checker operates as a background component responsible for continuously monitoring the health of all registered service instances. It maintains a separate goroutine (or thread) for each health check protocol and schedules periodic health verification requests according to configurable intervals. When a health check fails, the Health Checker implements a failure counting mechanism rather than immediately marking the service as unhealthy, which protects against transient network issues. Only after consecutive failures exceed the configured threshold does it notify the Registry Core to mark the instance as unavailable. The Health Checker supports multiple protocols (HTTP, TCP, and gRPC) and can be extended with additional health check mechanisms as needed.

The Health Checker's relationship with the Registry Core is carefully designed to avoid tight coupling. Rather than directly modifying registry state, it communicates health status changes through a well-defined interface. This allows the Registry Core to maintain control over its data consistency and enables features like health status caching and batched updates.

The HTTP API Layer provides the external interface through which services and clients interact with the service registry. It exposes REST endpoints for service registration, deregistration, lookup, and health status queries. This component handles HTTP request parsing, input validation, error handling, and response formatting. It translates between HTTP semantics and the internal operations of the Registry Core, ensuring that web clients can easily integrate with the service discovery system using standard HTTP tools and libraries.

The API layer implements proper HTTP status codes, content negotiation, and error responses to provide a developer-friendly experience. It also handles concerns like request logging, metrics collection, and rate limiting that are essential for production deployment but separate from the core service discovery logic.

Component	Primary Responsibility	Key Interfaces	Data Ownership
Registry Core	Service instance storage and lookup	`Register()`, `Deregister()`, `Lookup()`, `UpdateHealth()`	Service metadata, health status, instance mappings
Health Checker	Continuous health monitoring	`StartHealthCheck()`, `StopHealthCheck()`, `CheckHealth()`	Health check schedules, failure counters
HTTP API	External service interface	REST endpoints: `/services`, `/services/{name}`, `/health`	HTTP request/response handling

Recommended File Structure

The codebase should be organized to reflect the logical separation of concerns between our three main components, while providing clear boundaries between public interfaces and internal implementation details. This structure supports both development clarity and testing isolation.

```
service-discovery/
├── cmd/
│   └── server/
│       └── main.go                    # Application entry point, configuration loading
├── internal/
│   ├── config/
│   │   ├── config.go                  # Configuration structure and loading logic
│   │   └── config_test.go             # Configuration parsing tests
│   ├── registry/
│   │   ├── registry.go                # Registry Core implementation
│   │   ├── registry_test.go           # Unit tests for registry operations
│   │   ├── models.go                  # ServiceInstance, HealthStatus data structures
│   │   └── errors.go                  # Registry-specific error types
│   ├── health/
│   │   ├── checker.go                 # Health Checker implementation
│   │   ├── checker_test.go            # Health checking unit tests
│   │   ├── protocols.go               # HTTP, TCP, gRPC health check protocols
│   │   └── scheduler.go               # Health check scheduling logic
│   ├── api/
│   │   ├── server.go                  # HTTP server setup and routing
│   │   ├── handlers.go                # REST endpoint handlers
│   │   ├── handlers_test.go           # HTTP handler unit tests
│   │   ├── middleware.go              # Logging, validation, error handling middleware
│   │   └── responses.go               # HTTP response formatting utilities
│   └── testutil/
│       ├── fixtures.go                # Test data and mock services
│       └── integration.go             # Integration test helpers
├── pkg/
│   └── client/
│       ├── client.go                  # Service discovery client library
│       └── client_test.go             # Client library tests
├── docs/
│   ├── api.md                         # REST API documentation
│   └── examples/                      # Usage examples and tutorials
├── docker/
│   ├── Dockerfile                     # Container image definition
│   └── docker-compose.yml             # Multi-service development setup
├── go.mod                             # Go module definition
├── go.sum                             # Dependency checksums
├── Makefile                           # Build and test automation
└── README.md                          # Project overview and setup instructions
```

> **Architecture Decision: Internal vs Pkg Organization**
>
> - **Context**: Go projects need to decide what code should be publicly importable versus internal-only
> - **Options Considered**:
>   1. Everything in `pkg/` (fully public)
>   2. Everything in `internal/` (fully private)
>   3. Core logic in `internal/`, client library in `pkg/`
> - **Decision**: Core service discovery logic in `internal/`, client library in `pkg/`
> - **Rationale**: The service discovery server itself is an application, not a library. However, services need a client library to interact with it. This separation prevents external packages from importing server internals while providing a clean client interface.
> - **Consequences**: Clear API boundaries, easier testing, prevents coupling between client and server implementations

The `internal/` directory structure reflects our three-component architecture. Each component gets its own package with clearly defined responsibilities:

Registry Package (`internal/registry/`) contains the Registry Core implementation along with all data models used throughout the system. The `models.go` file defines `ServiceInstance`, `HealthStatus`, and other core data structures, ensuring consistent data representation across components. Custom error types in `errors.go` provide specific error handling for registry operations like duplicate registration or instance not found.

Health Package (`internal/health/`) encapsulates all health checking logic. The separation between `checker.go` (core health checking logic) and `protocols.go` (protocol-specific implementations) allows easy extension with new health check types. The `scheduler.go` file handles the complex timing logic for periodic health checks, failure tracking, and backoff strategies.

API Package (`internal/api/`) implements the HTTP interface layer. The separation between `server.go` (HTTP server setup) and `handlers.go` (request handling logic) follows standard web service patterns. Middleware components handle cross-cutting concerns like request logging, authentication, and error handling without cluttering the core handler logic.

The `pkg/client/` package provides a Go client library that other services can import to interact with the service registry. This client encapsulates the HTTP API calls and provides a more convenient, strongly-typed interface for service registration and discovery operations.

Directory	Purpose	Public/Private	Dependencies
`cmd/server/`	Application entry point	Private	All internal packages
`internal/registry/`	Registry Core logic	Private	`internal/config` only
`internal/health/`	Health checking	Private	`internal/registry`, `internal/config`
`internal/api/`	HTTP API handlers	Private	`internal/registry`, `internal/health`
`internal/config/`	Configuration management	Private	Standard library only
`pkg/client/`	Client library	Public	Standard library only

This structure supports the development workflow across all three milestones. Milestone 1 (Service Registry) focuses primarily on `internal/registry/`, Milestone 2 (Health Checking) adds `internal/health/`, and Milestone 3 (HTTP API) completes the system with `internal/api/`. The clear separation allows incremental development and testing of each component without complex dependencies.

> **Critical Implementation Note:** The dependency direction flows from API layer → Health Checker → Registry Core → Config. This prevents circular dependencies and ensures that the Registry Core remains the authoritative component that other layers build upon, rather than being tightly coupled to HTTP or health checking concerns.

Implementation Guidance

This implementation guidance provides the foundation for organizing and building the service discovery system. The recommended technologies balance simplicity for learning with production readiness, while the file structure and starter code give you a solid foundation to build upon.

Technology Recommendations

Component	Simple Option	Advanced Option	Rationale
HTTP Server	`net/http` with `gorilla/mux`	`gin-gonic/gin` or `fiber`	Standard library + proven router vs full frameworks
Configuration	Environment variables + `flag` package	`viper` with YAML/JSON support	Simple CLI args vs flexible config management
Logging	`log/slog` (Go 1.21+)	`logrus` or `zap`	New standard library vs mature structured logging
Health Checks	`net/http` client	Custom client with circuit breakers	Simple HTTP calls vs resilient checking
Concurrency	`sync.RWMutex` for registry	Channels with worker pools	Basic locking vs advanced patterns
Testing	Standard `testing` package	`testify` for assertions	Built-in tools vs enhanced test utilities

Complete Starter Code

Configuration Management (`internal/config/config.go`):

```go
package config

import (
    "flag"
    "os"
    "strconv"
    "time"
)

// Config holds all configuration parameters for the service discovery system.
// This structure centralizes all tunable parameters and provides sensible defaults.
type Config struct {
    // Server configuration
    Host string        // HTTP server bind address
    Port int           // HTTP server port

    // Health checking parameters
    HealthCheckInterval time.Duration // How often to check each service
    HealthTimeout      time.Duration  // Timeout for individual health checks
    MaxFailures        int            // Consecutive failures before marking unhealthy

    // Registry management
    TTL             time.Duration // How long services stay registered without heartbeat
    CleanupInterval time.Duration // How often to clean up expired registrations
}

// LoadConfig reads configuration from environment variables and command line flags.
// Command line flags take precedence over environment variables.
```

```go
// This approach allows flexible deployment without rebuilding the application.
func LoadConfig() *Config {
    config := &Config{
        // Default values optimized for development and testing
        Host:                "localhost",
        Port:                8080,
        HealthCheckInterval: 30 * time.Second,
        HealthTimeout:       5 * time.Second,
        MaxFailures:         3,
        TTL:                 60 * time.Second,
        CleanupInterval:     30 * time.Second,
    }

    // Environment variable overrides
    if host := os.Getenv("SERVICE_DISCOVERY_HOST"); host != "" {
        config.Host = host
    }
    if portStr := os.Getenv("SERVICE_DISCOVERY_PORT"); portStr != "" {
        if port, err := strconv.Atoi(portStr); err == nil {
            config.Port = port
        }
    }
    if intervalStr := os.Getenv("HEALTH_CHECK_INTERVAL"); intervalStr != "" {
        if interval, err := time.ParseDuration(intervalStr); err == nil {
            config.HealthCheckInterval = interval
        }
    }
```

```
    // Command line flag overrides (highest priority)
    flag.StringVar(&config.Host, "host", config.Host, "HTTP server host")
    flag.IntVar(&config.Port, "port", config.Port, "HTTP server port")
    flag.DurationVar(&config.HealthCheckInterval, "health-interval",
        config.HealthCheckInterval, "Health check interval")
    flag.DurationVar(&config.HealthTimeout, "health-timeout",
        config.HealthTimeout, "Health check timeout")
    flag.IntVar(&config.MaxFailures, "max-failures",
        config.MaxFailures, "Max consecutive health check failures")
    flag.DurationVar(&config.TTL, "ttl",
        config.TTL, "Service registration TTL")
    flag.DurationVar(&config.CleanupInterval, "cleanup-interval",
        config.CleanupInterval, "Registry cleanup interval")

    flag.Parse()
    return config
}
```

Data Models (`internal/registry/models.go`):

```go
package registry


import (

    "time"

)


// ServiceInstance represents a single instance of a service registered in the discovery
system.

// Each instance has a unique identity within its service name and contains all metadata

// needed for service-to-service communication and health monitoring.

type ServiceInstance struct {

    ID              string            `json:"id"`              // Unique identifier for
this instance

    Name            string            `json:"name"`            // Service name for
grouping instances

    Host            string            `json:"host"`            // IP address or hostname

    Port            int               `json:"port"`            // TCP port number

    Tags            map[string]string `json:"tags"`            // Arbitrary metadata key-
value pairs

    HealthEndpoint  string            `json:"health_endpoint"` // HTTP path for health
checks (e.g., "/health")

    RegisteredAt    time.Time         `json:"registered_at"`   // When this instance was
first registered

    LastHeartbeat   time.Time         `json:"last_heartbeat"`  // Last successful
communication

}


// HealthStatus tracks the current health state and failure history for a service instance.

// This structure enables sophisticated health checking logic that can distinguish between

// transient network issues and genuine service failures.

type HealthStatus struct {

    Status       string    `json:"status"`        // "healthy", "unhealthy", or "unknown"
```

```
    LastCheck    time.Time `json:"last_check"`    // Timestamp of most recent health check attempt
    FailureCount int       `json:"failure_count"` // Consecutive failed health checks
    LastError    string    `json:"last_error"`    // Error message from most recent failed check
}

// RegistryEntry combines service instance information with its current health status.
// This internal structure optimizes storage and lookup operations within the registry.
type RegistryEntry struct {
    Instance     ServiceInstance `json:"instance"`
    Health       HealthStatus    `json:"health"`
    HealthTicker *time.Ticker    `json:"-"` // For health check scheduling (not serialized)
}

// ServiceList represents the response format for service discovery queries.
// It includes only healthy instances along with metadata about the query results.
type ServiceList struct {
    ServiceName     string            `json:"service_name"`
    HealthyCount    int               `json:"healthy_count"`
    UnhealthyCount  int               `json:"unhealthy_count"`
    Instances       []ServiceInstance `json:"instances"`
    QueryTimestamp  time.Time         `json:"query_timestamp"`
}
```

HTTP Utilities (`internal/api/responses.go`):

```go
package api

import (
    "encoding/json"
    "net/http"
)

// ErrorResponse provides a consistent format for API error responses.
// This structure ensures clients can reliably parse error information.
type ErrorResponse struct {
    Error   string `json:"error"`
    Code    int    `json:"code"`
    Message string `json:"message"`
}

// SuccessResponse wraps successful API responses with metadata.
type SuccessResponse struct {
    Data    interface{} `json:"data"`
    Success bool        `json:"success"`
}

// WriteJSONResponse sends a JSON response with the specified status code.
// This utility function handles content-type headers and error handling consistently.
func WriteJSONResponse(w http.ResponseWriter, statusCode int, data interface{}) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(statusCode)

    if err := json.NewEncoder(w).Encode(data); err != nil {
        // If JSON encoding fails, we can't send a JSON error, so send plain text
```

```
        http.Error(w, "Internal server error: failed to encode JSON",
            http.StatusInternalServerError)
        return
    }
}

// WriteErrorResponse sends a standardized error response.
func WriteErrorResponse(w http.ResponseWriter, statusCode int, message string) {
    response := ErrorResponse{
        Error:   http.StatusText(statusCode),
        Code:    statusCode,
        Message: message,
    }
    WriteJSONResponse(w, statusCode, response)
}

// WriteSuccessResponse sends a standardized success response.
func WriteSuccessResponse(w http.ResponseWriter, data interface{}) {
    response := SuccessResponse{
        Data:    data,
        Success: true,
    }
    WriteJSONResponse(w, http.StatusOK, response)
}
```

Core Logic Skeletons

Registry Core Interface (`internal/registry/registry.go`):

```go
package registry


import (

    "sync"

    "time"

    "context"

)


// ServiceRegistry maintains the authoritative database of service instances and their
health status.

// All operations are thread-safe and support concurrent access from multiple goroutines.

type ServiceRegistry struct {

    // Services maps service names to their registered instances

    // Key: service name, Value: map of instance ID to RegistryEntry

    services map[string]map[string]*RegistryEntry



    // Protects concurrent access to the services map

    mutex sync.RWMutex



    // Configuration parameters

    config *Config



    // Cleanup ticker for TTL enforcement

    cleanupTicker *time.Ticker

    cleanupDone   chan bool

}


// NewServiceRegistry creates and initializes a new service registry.

func NewServiceRegistry(config *Config) *ServiceRegistry {
```

```
    registry := &ServiceRegistry{
        services:    make(map[string]map[string]*RegistryEntry),
        config:      config,
        cleanupDone: make(chan bool),
    }

    // Start background cleanup process
    registry.cleanupTicker = time.NewTicker(config.CleanupInterval)
    go registry.cleanupExpiredServices()

    return registry
}

// Register adds a new service instance to the registry and returns its unique ID.
// If the instance already exists, it updates the registration timestamp.
func (r *ServiceRegistry) Register(instance ServiceInstance) (string, error) {
    // TODO 1: Generate unique instance ID if not provided (use host:port or UUID)
    // TODO 2: Acquire write lock on registry mutex
    // TODO 3: Create service name map if this is the first instance of this service
    // TODO 4: Check if instance already exists - if so, update registration time
    // TODO 5: Create RegistryEntry with initial health status "unknown"
    // TODO 6: Set RegisteredAt and LastHeartbeat to current time
    // TODO 7: Store entry in services map under service name and instance ID
    // TODO 8: Release lock and return instance ID
    // Hint: Use fmt.Sprintf("%s:%d", host, port) for simple ID generation
    return "", nil
}
```

```go
// Deregister removes a service instance from the registry by service name and instance ID.
func (r *ServiceRegistry) Deregister(serviceName, instanceID string) error {
    // TODO 1: Acquire write lock on registry mutex
    // TODO 2: Check if service name exists in services map
    // TODO 3: Check if instance ID exists for that service
    // TODO 4: Remove instance from service instances map
    // TODO 5: If no instances remain for service, remove service entirely
    // TODO 6: Release lock and return success
    // Hint: Return custom error types for "service not found" vs "instance not found"
    return nil
}

// Lookup returns all healthy instances of the specified service.
func (r *ServiceRegistry) Lookup(serviceName string) ([]ServiceInstance, error) {
    // TODO 1: Acquire read lock on registry mutex
    // TODO 2: Check if service exists in services map
    // TODO 3: Iterate through all instances for this service
    // TODO 4: Filter instances where health status is "healthy"
    // TODO 5: Collect ServiceInstance objects (not RegistryEntry) into result slice
    // TODO 6: Release lock and return healthy instances
    // Hint: Consider returning empty slice vs error when service exists but no healthy instances
    return nil, nil
}

// UpdateHealth updates the health status for a specific service instance.
// This method is called by the Health Checker component.
func (r *ServiceRegistry) UpdateHealth(serviceName, instanceID string, status HealthStatus) error {
```

```
    // TODO 1: Acquire write lock on registry mutex
    // TODO 2: Validate service and instance exist
    // TODO 3: Update the HealthStatus field of the RegistryEntry
    // TODO 4: Update LastHeartbeat time if status is "healthy"
    // TODO 5: Release lock and return success
    // Hint: This is the primary interface between Health Checker and Registry
    return nil
}

// GetAllServices returns a complete snapshot of all registered services and instances.
// Used by the HTTP API for administrative queries.
func (r *ServiceRegistry) GetAllServices() map[string][]ServiceInstance {
    // TODO 1: Acquire read lock on registry mutex
    // TODO 2: Create result map[string][]ServiceInstance
    // TODO 3: Iterate through all services in registry
    // TODO 4: For each service, collect all instances (healthy and unhealthy)
    // TODO 5: Copy ServiceInstance data (not RegistryEntry) to result
    // TODO 6: Release lock and return complete service map
    return nil
}

// cleanupExpiredServices runs as a background goroutine to remove stale registrations.
func (r *ServiceRegistry) cleanupExpiredServices() {
    for {
        select {
        case <-r.cleanupTicker.C:
            // TODO 1: Acquire write lock on registry mutex
            // TODO 2: Get current time and calculate TTL cutoff
```

```
            // TODO 3: Iterate through all services and instances
            // TODO 4: Remove instances where LastHeartbeat + TTL < now
            // TODO 5: Remove empty services with no remaining instances
            // TODO 6: Release lock
            // Hint: Be careful about modifying maps while iterating
        case <-r.cleanupDone:
            r.cleanupTicker.Stop()
            return
        }
    }
}
```

Milestone Checkpoints

Milestone 1 Checkpoint - Service Registry:

```bash
# Run registry core tests
go test ./internal/registry/... -v

# Expected output should show:
# - TestRegister: successful registration with auto-generated ID
# - TestDeregister: successful removal and error for non-existent instances
# - TestLookup: returns registered instances, empty for non-existent services
# - TestConcurrentAccess: no race conditions under concurrent load

# Manual verification:
# 1. Create registry instance in main.go
# 2. Register a few test services
# 3. Verify Lookup returns expected instances
# 4. Wait for TTL expiration and verify cleanup
```

Milestone 2 Checkpoint - Health Checking:

```bash
# Run health checker tests
go test ./internal/health/... -v

# Expected behavior:
# - Health checks execute on schedule
# - Failed instances get failure count incremented
# - Healthy instances reset failure count to zero
# - Registry receives proper UpdateHealth calls

# Manual verification:
# 1. Register service with valid health endpoint
# 2. Observe periodic health check logs
# 3. Stop the service and watch failure count increase
# 4. Verify instance marked unhealthy after MaxFailures
```

Milestone 3 Checkpoint - HTTP API:

```bash
# Run API tests
go test ./internal/api/... -v

# Test with curl:
curl -X POST http://localhost:8080/services \
  -H "Content-Type: application/json" \
  -d '{"name":"test-service","host":"localhost","port":3000}'

curl http://localhost:8080/services/test-service

# Expected: JSON response with service instances
# Should see: proper HTTP status codes, error handling, JSON formatting
```

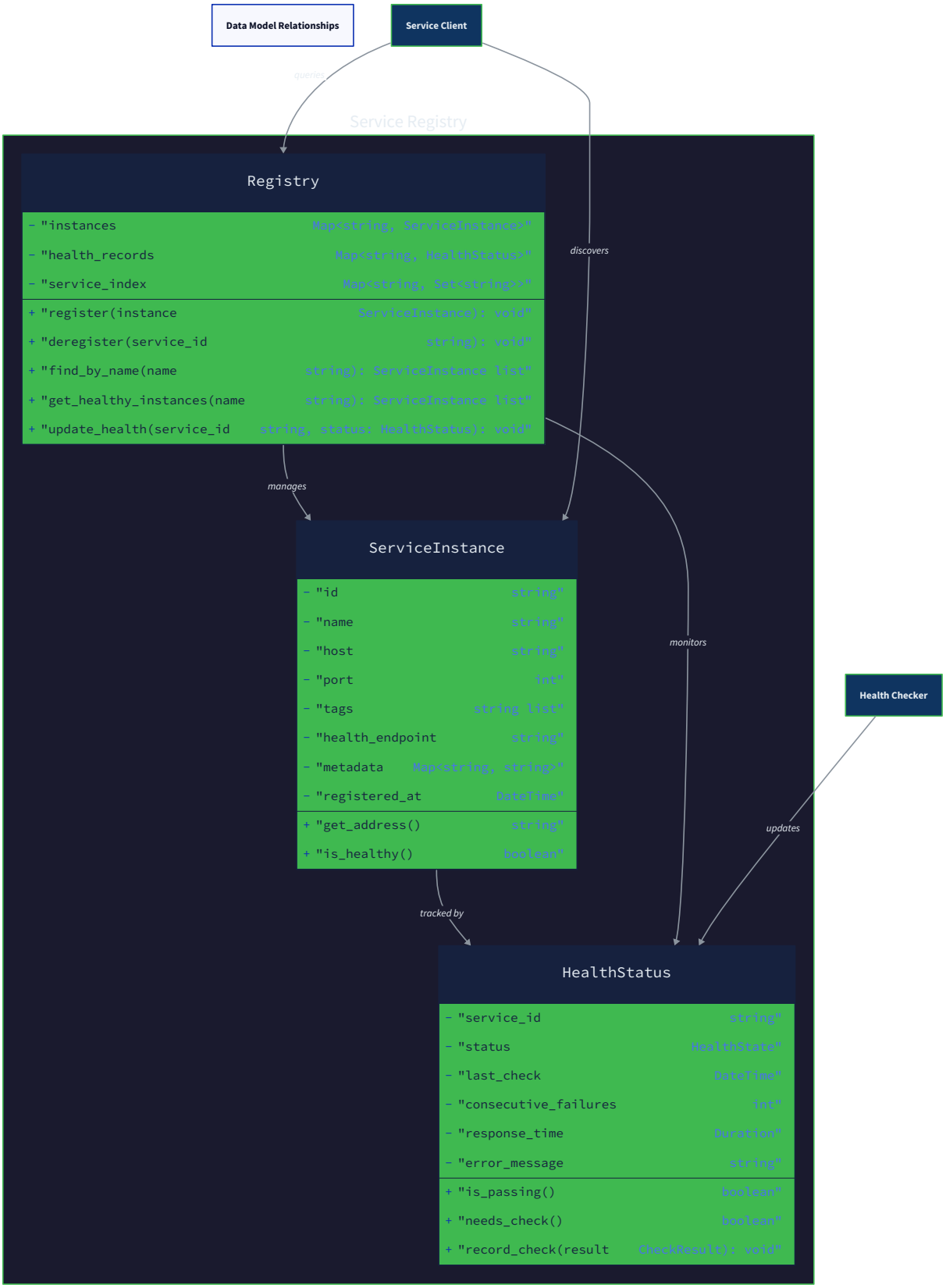
Data Model

> **Milestone(s):** All milestones (1-3) - The data model provides the foundational structures for service registration (Milestone 1), health status tracking (Milestone 2), and HTTP API responses (Milestone 3).

Mental Model: The Employee Directory

Think of our service registry as a comprehensive employee directory for a large company. Each **service instance** is like an employee record containing their name, desk location (host and port), department tags, and how to reach them for status updates (health endpoint). The **health status** is like a wellness tracker showing whether each employee is currently available, when they were last checked on, and how many consecutive sick days they've had. The **service registry** itself is like the HR database that maintains all employee records, tracks their wellness status, and provides lookup capabilities when other departments need to find someone.

Just as an employee directory needs to handle new hires (registration), departures (deregistration), relocations (updates), and sick leave (health status changes), our data model must capture all the information needed to track service instances throughout their lifecycle. The key insight is that we're not just storing static information—we're maintaining a living directory that reflects the current state of our distributed system.


Data Model Relationships
Service Client
queries
Service Registry
Registry
- "instances Map<string, ServiceInstance>"
- "health_records Map<string, HealthStatus>"
- "service_index Map<string, Set<string>>"
+ "register(instance ServiceInstance): void"
+ "deregister(service_id string): void"
+ "find_by_name(name string): ServiceInstance list"
+ "get_healthy_instances(name string): ServiceInstance list"
+ "update_health(service_id string, status: HealthStatus): void"
discovers
manages
ServiceInstance
- "id string"
- "name string"
- "host string"
- "port int"
- "tags string list"
- "health_endpoint string"
- "metadata Map<string, string>"
- "registered_at DateTime"
+ "get_address() string"
+ "is_healthy() boolean"
monitors
Health Checker
updates
tracked by
HealthStatus
- "service_id string"
- "status HealthState"
- "last_check DateTime"
- "consecutive_failures int"
- "response_time Duration"
- "error_message string"
+ "is_passing() boolean"
+ "needs_check() boolean"
+ "record_check(result CheckResult): void"

Core Data Structures

The service discovery system centers around three primary data structures that work together to maintain a complete picture of service availability and health. Each structure serves a specific purpose while maintaining clear relationships with the others.

The `ServiceInstance` represents a single running instance of a service that has registered itself with our registry. This is the fundamental unit of service discovery—each instance contains all the information needed for other services to connect to it and verify its health. The structure captures both the networking details required for communication and the metadata needed for service selection and routing decisions.

Field	Type	Description
name	string	Service name used for discovery lookups (e.g., "user-service", "payment-api")
host	string	Network host where the service instance is running (IP address or hostname)
port	int	TCP port number on which the service instance is listening for requests
tags	[]string	Metadata labels for filtering and routing (e.g., "production", "v2", "us-west")
health_endpoint	string	HTTP endpoint path for health checks (e.g., "/health", "/status")

The `HealthStatus` tracks the current wellness state of each service instance. This structure is updated continuously by our health checking system and determines whether an instance should be included in discovery results. The health status provides both current state information and historical context about recent failures.

Field	Type	Description
status	string	Current health state: "healthy", "unhealthy", or "unknown"
last_check	timestamp	When the most recent health check was performed
failure_count	int	Number of consecutive failed health checks since last success

The `ServiceRegistry` serves as the central repository that coordinates all service discovery operations. It maintains the complete mapping between service names and their instances, integrates with the health checking system, and provides thread-safe access to registry data. This structure represents the core state of our entire service discovery system.

Field	Type	Description
services	map[string][]RegistryEntry	Map from service names to lists of registered instances
health_checker	HealthChecker	Component responsible for monitoring instance health
config	Config	System configuration including timeouts, intervals, and limits

Supporting Data Structures

Beyond the core structures, several supporting types handle configuration, internal state management, and API interactions. These structures provide the scaffolding that enables the core components to work together effectively.

The `Config` structure centralizes all system configuration parameters, making it easy to adjust behavior without code changes. This structure determines how aggressively we check health, how long we wait for responses, and how quickly we clean up failed instances.

Field	Type	Description
Port	int	HTTP port for the registry API server
Host	string	Host address to bind the API server
HealthCheckInterval	duration	How frequently to check each instance's health
HealthTimeout	duration	Maximum time to wait for health check responses
MaxFailures	int	Consecutive failures before marking instance unhealthy
TTL	duration	Time-to-live for service registrations before automatic cleanup
CleanupInterval	duration	How frequently to remove expired registrations

The `RegistryEntry` serves as an internal structure that combines instance information with its health status. This structure is not exposed directly through APIs but provides the registry core with a unified view of each service instance's complete state.

Field	Type	Description
instance	ServiceInstance	The service instance registration details
health	HealthStatus	Current health status and check history
registered_at	timestamp	When this instance was first registered
instance_id	string	Unique identifier for this specific instance

API Response Structures

The HTTP API layer requires standardized response formats that provide consistent interfaces for clients. These structures ensure that all API responses follow the same patterns, making the system easier to integrate with and debug.

The `ServiceList` structure formats discovery query responses, providing clients with all the information needed to connect to healthy service instances. This structure includes both the instance details and sufficient metadata for clients to make intelligent routing decisions.

Field	Type	Description
service_name	string	Name of the requested service
instances	[]ServiceInstance	List of healthy instances available for connection
total_count	int	Total number of registered instances (including unhealthy)
healthy_count	int	Number of instances currently passing health checks

The `ErrorResponse` provides standardized error reporting across all API endpoints. Consistent error formatting helps clients handle failures gracefully and provides operators with clear diagnostic information.

Field	Type	Description
error	string	Human-readable error message describing what went wrong
error_code	string	Machine-readable error identifier for programmatic handling
details	map[string]interface{}	Additional context information specific to the error type

The `SuccessResponse` wraps successful API operations with metadata about the operation result. This structure provides confirmation of successful operations along with any relevant result data.

Field	Type	Description
success	bool	Always true for successful responses
data	interface{}	The actual result data (varies by endpoint)
message	string	Optional human-readable success message

Data Relationships and Lifecycle

Understanding how these data structures relate to each other reveals the complete picture of service discovery operations. The relationships between structures define the data flow and lifecycle management patterns that drive the entire system.

Each `ServiceInstance` in the registry has a corresponding `HealthStatus` that tracks its current wellness state. This one-to-one relationship is maintained internally through the `RegistryEntry` structure, which acts as a container that keeps instance data and health information synchronized. When a service instance is registered, both the instance details and an initial health status are created together, ensuring they remain linked throughout the instance's lifetime.

The `ServiceRegistry` maintains collections of these linked instance-health pairs, organized by service name for efficient lookup operations. The registry structure serves as the authoritative source of truth about which services exist and which instances are currently available. This centralized approach ensures

consistency across all discovery operations while providing the single point of control needed for health checking and cleanup operations.

Service instances progress through a well-defined lifecycle that's reflected in the data model. Upon registration, an instance starts with "unknown" health status until the first health check completes. Successful health checks transition the instance to "healthy" status, making it available for discovery queries. Failed health checks increment the failure count, and consecutive failures eventually transition the instance to "unhealthy" status, removing it from discovery results while keeping the registration record for potential recovery.

> The critical insight here is that instance registration and health status are managed as a unified lifecycle—you cannot have healthy instances without registration, and you cannot have useful registrations without health tracking.

Architecture Decision Records

The data model design involves several key architectural decisions that significantly impact system behavior, performance, and maintainability. Each decision represents a trade-off between different approaches, and understanding these choices helps explain why the system behaves as it does.

> **Decision: Embedded Health Status vs. Separate Health Store**
>
> - **Context**: We need to track health status for each registered service instance, and we must decide whether to store health data alongside instance data or in a separate data structure.
> - **Options Considered**: Embedded health in RegistryEntry, separate health map indexed by instance ID, external health database
> - **Decision**: Embed health status directly in RegistryEntry structure
> - **Rationale**: Co-locating instance and health data ensures atomic updates, simplifies lookup operations, and eliminates consistency issues between separate stores. The performance overhead is minimal since we access both pieces of data together in most operations.
> - **Consequences**: Enables atomic health updates, simplifies concurrent access patterns, but increases memory usage per instance and couples health checking to registry storage.

Option	Pros	Cons
Embedded health	Atomic updates, simpler lookups, consistency guaranteed	Higher memory usage, coupled storage
Separate health map	Lower memory per instance, flexible health backends	Consistency challenges, complex updates
External health database	Scalable storage, independent scaling	Network overhead, eventual consistency

Decision: Service Name as Primary Index vs. Instance ID

- **Context**: The registry must support both service discovery (find instances by name) and health updates (update specific instances), requiring an indexing strategy that supports both access patterns efficiently.
- **Options Considered**: Service name primary index with linear instance search, instance ID primary index with reverse name lookup, dual indexing with both approaches
- **Decision**: Service name as primary index with instance ID for secondary access
- **Rationale**: Service discovery is the primary use case and must be optimized for performance. Health updates happen less frequently and can tolerate linear search within a service's instance list, which is typically small (under 100 instances per service).
- **Consequences**: Fast service discovery lookups, simple data structure, but slower instance-specific operations and potential for duplicate instance handling.

Option	Pros	Cons
Service name primary	Fast discovery, simple structure	Slower instance updates
Instance ID primary	Fast instance operations	Slow discovery, complex structure
Dual indexing	Fast both operations	Memory overhead, consistency complexity

> **Decision: Failure Count vs. Failure History**
>
> - **Context**: Health checking needs to distinguish between transient failures and persistent problems, requiring a decision about how much failure history to maintain for each instance.
> - **Options Considered**: Simple failure counter, sliding window of recent checks, complete failure history with timestamps
> - **Decision**: Simple failure counter with reset on success
> - **Rationale**: A failure counter provides sufficient information to detect persistent failures while keeping memory usage constant. More complex history tracking adds little value since we only need to know "is this instance consistently failing" rather than detailed failure patterns.
> - **Consequences**: Enables effective failure detection with minimal memory overhead, but loses information about failure patterns and recovery timing.

Option	Pros	Cons
Simple counter	Low memory, easy logic	Limited failure insight
Sliding window	Better failure patterns	Higher memory, complex logic
Complete history	Full failure analysis	Memory growth, processing overhead

Data Validation and Constraints

The data model enforces several validation rules and constraints that ensure system correctness and prevent common configuration errors. These constraints are checked at data creation time and help maintain registry integrity throughout system operation.

Service instance validation ensures that all networking information is properly formatted and reachable. The service name must follow DNS-compatible naming conventions since it's used for discovery lookups that may integrate with DNS-based systems. Host addresses must be valid IP addresses or resolvable hostnames, and port numbers must fall within the valid TCP port range (1-65535). Health endpoint paths must be valid HTTP paths starting with a forward slash.

Health status validation maintains consistency in status reporting and prevents invalid state transitions. The status field accepts only predefined values ("healthy", "unhealthy", "unknown") to ensure consistent interpretation across all system components. Failure counts must be non-negative integers, and timestamp fields must represent valid time values to prevent temporal inconsistencies in health tracking.

Configuration validation prevents system misconfiguration that could lead to performance problems or operational failures. Health check intervals must be positive values to ensure checks actually occur, and timeouts must be shorter than intervals to prevent check backlog. The maximum failure threshold must be at least 1 to ensure that failures can actually trigger status changes, and TTL values must be longer than health check intervals to prevent premature instance expiration.

Memory Management and Scaling Considerations

The data model's memory usage patterns directly impact system scalability and operational characteristics. Understanding these patterns helps operators plan capacity and detect potential memory issues before they impact system performance.

Each service instance consumes a predictable amount of memory based on the string lengths in its service name, host address, tags, and health endpoint path. The base structure overhead is approximately 200-300 bytes per instance, with additional memory proportional to the total length of string fields. Tags have the highest variability in memory usage since they're user-defined and can vary significantly between deployments.

Health status tracking adds minimal memory overhead per instance, consuming approximately 50-100 bytes for status strings, timestamps, and failure counters. The memory usage remains constant over time since we don't maintain failure history beyond the current failure count, making memory consumption predictable and bounded.

The registry's memory usage scales linearly with the number of registered instances across all services. With typical instance metadata, the system can comfortably handle 10,000-50,000 service instances in memory on modern server hardware. Memory usage becomes a concern primarily in very large deployments or when instance metadata contains unusually long tag lists or host names.

> A critical scaling insight is that memory usage per instance remains constant regardless of system age—we don't accumulate historical data that causes memory to grow over time, making the system's memory footprint predictable and stable.

Common Pitfalls

Understanding common mistakes in data model implementation helps avoid subtle bugs that can cause system-wide failures or performance degradation.

⚠️ **Pitfall: Mutable Shared Structures** Many implementations accidentally share data structures between concurrent operations, leading to race conditions where one operation modifies data while another is reading it. This typically manifests as inconsistent service lists returned from discovery queries or health status updates being lost. The fix requires ensuring that all public API methods return deep copies of data structures rather than direct references to internal state, and that all internal modifications use proper synchronization.

⚠️ **Pitfall: Unbounded Tag Lists** Service instances with very large tag lists (hundreds of tags) can consume excessive memory and slow down registry operations. This often happens when automated systems generate tags containing detailed metadata or when unique identifiers are stored as tags rather than in dedicated fields. The solution involves validating tag list sizes during registration and providing guidance on appropriate tag usage patterns.

⚠️ **Pitfall: String Interning Neglect** Service names and common tag values are often repeated across many instances, but storing them as separate strings wastes memory and reduces cache locality. For example, 1000 instances of "user-service" store the service name 1000 times instead of referencing a single string. Implementing string interning for common values can significantly reduce memory usage in large deployments.

⚠️ **Pitfall: Timestamp Precision Inconsistency** Mixing different timestamp precisions (seconds vs. milliseconds vs. nanoseconds) across the health checking system leads to incorrect timeout calculations and inconsistent health status updates. This typically manifests as health checks that timeout too quickly or too slowly. The fix requires standardizing on a single timestamp precision throughout the system and explicitly converting external timestamps to the internal format.

⚠️ **Pitfall: Missing Instance Identity** Failing to assign unique identifiers to service instances makes it impossible to distinguish between multiple instances of the same service running on the same host with different ports. This leads to registration conflicts and incorrect health status updates. The solution requires generating unique instance IDs during registration and using these IDs for all instance-specific operations.

Implementation Guidance

The data model implementation focuses on creating clean, thread-safe data structures that support the concurrent access patterns required by service discovery operations. The Go language provides excellent tools for building these structures with proper synchronization and memory management.

Technology Recommendations

Component	Simple Option	Advanced Option
Data Structures	Standard Go structs with json tags	Protocol Buffers with generated types
Serialization	encoding/json for API responses	MessagePack for performance-critical paths
Thread Safety	sync.RWMutex for registry access	lock-free data structures with sync/atomic
Time Handling	time.Time for timestamps	Monotonic clocks for interval measurements
Validation	Manual validation in constructors	JSON Schema validation with external library

Recommended File Structure

```
internal/registry/
  types.go              ← Core data structures (ServiceInstance, HealthStatus, etc.)
  registry.go           ← ServiceRegistry implementation
  validation.go         ← Input validation and constraint checking
  config.go             ← Configuration loading and validation
  types_test.go         ← Unit tests for data structure behavior
api/
  responses.go          ← API response types (ServiceList, ErrorResponse, etc.)
  responses_test.go     ← API response serialization tests
```

Infrastructure Starter Code

Here's a complete configuration management system that handles loading settings from multiple sources and validating constraints:

```go
package registry

import (
    "encoding/json"
    "fmt"
    "os"
    "time"
)

// Config holds all system configuration parameters with validation
type Config struct {
    Port                int           `json:"port"`
    Host                string        `json:"host"`
    HealthCheckInterval time.Duration `json:"health_check_interval"`
    HealthTimeout       time.Duration `json:"health_timeout"`
    MaxFailures         int           `json:"max_failures"`
    TTL                 time.Duration `json:"ttl"`
    CleanupInterval     time.Duration `json:"cleanup_interval"`
}

// LoadConfig reads configuration from environment and config files
func LoadConfig() (*Config, error) {
    config := &Config{
        Port:                8080,
        Host:                "0.0.0.0",
        HealthCheckInterval: 30 * time.Second,
        HealthTimeout:       5 * time.Second,
        MaxFailures:         3,
```

```
        TTL:                300 * time.Second,
        CleanupInterval:    60 * time.Second,
    }

    // Load from config file if it exists
    if configFile := os.Getenv("CONFIG_FILE"); configFile != "" {
        if data, err := os.ReadFile(configFile); err == nil {
            json.Unmarshal(data, config)
        }
    }

    // Override with environment variables
    if port := os.Getenv("PORT"); port != "" {
        if p, err := strconv.Atoi(port); err == nil {
            config.Port = p
        }
    }

    return config, config.Validate()
}

// Validate ensures configuration values are reasonable
func (c *Config) Validate() error {
    if c.Port < 1 || c.Port > 65535 {
        return fmt.Errorf("port must be between 1 and 65535")
    }
    if c.HealthCheckInterval <= 0 {
```

```
        return fmt.Errorf("health check interval must be positive")
    }
    if c.HealthTimeout >= c.HealthCheckInterval {
        return fmt.Errorf("health timeout must be less than check interval")
    }
    if c.MaxFailures < 1 {
        return fmt.Errorf("max failures must be at least 1")
    }
    if c.TTL <= c.HealthCheckInterval {
        return fmt.Errorf("TTL must be longer than health check interval")
    }
    return nil
}
```

Here's a complete validation system that checks all input constraints and provides detailed error messages:

```go
package registry

import (
    "fmt"
    "net"
    "net/url"
    "regexp"
    "strconv"
    "strings"
)

var (
    serviceNameRegex = regexp.MustCompile(`^[a-z0-9]([a-z0-9\-]{0,61}[a-z0-9])?$`)
    tagRegex         = regexp.MustCompile(`^[a-zA-Z0-9]([a-zA-Z0-9\-_]{0,61}[a-zA-Z0-9])?$`)
)

// ValidateServiceInstance checks all constraints on service instance data
func ValidateServiceInstance(instance *ServiceInstance) error {
    if err := validateServiceName(instance.Name); err != nil {
        return fmt.Errorf("invalid service name: %w", err)
    }

    if err := validateHost(instance.Host); err != nil {
        return fmt.Errorf("invalid host: %w", err)
    }

    if err := validatePort(instance.Port); err != nil {
        return fmt.Errorf("invalid port: %w", err)
```

```
    }

    if err := validateTags(instance.Tags); err != nil {
        return fmt.Errorf("invalid tags: %w", err)
    }

    if err := validateHealthEndpoint(instance.HealthEndpoint); err != nil {
        return fmt.Errorf("invalid health endpoint: %w", err)
    }

    return nil
}

func validateServiceName(name string) error {
    if name == "" {
        return fmt.Errorf("service name cannot be empty")
    }
    if len(name) > 63 {
        return fmt.Errorf("service name too long (max 63 characters)")
    }
    if !serviceNameRegex.MatchString(name) {
        return fmt.Errorf("service name must be DNS-compatible (lowercase, digits,
hyphens)")
    }
    return nil
}

func validateHost(host string) error {
```

```
    if host == "" {
        return fmt.Errorf("host cannot be empty")
    }

    // Try parsing as IP address first
    if net.ParseIP(host) != nil {
        return nil
    }

    // Validate as hostname
    if len(host) > 253 {
        return fmt.Errorf("hostname too long (max 253 characters)")
    }

    parts := strings.Split(host, ".")
    for _, part := range parts {
        if len(part) == 0 || len(part) > 63 {
            return fmt.Errorf("invalid hostname format")
        }
    }

    return nil
}

func validatePort(port int) error {
    if port < 1 || port > 65535 {
        return fmt.Errorf("port must be between 1 and 65535")
```

```
    }
    return nil
}

func validateTags(tags []string) error {
    if len(tags) > 20 {
        return fmt.Errorf("too many tags (max 20)")
    }

    seen := make(map[string]bool)
    for _, tag := range tags {
        if tag == "" {
            return fmt.Errorf("tags cannot be empty")
        }
        if len(tag) > 63 {
            return fmt.Errorf("tag too long (max 63 characters): %s", tag)
        }
        if !tagRegex.MatchString(tag) {
            return fmt.Errorf("invalid tag format: %s", tag)
        }
        if seen[tag] {
            return fmt.Errorf("duplicate tag: %s", tag)
        }
        seen[tag] = true
    }

    return nil
```

```
}

func validateHealthEndpoint(endpoint string) error {
    if endpoint == "" {
        return fmt.Errorf("health endpoint cannot be empty")
    }
    if !strings.HasPrefix(endpoint, "/") {
        return fmt.Errorf("health endpoint must start with /")
    }

    u, err := url.Parse("http://example.com" + endpoint)
    if err != nil {
        return fmt.Errorf("invalid URL path format")
    }

    if u.Path != endpoint {
        return fmt.Errorf("health endpoint contains invalid characters")
    }

    return nil
}
```

Core Logic Skeleton Code

Here are the essential data structure definitions with comprehensive documentation and TODO comments for implementation:

```go
package registry

import (
    "sync"
    "time"
)

// ServiceInstance represents a single running instance of a service
// that has registered itself with the service registry for discovery.
type ServiceInstance struct {
    Name           string   `json:"name"`            // Service name for discovery lookups
    Host           string   `json:"host"`            // Network host (IP or hostname)
    Port           int      `json:"port"`            // TCP port number
    Tags           []string `json:"tags"`            // Metadata labels for filtering
    HealthEndpoint string   `json:"health_endpoint"` // HTTP path for health checks
}

// HealthStatus tracks the current wellness state of a service instance
// including recent check results and failure history.
type HealthStatus struct {
    Status       string    `json:"status"`        // "healthy", "unhealthy", or "unknown"
    LastCheck    time.Time `json:"last_check"`    // When most recent check occurred
    FailureCount int       `json:"failure_count"` // Consecutive failures since last
success
}

// RegistryEntry combines instance and health data for internal storage
// This structure is not exposed through public APIs but provides unified
// instance management within the registry core.
```

```go
type RegistryEntry struct {
    Instance     ServiceInstance `json:"instance"`
    Health       HealthStatus    `json:"health"`
    RegisteredAt time.Time       `json:"registered_at"`
    InstanceID   string          `json:"instance_id"`
}

// ServiceRegistry maintains the complete state of all registered services
// and coordinates between registration, health checking, and discovery operations.
type ServiceRegistry struct {
    // TODO 1: Add mutex for thread-safe access to registry data
    // TODO 2: Add services map storing RegistryEntry slices by service name
    // TODO 3: Add health_checker component for monitoring instance health
    // TODO 4: Add config for system behavior parameters
    // TODO 5: Add cleanup ticker for removing expired registrations
    // Hint: Use sync.RWMutex to allow concurrent reads but exclusive writes
}

// NewServiceRegistry creates a new registry with proper initialization
func NewServiceRegistry(config *Config) *ServiceRegistry {
    // TODO 1: Initialize the ServiceRegistry struct with empty maps and config
    // TODO 2: Start the cleanup goroutine using CleanupInterval from config
    // TODO 3: Initialize health checker component with registry reference
    // TODO 4: Return configured registry ready for service registration
    // Hint: Use make() to initialize the services map
}

// ServiceList formats discovery query responses with healthy instances
```

```
type ServiceList struct {
    ServiceName  string            `json:"service_name"`
    Instances    []ServiceInstance `json:"instances"`
    TotalCount   int               `json:"total_count"`
    HealthyCount int               `json:"healthy_count"`
}

// ErrorResponse provides standardized error reporting for API endpoints
type ErrorResponse struct {
    Error     string                 `json:"error"`
    ErrorCode string                 `json:"error_code"`
    Details   map[string]interface{} `json:"details,omitempty"`
}

// SuccessResponse wraps successful operations with result metadata
type SuccessResponse struct {
    Success bool        `json:"success"`
    Data    interface{} `json:"data"`
    Message string      `json:"message,omitempty"`
}
```

Milestone Checkpoints

After implementing the data structures, verify your implementation with these specific tests:

Milestone 1 Checkpoint - Basic Data Structures:

- Create a ServiceInstance with all fields populated and verify JSON serialization works correctly
- Validate that service name validation rejects invalid names (empty, too long, invalid characters)
- Test that host validation accepts both IP addresses and valid hostnames
- Verify that tag validation prevents empty tags, duplicates, and oversized tag lists

Expected behavior: All validation functions should return specific error messages explaining what's wrong with invalid input. JSON serialization should produce clean, readable output with all field names in

snake_case.

Milestone 2 Checkpoint - Health Status Integration:

- Create RegistryEntry instances with various health statuses and verify status transitions work correctly
- Test that failure count increments properly and resets when health recovers
- Verify that timestamp tracking accurately reflects check timing

Expected behavior: Health status should transition between "unknown" → "healthy" → "unhealthy" based on check results. Failure counts should increment on consecutive failures and reset to 0 on successful checks.

Milestone 3 Checkpoint - API Response Formatting:

- Generate ServiceList responses for services with mixed healthy/unhealthy instances
- Create ErrorResponse objects with various error codes and verify JSON formatting
- Test SuccessResponse wrapping of different data types

Expected behavior: ServiceList should only include healthy instances in the instances array but report accurate total/healthy counts. Error responses should be consistently formatted with clear error messages.

Service Registry Core

> **Milestone(s):** Milestone 1 (Service Registry) - This section covers the central component that handles service registration, deregistration, and lookup operations.

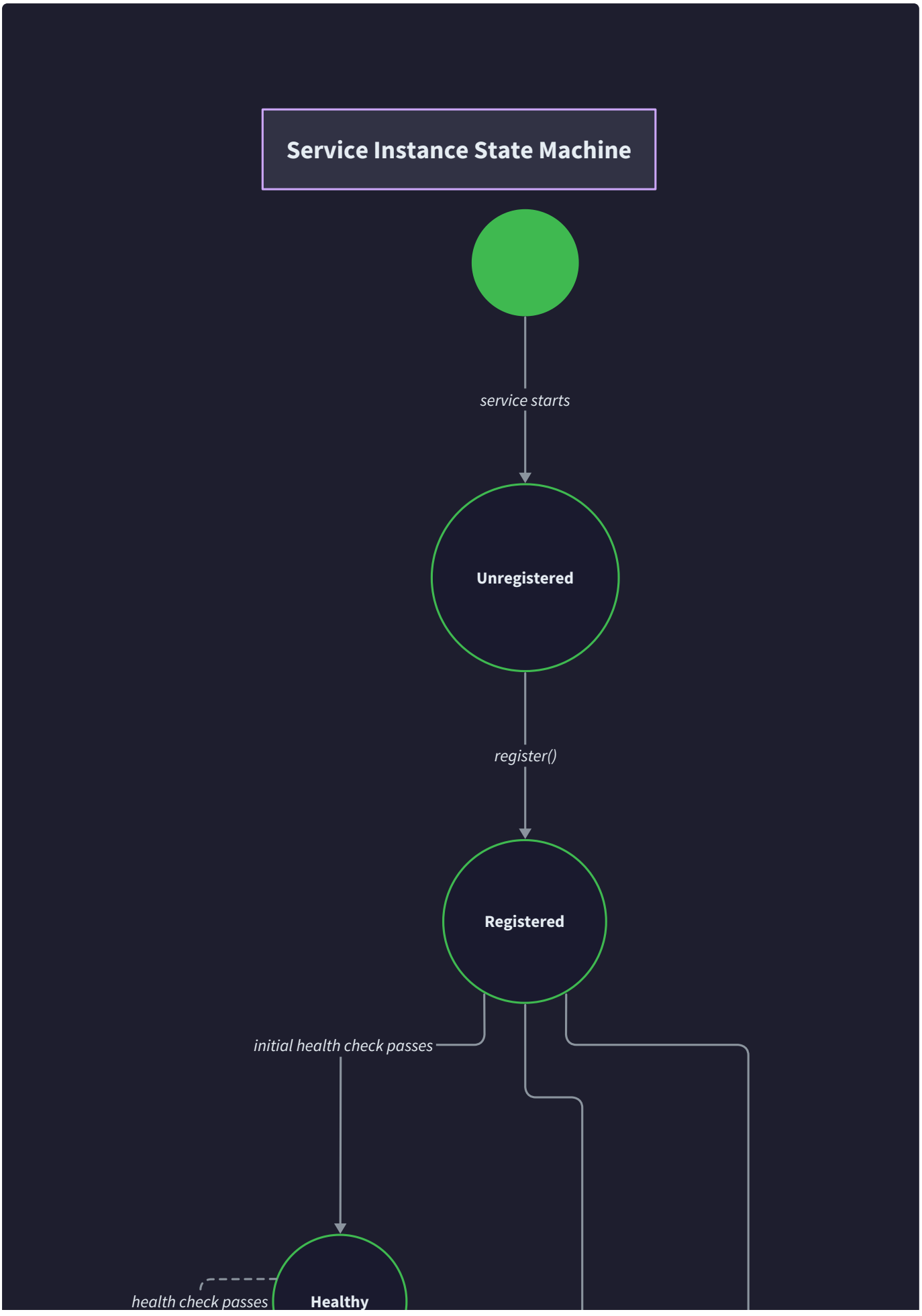
Mental Model: The Hotel Reception Desk

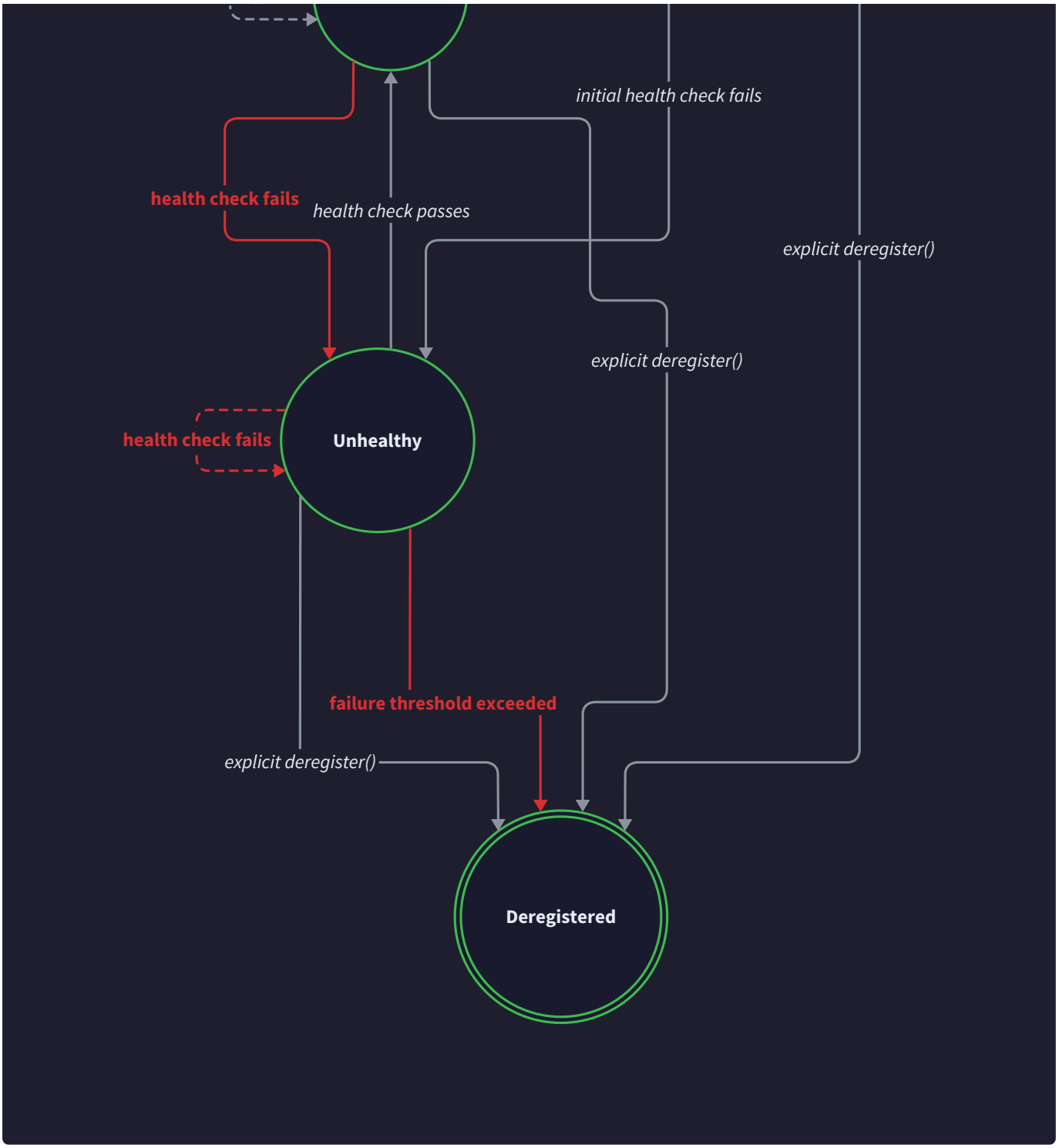
Before diving into technical details, let's build intuition about what the service registry core does by thinking of it like a hotel reception desk. When you arrive at a hotel, you approach the front desk to check in. The receptionist takes your information (name, room preferences, special requests), assigns you a room number, gives you a key card, and records your stay in their system. Throughout your stay, if someone calls asking for you, the receptionist can look up your room number and connect the call. When you check out, they remove your information from the active guest registry.

The service registry core works exactly like this hotel reception system. When a service instance starts up, it "checks in" by registering itself with the registry, providing its name, network address (host and port), and any special attributes (tags). The registry assigns it a unique instance ID and records all this information. When other services need to find instances of a particular service type, they ask the registry to "look up" that service name, and the registry returns a list of all currently registered instances, just like a receptionist providing room numbers for guests. When a service shuts down or fails, it gets "checked out" of the registry, either explicitly through deregistration or automatically through health check failures.

This mental model helps us understand the core responsibilities: **accepting registrations** (check-in), **providing lookups** (directory assistance), **maintaining current state** (guest registry), and **handling**

departures (check-out). Just as a hotel reception desk must handle multiple guests simultaneously, our registry core must safely handle concurrent operations from many services registering, deregistering, and looking up other services at the same time.


Service Instance State Machine
service starts
Unregistered
register()
Registered
initial health check passes
health check passes
Healthy



Registry Interface

The service registry core exposes a clean interface that encapsulates all service management operations. This interface defines the contract between the registry and its clients (both internal components like the health checker and external clients via the HTTP API layer).

Method Name	Parameters	Returns	Description
`Register(instance ServiceInstance)`	`instance`: Service metadata including name, host, port, tags	`string`: Unique instance ID for this registration	Adds a new service instance to the registry with generated unique identifier and current timestamp
`Deregister(instanceID string)`	`instanceID`: Unique identifier returned during registration	`error`: nil on success, error if instance not found	Removes the specified service instance from active registry and stops health checking
`Lookup(serviceName string)`	`serviceName`: Name of service type to find	`ServiceList`: Container with healthy instances, counts, metadata	Returns all currently healthy instances of the specified service name
`GetAllServices()`	None	`map[string]ServiceList`: Map of service name to instance list	Returns complete snapshot of all registered services and their healthy instances
`UpdateHealth(serviceName, instanceID string, status HealthStatus)`	Service name, instance ID, new health status object	`error`: nil on success, error if instance not found	Updates health status for specified instance, used by health checker component

Method Name	Parameters	Returns	Description
`GetRegistryStats()`	None	`RegistryStats`: Counts and metrics about registry state	Returns statistics about total services, instances, health status distribution
`Cleanup()`	None	`int`: Number of expired entries removed	Removes expired registrations based on TTL and cleanup policy

The interface design follows several important principles. First, it uses **value objects** like `ServiceInstance` and `HealthStatus` to ensure data consistency and make the API self-documenting. Second, it separates **read operations** (`Lookup`, `GetAllServices`) from **write operations** (`Register`, `Deregister`, `UpdateHealth`) to enable different concurrency patterns. Third, it returns **structured data** rather than raw maps or slices, making it easier for callers to handle the results correctly.

The `Register` method returns a unique instance ID that becomes the handle for all future operations on that specific instance. This design choice is crucial because it allows multiple instances of the same service to run on the same host (perhaps on different ports) while maintaining distinct identities. The instance ID also provides a stable identifier even if service metadata changes.

The `Lookup` method intentionally returns only healthy instances, implementing the core service discovery principle that clients should never receive references to services that cannot handle requests. The health filtering happens transparently within the registry, so clients don't need to understand health status details.

Core Algorithms

The service registry core implements three fundamental algorithms that handle the primary use cases: registration, deregistration, and lookup. Each algorithm must handle concurrent access safely while maintaining data consistency.

Service Registration Algorithm

The registration process creates a new registry entry and initializes health checking for the instance:

1. **Generate unique instance ID** using a combination of timestamp, service name, and random component to ensure global uniqueness across all registrations
2. **Validate service instance data** by checking required fields (name, host, port), validating host format and port range, and ensuring service name follows naming conventions

3. **Acquire write lock** on the registry's service map to ensure exclusive access during the modification operation
4. **Check for duplicate registration** by examining existing entries for the same service name and looking for identical host-port combinations within that service
5. **Create registry entry** with the validated service instance, initial health status set to "unknown", current registration timestamp, and generated instance ID
6. **Store entry in registry** by adding it to the services map under the appropriate service name, creating the service name key if this is the first instance
7. **Release write lock** to allow other operations to proceed
8. **Initialize health checking** by scheduling the first health check for this instance and adding it to the health checker's monitoring list
9. **Log registration event** with service name, instance ID, and network address for debugging and auditing purposes
10. **Return instance ID** to the caller for use in future deregistration or update operations

The registration algorithm includes important safety checks. Duplicate detection prevents the same service instance from being registered multiple times, which could lead to double health checking or incorrect instance counts. The validation step catches configuration errors early before they can cause runtime problems.

Service Deregistration Algorithm

The deregistration process removes an instance from active service and cleans up associated resources:

1. **Validate instance ID format** to ensure it matches the expected structure and hasn't been corrupted
2. **Acquire write lock** on the registry's service map to prevent concurrent modifications during removal
3. **Locate registry entry** by searching through service name maps to find the entry with matching instance ID
4. **Verify entry exists** and return appropriate error if the instance ID is not found in the registry
5. **Remove entry from services map** and clean up the service name key if this was the last instance of that service type
6. **Release write lock** to restore concurrent access for other operations
7. **Stop health checking** by removing this instance from the health checker's monitoring schedule
8. **Log deregistration event** with service name, instance ID, and reason for removal (explicit vs. health failure)
9. **Update registry statistics** to reflect the reduced instance count and service availability
10. **Return success confirmation** to indicate the deregistration completed successfully

The deregistration algorithm handles both explicit deregistration (when services shut down cleanly) and implicit deregistration (when health checks detect failures). The cleanup step is important for preventing memory leaks in long-running registry instances.

Service Lookup Algorithm

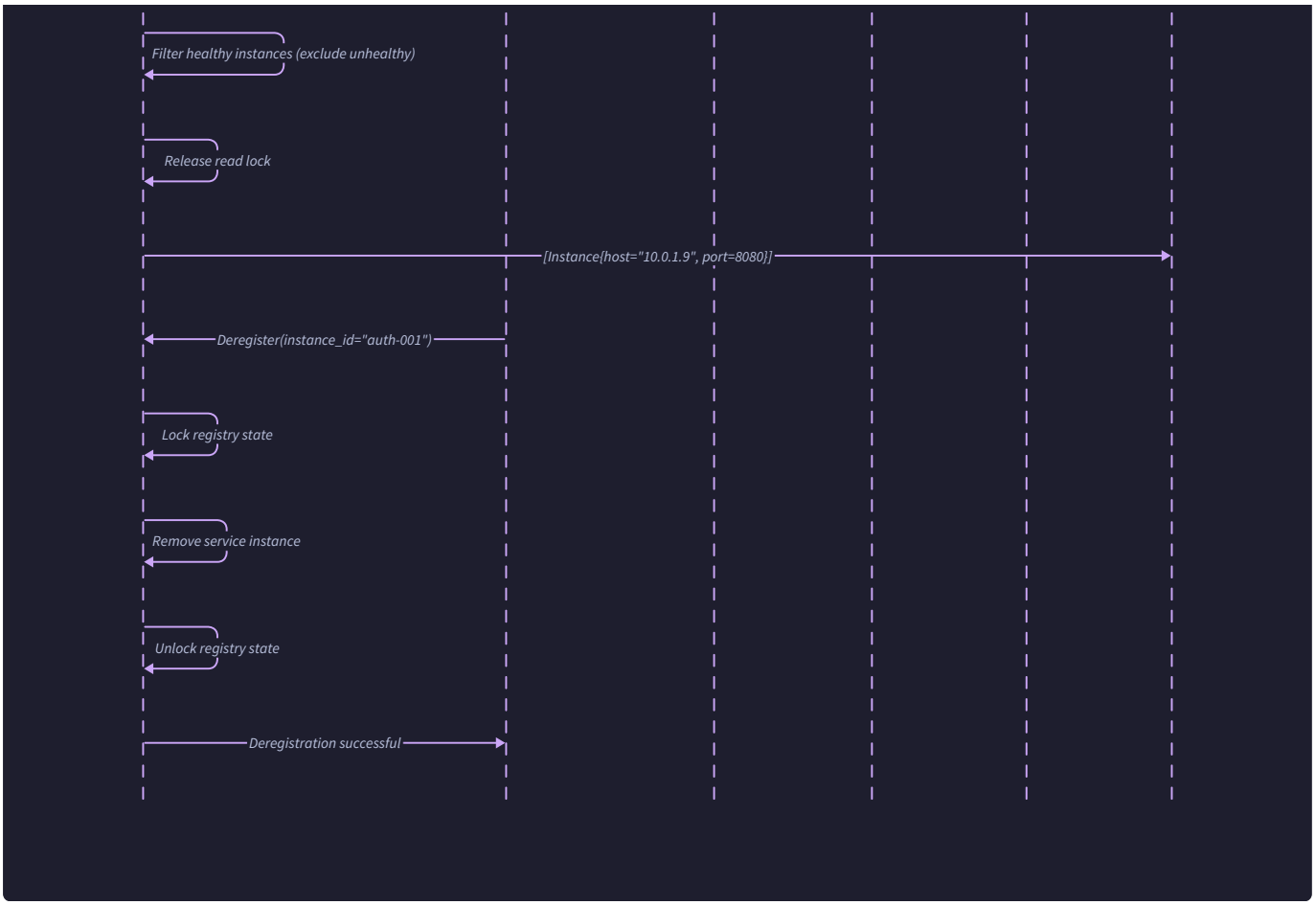
The lookup process finds healthy instances of a requested service and returns them in a client-friendly format:

1. **Acquire read lock** on the registry's service map to ensure consistent view of data during the lookup operation
2. **Check if service name exists** in the registry and return empty result if no instances have ever been registered for this service
3. **Retrieve all instances** for the requested service name from the services map
4. **Filter for healthy instances** by examining the health status of each registered instance and including only those marked as healthy
5. **Sort instances by health score** to provide deterministic ordering and potentially prioritize recently healthy instances
6. **Release read lock** to allow concurrent access to resume
7. **Build service list response** with healthy instances, total instance count, healthy instance count, and service metadata
8. **Log lookup operation** with service name, number of instances found, and requesting client information for observability
9. **Return service list** containing the filtered and formatted results

The lookup algorithm implements **read-preferring locking** to allow multiple concurrent lookups while still providing consistency. The health filtering is the critical step that ensures clients only receive references to services that can actually handle requests.


Registry
Service A
Service B
Service C
Health Checker
Client
Register(name="auth", host="10.0.1.5", port=8080)
Lock registry state
Validate registration data
Store service instance
Unlock registry state
Registration successful (instance_id="auth-001")
Register(name="user", host="10.0.1.7", port=8081)
Lock registry state
Store service instance
Unlock registry state
Registration successful (instance_id="user-001")
Lookup(service_name="auth")
Read lock registry state
Filter healthy instances
Release read lock
[Instance{host="10.0.1.5", port=8080}]
UpdateHealth(instance_id="auth-001", status=HEALTHY)


Lock registry state
Update health status
Unlock registry state
Register(name="auth", host="10.0.1.9", port=8080)
Lock registry state
Store service instance
Unlock registry state
Registration successful (instance_id="auth-002")
Lookup(service_name="auth")
Read lock registry state
Filter healthy instances
Release read lock
[Instance{host="10.0.1.5"}, Instance{host="10.0.1.9"}]
UpdateHealth(instance_id="auth-001", status=UNHEALTHY)
Lock registry state
Update health status
Unlock registry state
Lookup(service_name="auth")
Read lock registry state



Architecture Decision Records

The service registry core involves several critical design decisions that affect performance, correctness, and operational characteristics.

Decision: Instance Identity and Uniqueness

- **Context**: Services may start multiple instances on the same host, restart with different configurations, or run in containers with dynamic addresses. The registry needs a way to uniquely identify each service instance across these scenarios.
- **Options Considered**: Host-port combination as ID, service-generated UUIDs, registry-generated sequential IDs, registry-generated composite IDs
- **Decision**: Registry-generated composite IDs combining timestamp, service name hash, and random component
- **Rationale**: Host-port combinations fail when services restart on the same address. Service-generated UUIDs require coordination and may collide. Sequential IDs don't provide enough entropy. Composite IDs provide uniqueness guarantees while encoding useful metadata.
- **Consequences**: Enables reliable instance tracking across restarts. Requires instance ID storage by services for deregistration. Provides debugging information through ID structure.

Option	Pros	Cons
Host-Port ID	Simple, natural mapping	Fails on restart, no multi-port support
Service UUID	Service controls identity	Collision risk, coordination needed
Sequential ID	Simple implementation	Poor distribution, no metadata
Composite ID	Unique, informative, scalable	Slightly more complex generation

Decision: Concurrent Access Pattern

- **Context**: Multiple services will register, deregister, and lookup concurrently. Health checker will update instance status in background. Registry must maintain consistency while providing good performance.
- **Options Considered**: Single mutex for all operations, read-write mutex with coarse locking, fine-grained per-service locking, lock-free data structures
- **Decision**: Read-write mutex with service-level granularity
- **Rationale**: Single mutex creates contention bottleneck. Lock-free structures are complex and error-prone for beginners. Fine-grained locking provides good concurrency while remaining comprehensible.
- **Consequences**: Enables concurrent lookups of different services. Requires careful lock ordering to prevent deadlocks. Balances performance with implementation complexity.

Option	Pros	Cons
Single Mutex	Simple, obviously correct	Poor concurrency, lookup bottleneck
Coarse RW Mutex	Better read concurrency	Write operations still block everything
Fine-grained Locks	Best concurrency	Complex deadlock prevention
Lock-free	Maximum performance	High implementation complexity

Decision: TTL and Cleanup Strategy

- **Context**: Services may crash without deregistering, leaving stale entries. Network partitions may prevent health checks. Registry needs automatic cleanup to prevent unbounded growth.
- **Options Considered**: No TTL with health-only cleanup, fixed TTL with periodic renewal, adaptive TTL based on health check intervals, hybrid health and TTL approach
- **Decision**: Hybrid approach with fixed TTL and health check override
- **Rationale**: Health-only cleanup fails during network partitions. Fixed TTL may remove healthy services during temporary network issues. Adaptive TTL adds complexity. Hybrid provides safety net while respecting health status.
- **Consequences**: Prevents memory leaks from stale registrations. Requires services to periodically re-register. Provides graceful degradation during network issues.

Option	Pros	Cons
Health-only	Respects actual service state	Fails during network partitions
Fixed TTL	Guarantees cleanup	May remove healthy services
Adaptive TTL	Adjusts to conditions	Complex tuning and implementation
Hybrid	Safety with flexibility	Requires careful parameter tuning

Common Pitfalls

Implementing a service registry involves several subtle issues that can cause serious problems in production. Understanding these pitfalls helps avoid them during initial development and debugging.

⚠️ Pitfall: Race Conditions in Registration

A common mistake is not properly synchronizing access to the registry's internal data structures during registration. This often manifests as corrupted service lists, duplicate entries, or panics when multiple services register simultaneously. The issue occurs because developers assume registration is infrequent and don't consider concurrent access patterns.

The problem becomes visible when running load tests or during startup scenarios where many services register simultaneously. Symptoms include inconsistent service counts, missing registrations, or runtime panics with map access violations.

Fix: Always acquire appropriate locks before modifying shared data structures. Use read-write mutexes to allow concurrent reads while protecting writes. Ensure lock granularity matches your consistency requirements.

⚠️ Pitfall: Memory Leaks from Failed Deregistrations

Services that crash or lose network connectivity cannot perform explicit deregistration, leading to stale entries accumulating in the registry over time. Without proper cleanup mechanisms, the registry's memory usage grows unbounded, eventually causing performance degradation or crashes.

This pitfall is particularly dangerous because it manifests slowly in production. Early testing with clean shutdowns won't reveal the issue, but long-running deployments with occasional service failures will experience gradual memory growth.

Fix: Implement TTL-based cleanup for all registrations. Run periodic background cleanup processes. Combine health check failures with automatic deregistration. Set reasonable upper bounds on registry size.

⚠️ Pitfall: Inconsistent Health State During Updates

When the health checker updates instance health status, there's a window where lookup operations might see inconsistent data. For example, a lookup might return an instance that failed health checks but hasn't been removed yet, or miss an instance that just became healthy.

This inconsistency can cause cascading failures where clients receive references to unhealthy services, leading to request timeouts and retry storms. The problem is particularly acute during network partitions or service deployment scenarios.

Fix: Use atomic updates for health status changes. Design the health update process to be idempotent. Consider implementing eventual consistency patterns where appropriate, and ensure clients have proper timeout and retry logic.

⚠️ Pitfall: Blocking Operations in Critical Paths

Registry operations that perform I/O, network calls, or expensive computations while holding locks can create severe performance bottlenecks. This commonly happens when developers add logging, metrics collection, or validation that makes external calls within locked sections.

The symptom is high latency for all registry operations, even simple lookups, because operations queue behind slow critical sections. This can make the entire service discovery system unresponsive.

Fix: Minimize work performed while holding locks. Move expensive operations outside critical sections. Use asynchronous patterns for non-essential operations like logging and metrics. Profile lock contention regularly.

⚠️ Pitfall: Inadequate Input Validation

Failing to properly validate service registration data can lead to runtime errors, injection attacks, or data corruption. Common validation failures include accepting malformed hostnames, invalid port numbers, excessively long service names, or special characters that break downstream systems.

These issues often surface when integrating with external systems or during security audits. Invalid data can propagate through the system, causing failures far from the original input point.

Fix: Implement comprehensive input validation for all registry operations. Validate hostname formats, port ranges, string lengths, and character sets. Sanitize input data and provide clear error messages for validation failures. Consider using schema validation libraries.

Implementation Guidance

This section provides concrete technical recommendations and starter code for implementing the service registry core. The focus is on Go-based implementation with alternatives noted for other languages.

Technology Recommendations

Component	Simple Option	Advanced Option
Concurrency	`sync.RWMutex` for basic locking	`sync.Map` for high-concurrency scenarios
ID Generation	`time.Now().UnixNano()` + `rand.Int63()`	UUID library with cryptographic randomness
Data Storage	In-memory `map[string]interface{}`	Embedded database like BadgerDB or BoltDB
Serialization	`encoding/json` for configuration	Protocol Buffers for performance-critical paths
Logging	Standard `log` package	Structured logging with `logrus` or `zap`
Configuration	Environment variables + flags	Configuration management library like Viper

Recommended File Structure

The service registry core should be organized as a self-contained package that can be imported by other components:

```
service-discovery/
├── cmd/
│   └── registry/
│       └── main.go                    ← Entry point for standalone registry
├── internal/
│   ├── registry/                      ← Core registry package
│   │   ├── registry.go                ← Main ServiceRegistry implementation
│   │   ├── registry_test.go           ← Unit tests for registry operations
│   │   ├── types.go                   ← Data structures and interfaces
│   │   ├── validation.go              ← Input validation utilities
│   │   └── config.go                  ← Configuration management
│   ├── health/                        ← Health checker component
│   │   └── checker.go
│   └── api/                           ← HTTP API layer
│       └── handlers.go
└── pkg/
    └── client/                        ← Client library for services
        └── client.go
```

Infrastructure Starter Code

Here's complete, working infrastructure code that handles the non-core functionality:

Configuration Management (`internal/registry/config.go`):

```go
package registry

import (
    "flag"
    "fmt"
    "os"
    "strconv"
    "time"
)

type Config struct {
    Port                int           `json:"port"`
    Host                string        `json:"host"`
    HealthCheckInterval time.Duration `json:"health_check_interval"`
    HealthTimeout       time.Duration `json:"health_timeout"`
    MaxFailures         int           `json:"max_failures"`
    TTL                 time.Duration `json:"ttl"`
    CleanupInterval     time.Duration `json:"cleanup_interval"`
}

func LoadConfig() (*Config, error) {
    config := &Config{
        Port:                8080,
        Host:               "localhost",
        HealthCheckInterval: 30 * time.Second,
        HealthTimeout:       5 * time.Second,
        MaxFailures:         3,
        TTL:                5 * time.Minute,
```

```
        CleanupInterval:   1 * time.Minute,
    }

    // Command line flags override defaults
    flag.IntVar(&config.Port, "port", config.Port, "Registry HTTP port")
    flag.StringVar(&config.Host, "host", config.Host, "Registry host address")
    flag.DurationVar(&config.HealthCheckInterval, "health-interval",
config.HealthCheckInterval, "Health check interval")
    flag.DurationVar(&config.HealthTimeout, "health-timeout", config.HealthTimeout, "Health
check timeout")
    flag.IntVar(&config.MaxFailures, "max-failures", config.MaxFailures, "Max health check
failures")
    flag.DurationVar(&config.TTL, "ttl", config.TTL, "Registration TTL")
    flag.DurationVar(&config.CleanupInterval, "cleanup-interval", config.CleanupInterval,
"Cleanup interval")
    flag.Parse()

    // Environment variables override flags
    if port := os.Getenv("REGISTRY_PORT"); port != "" {
        if p, err := strconv.Atoi(port); err == nil {
            config.Port = p
        }
    }
    if host := os.Getenv("REGISTRY_HOST"); host != "" {
        config.Host = host
    }
    if interval := os.Getenv("HEALTH_CHECK_INTERVAL"); interval != "" {
        if d, err := time.ParseDuration(interval); err == nil {
            config.HealthCheckInterval = d
        }
```

```
    }

    return config, nil
}
```

Input Validation (`internal/registry/validation.go`):

```go
package registry


import (

    "errors"

    "fmt"

    "net"

    "strings"

)


func ValidateServiceInstance(instance ServiceInstance) error {

    if strings.TrimSpace(instance.Name) == "" {

        return errors.New("service name cannot be empty")

    }



    if len(instance.Name) > 64 {

        return errors.New("service name too long (max 64 characters)")

    }



    if !isValidServiceName(instance.Name) {

        return errors.New("service name must contain only letters, numbers, hyphens, and
underscores")

    }



    if strings.TrimSpace(instance.Host) == "" {

        return errors.New("host cannot be empty")

    }



    if net.ParseIP(instance.Host) == nil && !isValidHostname(instance.Host) {
```

```
        return errors.New("host must be valid IP address or hostname")
    }

    if instance.Port < 1 || instance.Port > 65535 {
        return errors.New("port must be between 1 and 65535")
    }

    for _, tag := range instance.Tags {
        if len(tag) > 32 {
            return fmt.Errorf("tag '%s' too long (max 32 characters)", tag)
        }
    }

    return nil
}

func isValidServiceName(name string) bool {
    for _, char := range name {
        if !((char >= 'a' && char <= 'z') ||
             (char >= 'A' && char <= 'Z') ||
             (char >= '0' && char <= '9') ||
             char == '-' || char == '_') {
            return false
        }
    }
    return true
}
```

```go
func isValidHostname(hostname string) bool {
    if len(hostname) > 253 {
        return false
    }
    for _, part := range strings.Split(hostname, ".") {
        if len(part) == 0 || len(part) > 63 {
            return false
        }
    }
    return true
}
```

Data Types (`internal/registry/types.go`):

```go
package registry

import "time"

type ServiceInstance struct {
    Name           string   `json:"name"`
    Host           string   `json:"host"`
    Port           int      `json:"port"`
    Tags           []string `json:"tags"`
    HealthEndpoint string   `json:"health_endpoint"`
}

type HealthStatus struct {
    Status       string    `json:"status"`
    LastCheck    time.Time `json:"last_check"`
    FailureCount int       `json:"failure_count"`
}

type RegistryEntry struct {
    Instance     ServiceInstance `json:"instance"`
    Health       HealthStatus    `json:"health"`
    RegisteredAt time.Time       `json:"registered_at"`
    InstanceID   string          `json:"instance_id"`
}

type ServiceList struct {
    ServiceName  string            `json:"service_name"`
    Instances    []ServiceInstance `json:"instances"`
    TotalCount   int               `json:"total_count"`
```

```
    HealthyCount int              `json:"healthy_count"`
}

type RegistryStats struct {
    TotalServices   int                    `json:"total_services"`
    TotalInstances  int                    `json:"total_instances"`
    HealthyInstances int                   `json:"healthy_instances"`
    HealthBreakdown map[string]int         `json:"health_breakdown"`
    LastCleanup     time.Time              `json:"last_cleanup"`
}
```

Core Logic Skeleton Code

The main registry implementation should be built by the learner. Here's the skeleton with detailed guidance:

Main Registry Structure (`internal/registry/registry.go`):

```go
package registry

import (
    "crypto/rand"
    "fmt"
    "sync"
    "time"
)

type ServiceRegistry struct {
    // TODO: Add field for storing service entries (map[string]map[string]*RegistryEntry)
    // TODO: Add field for protecting concurrent access (sync.RWMutex)
    // TODO: Add field for configuration (*Config)
    // TODO: Add field for cleanup ticker (*time.Ticker)
}

// NewServiceRegistry creates a new service registry with the given configuration
func NewServiceRegistry(config *Config) *ServiceRegistry {
    registry := &ServiceRegistry{
        // TODO: Initialize services map
        // TODO: Initialize mutex
        // TODO: Store config
    }

    // TODO: Start cleanup goroutine using config.CleanupInterval
    // TODO: Set up cleanup ticker and background cleanup loop

    return registry
```

```
}

// Register adds a new service instance to the registry
func (r *ServiceRegistry) Register(instance ServiceInstance) (string, error) {
    // TODO 1: Validate the service instance using ValidateServiceInstance
    // TODO 2: Generate unique instance ID using generateInstanceID helper
    // TODO 3: Acquire write lock on registry mutex
    // TODO 4: Check if service name exists in services map, create if needed
    // TODO 5: Check for duplicate host:port combination in existing instances
    // TODO 6: Create RegistryEntry with instance, initial health status, current time,
instance ID
    // TODO 7: Add entry to services map under service name and instance ID
    // TODO 8: Release write lock
    // TODO 9: Log registration event
    // TODO 10: Return instance ID and nil error

    // Hint: Initial health status should be "unknown" with zero failure count
    // Hint: Services map structure: services[serviceName][instanceID] = entry
}

// Deregister removes a service instance from the registry
func (r *ServiceRegistry) Deregister(instanceID string) error {
    // TODO 1: Validate instance ID format (non-empty, reasonable length)
    // TODO 2: Acquire write lock on registry mutex
    // TODO 3: Search through all services to find entry with matching instance ID
    // TODO 4: If found, remove entry from the services map
    // TODO 5: If service has no more instances, remove service name key entirely
    // TODO 6: Release write lock
```

```
    // TODO 7: Log deregistration event
    // TODO 8: Return nil on success, error if instance ID not found

    // Hint: You'll need nested loops to search services[serviceName][instanceID]
    // Hint: Use delete() to remove map entries
}

// Lookup returns all healthy instances of the specified service
func (r *ServiceRegistry) Lookup(serviceName string) (*ServiceList, error) {
    // TODO 1: Acquire read lock on registry mutex
    // TODO 2: Check if service name exists in services map
    // TODO 3: If not found, return empty ServiceList (not an error condition)
    // TODO 4: Iterate through all instances for the service name
    // TODO 5: Filter instances where health status is "healthy"
    // TODO 6: Count total instances and healthy instances
    // TODO 7: Release read lock
    // TODO 8: Build ServiceList with results and return

    // Hint: ServiceList should include service name, healthy instances only, counts
    // Hint: Empty result is valid - services might have no healthy instances
}

// UpdateHealth updates the health status for a specific instance
func (r *ServiceRegistry) UpdateHealth(serviceName, instanceID string, status HealthStatus)
error {
    // TODO 1: Acquire write lock on registry mutex
    // TODO 2: Verify service name exists in services map
    // TODO 3: Verify instance ID exists under that service name
```

```
    // TODO 4: Update the health status in the registry entry
    // TODO 5: Update last_check timestamp to current time
    // TODO 6: Release write lock
    // TODO 7: Return nil on success, error if service/instance not found

    // Hint: This method is called by the health checker component
    // Hint: Always update last_check even if status didn't change
}

// GetAllServices returns a complete snapshot of the registry
func (r *ServiceRegistry) GetAllServices() map[string]*ServiceList {
    // TODO 1: Acquire read lock on registry mutex
    // TODO 2: Create result map for service name to ServiceList
    // TODO 3: Iterate through all services in registry
    // TODO 4: For each service, build ServiceList with healthy instances
    // TODO 5: Add ServiceList to result map
    // TODO 6: Release read lock
    // TODO 7: Return complete result map

    // Hint: Similar to Lookup but for all services
    // Hint: Filter for healthy instances in each service
}

// Helper function to generate unique instance IDs
func generateInstanceID(serviceName string) string {
    // TODO 1: Get current timestamp as nanoseconds
    // TODO 2: Generate random bytes using crypto/rand
    // TODO 3: Combine timestamp, service name hash, and random component
```

```
    // TODO 4: Format as string and return



    // Hint: Format like "servicename-timestamp-randomhex"

    // Hint: Use fmt.Sprintf for string formatting

}


// Background cleanup process (called by ticker)

func (r *ServiceRegistry) cleanup() {

    // TODO 1: Acquire write lock on registry mutex

    // TODO 2: Iterate through all services and instances

    // TODO 3: Check TTL expiration based on registered_at timestamp and config.TTL

    // TODO 4: Remove expired entries that also failed recent health checks

    // TODO 5: Remove empty service name keys after instance removal

    // TODO 6: Release write lock

    // TODO 7: Log cleanup statistics (number of entries removed)



    // Hint: Only remove entries that are both expired AND unhealthy

    // Hint: Healthy services should be allowed to re-register before TTL expires

}
```

Milestone Checkpoints

After implementing the core registry, verify these behaviors:

Basic Functionality Test:

```bash
# Run unit tests
go test ./internal/registry/... -v

# Expected output should show all test cases passing:
# TestServiceRegistration
# TestServiceDeregistration
# TestServiceLookup
# TestConcurrentOperations
```

Manual Verification Steps:

1. Create a simple test program that registers a service instance
2. Verify the registration returns a non-empty instance ID
3. Perform a lookup for that service name and confirm the instance appears
4. Deregister using the instance ID and verify lookup returns empty results
5. Test concurrent registrations by starting multiple goroutines registering different services

Expected Behavior:

- Registration should complete in < 10ms for single instances
- Lookup should return results in < 5ms for services with < 100 instances
- Concurrent operations should not cause panics or data corruption
- Memory usage should remain stable during register/deregister cycles

Debugging Signs:

- If registrations hang: Check for deadlocks in mutex usage
- If lookups return stale data: Verify proper locking around health updates
- If memory grows continuously: Ensure cleanup process is running and working
- If panics occur: Check for concurrent map access without proper locking

Health Checker

> **Milestone(s):** Milestone 2 (Health Checking) - This section covers the background component that monitors service health and automatically removes failed instances from the registry.

Mental Model: The Security Guard Rounds

Think of the health checker as a security guard making regular rounds through an office building. Just as a security guard walks predetermined routes at scheduled intervals, checking that all doors are locked, lights are working, and areas are secure, the health checker periodically visits each registered service to verify it's still responding and functioning properly.

When the security guard discovers a problem during rounds—perhaps a door is unlocked that should be locked, or a light has burned out—they take note and report it to building management. Similarly, when the health checker detects that a service isn't responding or is returning error responses, it records the failure and updates the registry to mark that instance as unhealthy.

The security guard doesn't immediately sound an alarm after finding one small issue—they understand that temporary problems happen. Maybe someone just stepped out and forgot to lock their office door, or a light bulb chose that exact moment to fail. Instead, the guard makes note of the issue and checks again on the next round. Only after seeing the same problem persist across multiple rounds does the guard escalate it as a genuine security concern.

This mirrors how the health checker handles service failures. A single failed health check doesn't immediately remove a service from the registry—networks have temporary hiccups, services might be momentarily overloaded, or a garbage collection pause might cause a timeout. The health checker maintains a **failure count** for each service instance and only marks it as unhealthy after seeing consecutive failures that exceed a configurable threshold.

The security guard also adapts their rounds based on the building's needs. High-security areas might get checked more frequently than storage closets. Critical systems require more attention than auxiliary services. Similarly, the health checker allows different services to configure their own check intervals and failure thresholds based on their criticality and expected response patterns.

> The key insight is that health checking is fundamentally about distinguishing between temporary service hiccups and genuine failures that require removing the service from active duty.

Health Checker Interface

The health checker component exposes a well-defined interface that separates the concerns of health monitoring from the core registry operations. This interface defines the contract between the health checker and both the registry core and the external services it monitors.

Method Name	Parameters	Returns	Description
`StartHealthChecker`	`registry *ServiceRegistry`	`error`	Initializes and starts the background health checking process for all registered services
`StopHealthChecker`	None	`error`	Gracefully shuts down health checking, completing in-flight checks before stopping
`CheckServiceHealth`	`instance ServiceInstance`	`HealthStatus, error`	Performs a single health check against the specified service instance
`UpdateCheckInterval`	`serviceName string, interval time.Duration`	`error`	Modifies the health check frequency for a specific service type
`GetHealthStats`	None	`HealthStats`	Returns aggregate health statistics across all monitored services
`ConfigureHealthCheck`	`serviceName string, config HealthCheckConfig`	`error`	Updates health check configuration for a specific service type

The interface design follows the principle of **separation of concerns**—the health checker focuses solely on monitoring and updating health status, while the registry core maintains the authoritative service data and handles lookups. This separation allows the health checker to operate as an independent background process without blocking registry operations.

The `CheckServiceHealth` method performs the actual health verification logic and returns a `HealthStatus` struct that encapsulates both the current health state and metadata about the check itself:

Field	Type	Description
`Status`	`string`	Current health state: "healthy", "unhealthy", or "unknown"
`LastCheck`	`time.Time`	Timestamp of the most recent health check attempt
`FailureCount`	`int`	Number of consecutive failed health checks for this instance
`ResponseTime`	`time.Duration`	How long the most recent health check took to complete
`LastError`	`string`	Error message from the most recent failed check, empty if healthy

The health checker maintains its own internal state to track check schedules and failure counts, but it delegates all registry updates to the registry core through well-defined callback methods. This ensures that the registry remains the single source of truth for service data while allowing the health checker to operate independently.

Health Check Algorithms

The health checker operates through several coordinated algorithms that handle different aspects of the health monitoring process. These algorithms work together to provide reliable failure detection while minimizing false positives from temporary network issues.

Health Check Scheduling Algorithm

The scheduling algorithm determines when to perform health checks for each registered service. Rather than checking all services simultaneously, which could create network congestion and registry contention, the scheduler spreads checks across time to ensure smooth operation.

1. **Initialize check schedule**: When a new service registers, calculate its first check time by adding the configured `HEALTH_CHECK_INTERVAL` to the current time, plus a random jitter of up to 30 seconds to prevent thundering herd effects.
2. **Maintain priority queue**: Store all pending health checks in a time-ordered priority queue, with the next check time as the priority. This allows efficient retrieval of the next service that needs checking.
3. **Background scheduler loop**: Run a continuous background goroutine that sleeps until the next scheduled check time, then wakes up to process due health checks.
4. **Batch processing**: When multiple checks become due simultaneously (within a 1-second window), group them into batches and process them concurrently to improve throughput.
5. **Reschedule after completion**: After completing a health check, calculate the next check time based on the service's configured interval and add it back to the priority queue.
6. **Handle registration changes**: When a service deregisters, remove all its pending health checks from the schedule. When a service's check interval changes, update its next scheduled time accordingly.

The scheduling algorithm ensures that health checks are distributed evenly across time, preventing resource spikes while maintaining responsiveness to service failures.

Health Check Execution Algorithm

The execution algorithm handles the actual process of checking a service's health and interpreting the results. This algorithm must handle various failure modes gracefully while distinguishing between different types of problems.

1. **Prepare health check request**: Construct an HTTP GET request to the service's health endpoint, typically `http://{host}:{port}/health`. Set appropriate timeout values based on the service's configured `HealthTimeout`.
2. **Execute request with timeout**: Send the HTTP request using a context with deadline to ensure the check doesn't hang indefinitely. The timeout should be shorter than the check interval to prevent overlapping checks.
3. **Interpret response status**: Evaluate the HTTP response according to standard health check conventions:
    - Status codes 200-299: Service is healthy
    - Status codes 400-499: Service is unhealthy (client errors indicate service problems)
    - Status codes 500-599: Service is unhealthy (server errors indicate service failures)
    - Network timeouts or connection refused: Service is unhealthy
4. **Parse response body**: If the response includes a JSON body, parse it to extract additional health information such as dependency status, resource utilization, or detailed error messages.
5. **Calculate response metrics**: Record the total response time and any relevant performance metrics that could indicate service degradation even when the service reports healthy.
6. **Handle check failures**: When a health check fails, increment the service instance's failure count and record the error details. If the failure count exceeds the configured `MaxFailures` threshold, mark the instance as unhealthy.
7. **Handle check successes**: When a health check succeeds, reset the failure count to zero and ensure the instance is marked as healthy in the registry.

The execution algorithm focuses on reliability and clear failure detection while providing detailed information for debugging when services become unhealthy.

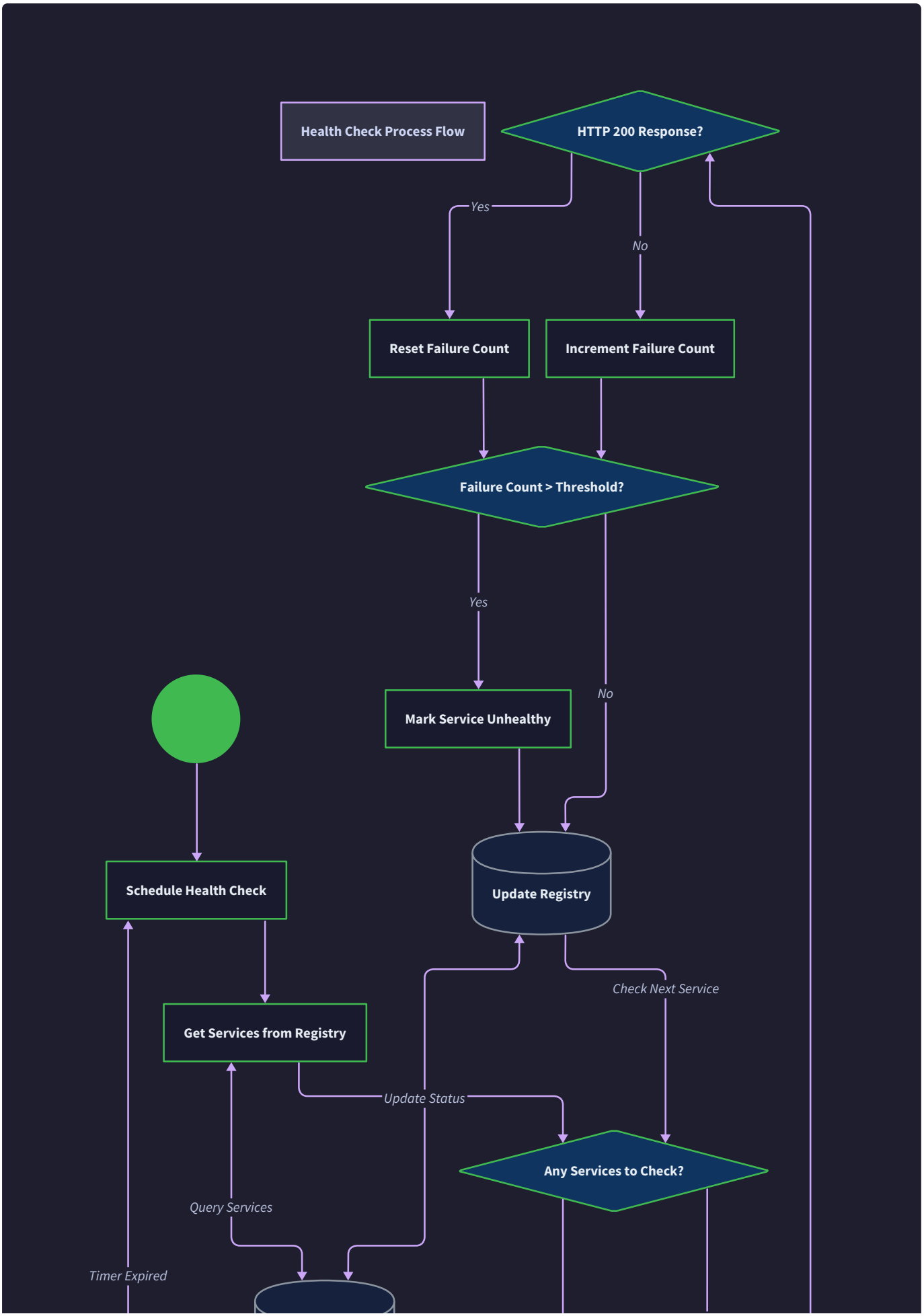
Registry Update Algorithm

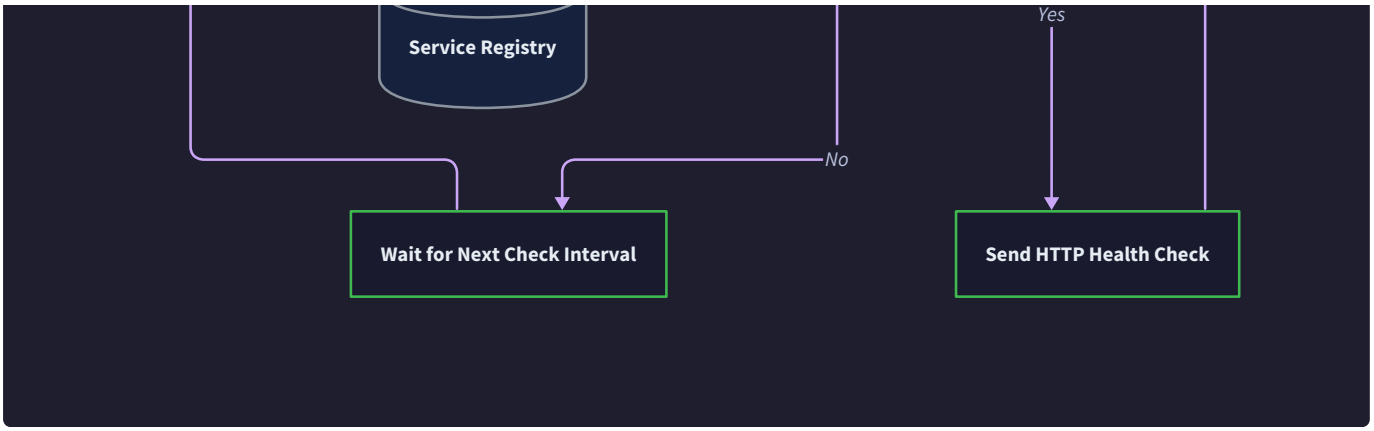
The registry update algorithm handles the process of applying health check results to the service registry state. This algorithm must coordinate carefully with the registry core to ensure consistency and avoid race conditions.

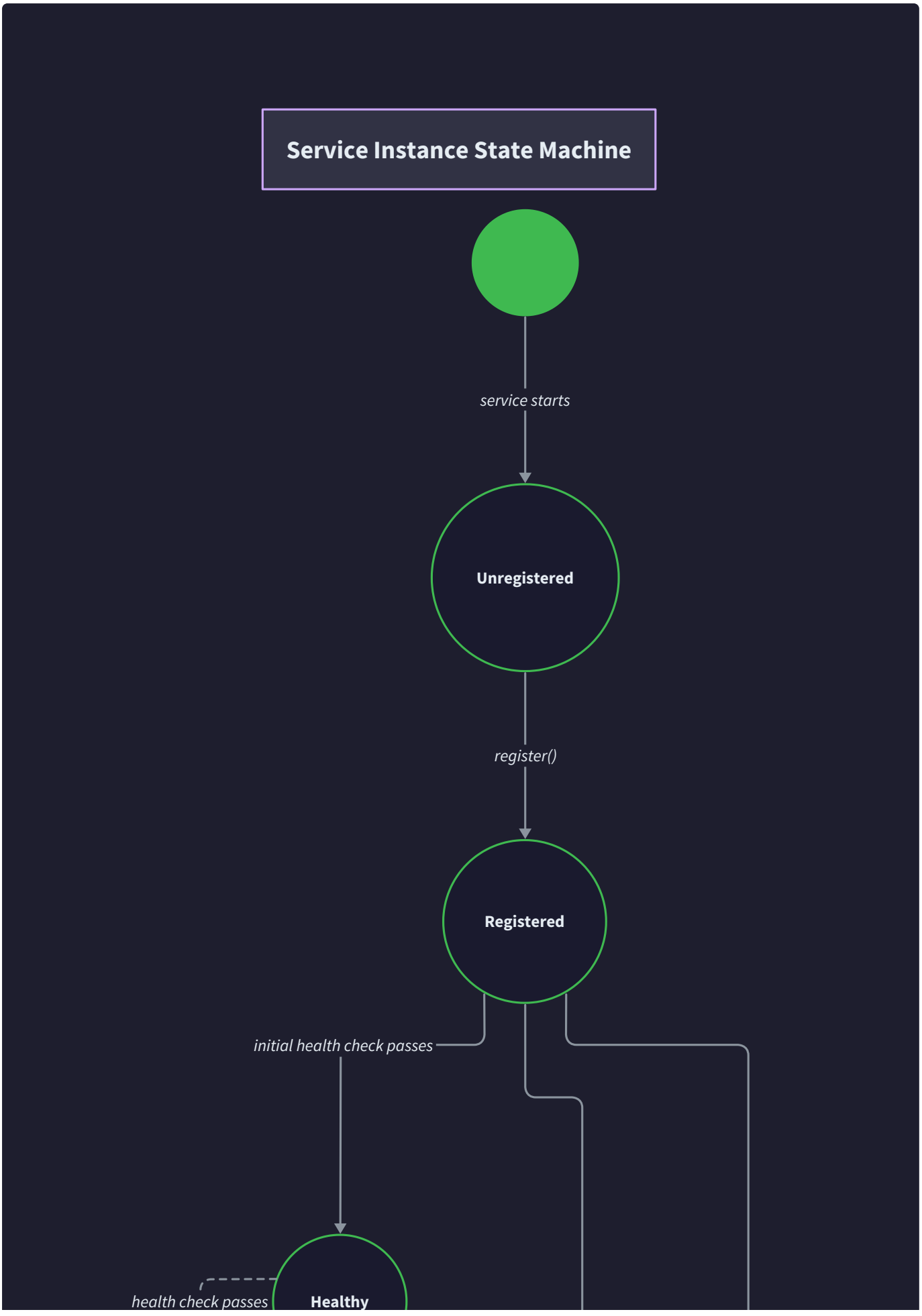
1. **Acquire instance lock**: Before updating health status, acquire a lock on the specific service instance to prevent concurrent modifications from registration/deregistration operations.

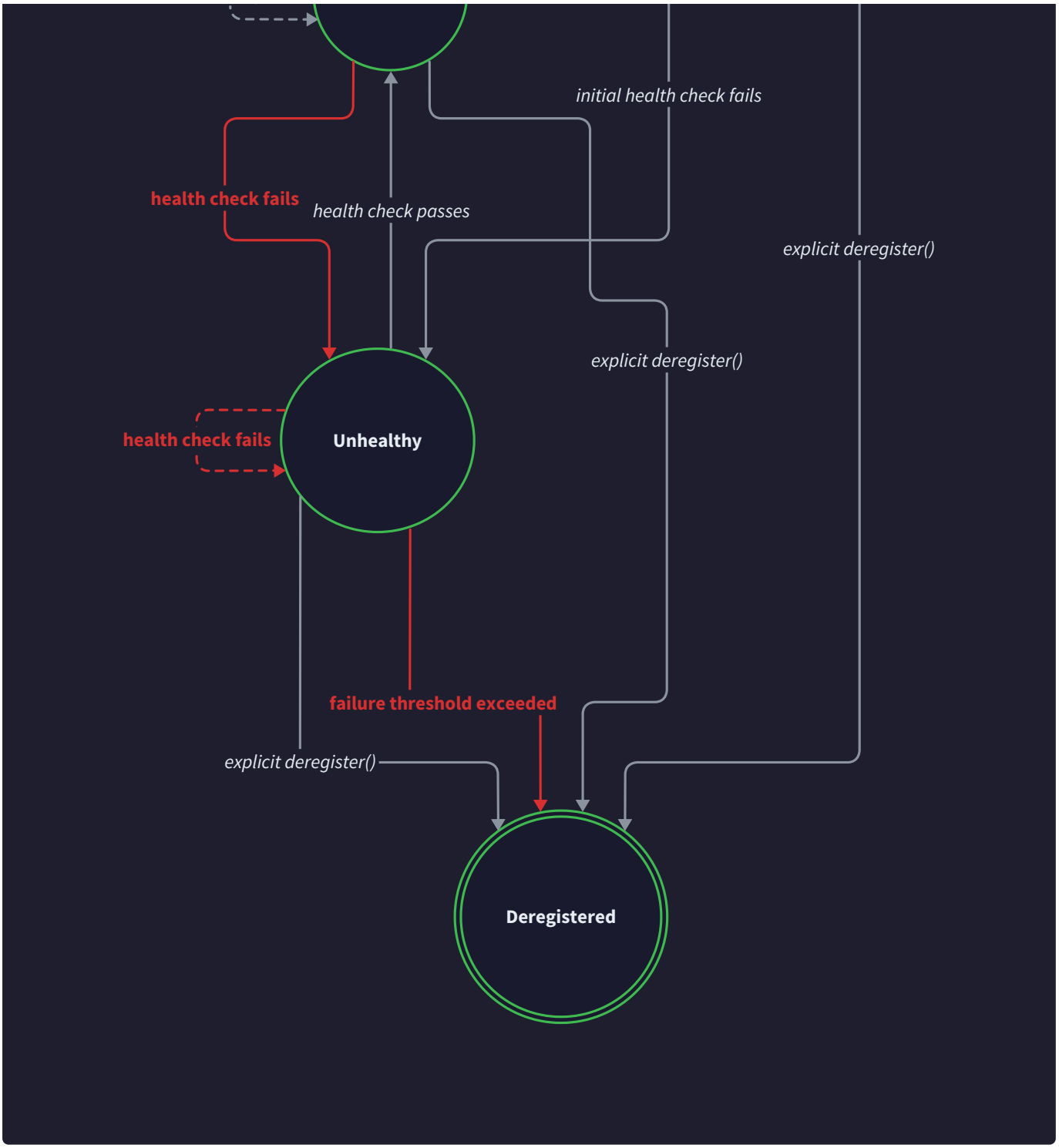
2. **Validate instance still exists**: Verify that the service instance still exists in the registry—it might have been deregistered while the health check was in progress.
3. **Update health status fields**: Apply the new health status information to the registry entry, including the check timestamp, failure count, and status.
4. **Evaluate unhealthy threshold**: If the failure count equals or exceeds `MaxFailures`, transition the instance from healthy to unhealthy status and remove it from active service lookups.
5. **Evaluate recovery threshold**: If an unhealthy instance succeeds a health check, immediately transition it back to healthy status and make it available for service lookups again.
6. **Update registry statistics**: Increment aggregate counters for total checks performed, current healthy/unhealthy counts, and average response times.
7. **Release instance lock**: Free the lock to allow other operations to proceed.
8. **Notify watchers**: If the registry supports change notifications, emit events for health status changes that other components might need to react to.

The registry update algorithm ensures that health status changes are applied atomically and consistently, maintaining registry integrity even under concurrent access.


Health Check Process Flow
HTTP 200 Response?
Yes
No
Reset Failure Count
Increment Failure Count
Failure Count > Threshold?
Yes
No
Mark Service Unhealthy
Schedule Health Check
Update Registry
Check Next Service
Get Services from Registry
Update Status
Any Services to Check?
Query Services
Timer Expired


Service Registry
Yes
No
Wait for Next Check Interval
Send HTTP Health Check


Service Instance State Machine
service starts
Unregistered
register()
Registered
initial health check passes
health check passes
Healthy



Architecture Decision Records

The health checker design involves several critical architectural decisions that significantly impact system behavior and reliability. Each decision represents a trade-off between different operational concerns.

Decision: Health Check Protocol Support

- **Context**: Services might expose health endpoints using different protocols (HTTP, TCP, gRPC) and the health checker needs to determine which protocols to support initially.
- **Options Considered**: HTTP-only, TCP socket checks only, Multi-protocol support from start
- **Decision**: Start with HTTP-only health checks, design interface to support additional protocols later
- **Rationale**: HTTP health checks are the most common pattern in microservices, provide rich status information through response codes and bodies, and have excellent tooling support. TCP checks only verify network connectivity without application-level health. Multi-protocol support adds significant complexity without clear immediate benefit.
- **Consequences**: Enables detailed health status reporting through HTTP response codes and JSON bodies. Requires all services to implement HTTP health endpoints. Creates a clear upgrade path for future protocol support.

Option	Pros	Cons
HTTP-only	Rich status info, standard pattern, simple implementation	Requires HTTP endpoint on all services
TCP-only	Minimal service requirements, very fast checks	No application-level health info
Multi-protocol	Supports diverse service types, maximum flexibility	Complex implementation, harder to maintain

Decision: Failure Threshold and Recovery

- **Context**: The health checker must decide how many consecutive failures indicate a truly unhealthy service versus temporary network issues, and how quickly to restore services that recover.
- **Options Considered**: Single failure removal, Graduated failure thresholds, Configurable per-service thresholds
- **Decision**: Configurable per-service failure thresholds with immediate recovery on first success
- **Rationale**: Different services have different reliability characteristics—a database connection might tolerate fewer failures than a cache service. Immediate recovery ensures that temporarily unhealthy services return to service quickly once they recover.
- **Consequences**: Reduces false positives from network hiccups while allowing fine-tuned reliability requirements per service type. Requires more configuration complexity but provides operational flexibility.

Option	Pros	Cons
Single failure	Simple, fast failure detection	High false positive rate from network blips
Graduated thresholds	Balances reliability with false positives	Fixed behavior doesn't match all service types
Configurable thresholds	Tunable per service needs, operational flexibility	More complex configuration and implementation

Decision: Check Interval Distribution

- **Context**: When checking hundreds of services, simultaneous health checks could create network congestion and overwhelm the registry with concurrent updates.
- **Options Considered**: Fixed interval checks, Random jitter distribution, Adaptive interval based on service health
- **Decision**: Fixed intervals with random startup jitter, plus adaptive intervals for unhealthy services
- **Rationale**: Random jitter prevents thundering herd effects when many services start simultaneously. Adaptive intervals allow faster detection of recovery for unhealthy services without increasing load on healthy services.
- **Consequences**: Smooth resource utilization and better recovery detection. Adds complexity to the scheduling algorithm but significantly improves system scalability.

Option	Pros	Cons
Fixed intervals	Simple implementation, predictable behavior	Can create resource spikes and contention
Random jitter	Prevents thundering herds, smooth resource usage	Less predictable check timing
Adaptive intervals	Optimizes check frequency based on health	Complex scheduling, harder to debug

Decision: Health Check State Persistence

- **Context**: The health checker accumulates failure counts and health history that could be valuable for debugging, but storing this data adds complexity and memory overhead.
- **Options Considered**: In-memory only, Persistent health history, Hybrid approach with recent history
- **Decision**: In-memory state with configurable history retention for recent failures
- **Rationale**: Full persistence adds significant complexity without clear operational benefit for a beginner-level project. In-memory state with limited history provides debugging value while keeping the implementation simple.
- **Consequences**: Enables useful debugging information without storage complexity. Health state is lost on restart, but this is acceptable for a learning project. Creates a clear upgrade path for persistent health metrics.

Option	Pros	Cons
In-memory only	Simple implementation, no storage overhead	No historical data for debugging
Persistent history	Rich debugging info, survives restarts	Complex storage, potential disk usage growth
Hybrid approach	Balance of simplicity and debugging value	Some memory overhead, limited history

Common Pitfalls

Understanding common mistakes in health checker implementation helps avoid reliability issues and operational problems that can affect the entire service discovery system.

⚠️ Pitfall: Network Partition False Positives

A common mistake is treating network timeouts the same as service failures, leading to mass service deregistration during network partitions. When the health checker loses network connectivity to a cluster of services, it shouldn't immediately mark them all as unhealthy—this could cause cascading failures where healthy services become unavailable due to network issues rather than actual service problems.

The issue occurs when the health checker runs from a single location and experiences network connectivity problems. For example, if the health checker is deployed in one availability zone and loses connectivity to services in another zone, it might incorrectly mark all those services as failed even though they're healthy and serving traffic from other clients.

Fix: Implement partition detection by monitoring the failure rate across all services. If more than a threshold percentage (e.g., 50%) of services fail simultaneously, treat this as a potential network partition and either pause health checking or mark the health checker itself as degraded rather than mass-deregistering services.

⚠️ Pitfall: Health Check Timeout Too Short

Setting health check timeouts too aggressively can cause false failures when services experience temporary load spikes or garbage collection pauses. A common mistake is setting timeouts to very small values (e.g., 100ms) that don't account for normal operational variance in service response times.

This becomes particularly problematic for services that might experience periodic background processing, garbage collection in managed languages, or brief CPU contention during normal operation. These temporary slowdowns don't indicate genuine service health problems but will trigger timeout failures if the health check timeout is too strict.

Fix: Set health check timeouts to be 2-3x the typical response time of the service's health endpoint under normal load. Monitor actual health check response times and adjust timeouts based on observed P95 or P99 latencies rather than average response times.

⚠️ Pitfall: Cascading Health Check Failures

A subtle issue occurs when health checks themselves consume significant resources on target services, and increased check frequency during failure scenarios overwhelms already-struggling services. This creates a positive feedback loop where service degradation triggers more frequent health checks, which consume more resources, leading to further degradation.

This problem is especially severe when multiple health checkers monitor the same services (e.g., during rolling deployments or in multi-region setups), or when services implement expensive health check logic that queries databases or external dependencies.

Fix: Implement health check backoff for failing services rather than increasing frequency. Design health endpoints to be lightweight and avoid expensive operations like database queries. Consider implementing health check rate limiting on services to prevent overwhelming them during degradation scenarios.

⚠️ Pitfall: Race Conditions in Health Status Updates

Concurrent health checks and service registration/deregistration can create race conditions where health status updates apply to the wrong service instance or overwrite newer registration information. This often manifests as services that appear to flap between healthy and unhealthy states, or healthy services that get stuck in unhealthy status.

The issue typically occurs when a service deregisters and re-registers quickly (e.g., during a rolling deployment) while health checks for the old instance are still in flight. The delayed health check update might apply to the new instance, incorrectly marking it as failed.

Fix: Use instance-specific identifiers that include registration timestamps or UUIDs to ensure health updates only apply to the correct service instance. Implement proper locking around health status updates and validate that the target instance still exists and matches before applying updates.

⚠️ Pitfall: Health Check Resource Leaks

Failing to properly clean up resources from health check operations can lead to gradual resource exhaustion, especially in systems that monitor large numbers of services. Common leaks include unclosed HTTP connections, goroutines that don't terminate cleanly, or accumulating timeout contexts that don't get canceled.

This problem often goes unnoticed during development with small numbers of services but becomes critical in production environments with hundreds or thousands of service instances being monitored continuously.

Fix: Use connection pooling for HTTP clients, implement proper context cancellation for timeout handling, and ensure goroutines have clear termination conditions. Add resource monitoring to track open connections, goroutines, and memory usage of the health checker component.

Implementation Guidance

The health checker implementation requires careful coordination between background processing, HTTP client management, and registry integration. This section provides the foundational components and structure needed to build a reliable health monitoring system.

Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Client	`net/http` with default client	`net/http` with custom transport and connection pooling
Scheduling	`time.Ticker` with goroutines	Priority queue with `container/heap`
Concurrency	Simple goroutines with channels	Worker pool with bounded concurrency
Configuration	Struct with hardcoded defaults	External config file with validation
Metrics	Simple counters in memory	Structured metrics with `expvar`

Recommended File Structure

```
project-root/
├── internal/health/
│   ├── checker.go              ← main health checker implementation
│   ├── checker_test.go         ← unit tests for health checking logic
│   ├── scheduler.go            ← check scheduling and timing logic
│   ├── client.go               ← HTTP client utilities for health checks
│   └── metrics.go              ← health check metrics and statistics
├── internal/registry/
│   └── registry.go             ← registry core (from previous section)
└── pkg/types/
    └── health.go               ← health-related type definitions
```

This structure separates health checking concerns into focused modules while maintaining clear boundaries with the registry component.

Health Check Client Utilities

```go
package health

import (
    "context"
    "net/http"
    "time"
)

// HealthCheckClient provides HTTP utilities for performing health checks
// with proper timeout handling and connection management.
type HealthCheckClient struct {
    client *http.Client
    timeout time.Duration
}

// NewHealthCheckClient creates a configured HTTP client optimized for health checks.
// Uses connection pooling and appropriate timeouts to handle high-frequency checking.
func NewHealthCheckClient(timeout time.Duration) *HealthCheckClient {
    transport := &http.Transport{
        MaxIdleConns:        100,
        MaxIdleConnsPerHost: 10,
        IdleConnTimeout:     30 * time.Second,
        DisableKeepAlives:   false,
    }

    return &HealthCheckClient{
        client: &http.Client{
```

```
            Transport: transport,
            Timeout:   timeout,
        },
        timeout: timeout,
    }
}

// CheckHealth performs a single health check against the specified endpoint.
// Returns the health status and any error encountered during the check.
func (hc *HealthCheckClient) CheckHealth(ctx context.Context, endpoint string)
(HealthStatus, error) {
    // TODO: Create HTTP GET request to health endpoint
    // TODO: Execute request with context timeout
    // TODO: Interpret HTTP status code (200-299 = healthy, others = unhealthy)
    // TODO: Measure response time for performance metrics
    // TODO: Parse response body for additional health details if present
    // TODO: Return structured HealthStatus with all collected information
    return HealthStatus{}, nil
}

// Close cleans up HTTP client resources including connection pools.
func (hc *HealthCheckClient) Close() error {
    // TODO: Close idle connections in transport
    // TODO: Clean up any background resources
    return nil
}
```

Health Check Scheduler

```go
package health

import (
    "context"
    "sync"
    "time"
)

// HealthCheckScheduler manages the timing and coordination of health checks
// across all registered service instances.
type HealthCheckScheduler struct {
    registry     *ServiceRegistry
    client       *HealthCheckClient
    checkInterval time.Duration
    maxFailures   int

    // Internal scheduling state
    scheduledChecks map[string]*ScheduledCheck
    scheduleTimer   *time.Timer
    stopChan        chan struct{}
    wg              sync.WaitGroup
    mutex           sync.RWMutex
}

// ScheduledCheck represents a pending health check for a service instance.
type ScheduledCheck struct {
    InstanceID   string
```

```
    ServiceName  string
    NextCheck    time.Time
    Interval     time.Duration
    FailureCount int
}

// NewHealthCheckScheduler creates a scheduler with the specified configuration.
func NewHealthCheckScheduler(registry *ServiceRegistry, client *HealthCheckClient, config
Config) *HealthCheckScheduler {
    return &HealthCheckScheduler{
        registry:        registry,
        client:          client,
        checkInterval:  config.HealthCheckInterval,
        maxFailures:    config.MaxFailures,
        scheduledChecks: make(map[string]*ScheduledCheck),
        stopChan:       make(chan struct{}),
    }
}

// Start begins the health checking background process.
func (s *HealthCheckScheduler) Start(ctx context.Context) error {
    // TODO: Initialize scheduled checks for all currently registered services
    // TODO: Start background goroutine for processing scheduled checks
    // TODO: Set up timer for next scheduled check
    // TODO: Handle service registration/deregistration events
    return nil
}

// Stop gracefully shuts down health checking.
```

```go
func (s *HealthCheckScheduler) Stop() error {

    // TODO: Signal stop to background goroutines

    // TODO: Wait for in-flight health checks to complete

    // TODO: Clean up scheduled check state

    return nil

}


// ScheduleHealthCheck adds a new service instance to the health checking schedule.

func (s *HealthCheckScheduler) ScheduleHealthCheck(instanceID string, instance
ServiceInstance) {

    // TODO: Calculate initial check time with random jitter

    // TODO: Create ScheduledCheck entry for the instance

    // TODO: Add to scheduledChecks map

    // TODO: Update next check timer if this check is sooner

}


// UnscheduleHealthCheck removes a service instance from health checking.

func (s *HealthCheckScheduler) UnscheduleHealthCheck(instanceID string) {

    // TODO: Remove from scheduledChecks map

    // TODO: Recalculate next check timer if needed

}


// processScheduledChecks is the main background loop for executing health checks.

func (s *HealthCheckScheduler) processScheduledChecks(ctx context.Context) {

    // TODO: Run continuous loop checking for due health checks

    // TODO: Execute health checks for instances whose NextCheck time has arrived

    // TODO: Update failure counts and registry health status based on results

    // TODO: Reschedule next check based on success/failure and configured intervals

    // TODO: Handle context cancellation for graceful shutdown
```

```
}


// executeHealthCheck performs a single health check and updates registry state.
func (s *HealthCheckScheduler) executeHealthCheck(ctx context.Context, check
*ScheduledCheck) {
    // TODO: Get current service instance details from registry
    // TODO: Construct health endpoint URL from instance host, port, and health endpoint
path
    // TODO: Use HealthCheckClient to perform the actual HTTP health check
    // TODO: Interpret results and update failure count accordingly
    // TODO: Call registry.UpdateHealth() with new health status
    // TODO: Calculate next check time based on current interval and jitter
}
```

Core Health Checker Implementation

```go
package health

import (
    "context"
    "sync"
    "time"
)

// HealthChecker is the main component that coordinates health checking
// across all registered service instances.
type HealthChecker struct {
    scheduler *HealthCheckScheduler
    client    *HealthCheckClient
    config    Config

    // Health statistics and monitoring
    stats     HealthStats
    statsMu   sync.RWMutex

    // Lifecycle management
    running   bool
    runningMu sync.RWMutex
}

// HealthStats provides aggregate statistics about health checking operations.
type HealthStats struct {
    TotalChecksPerformed int64
```

```
    TotalFailures       int64
    TotalSuccesses      int64
    AverageResponseTime time.Duration
    LastCheckTime       time.Time
    CurrentlyHealthy    int
    CurrentlyUnhealthy  int
}

// NewHealthChecker creates a new health checker with the specified configuration.
func NewHealthChecker(registry *ServiceRegistry, config Config) (*HealthChecker, error) {
    // TODO: Create HealthCheckClient with configured timeout
    // TODO: Create HealthCheckScheduler with registry and client
    // TODO: Initialize HealthStats structure
    // TODO: Validate configuration parameters
    return &HealthChecker{}, nil
}

// Start begins health checking for all registered services.
func (hc *HealthChecker) Start(ctx context.Context) error {
    // TODO: Check if already running and return error if so
    // TODO: Mark as running under lock
    // TODO: Start the health check scheduler
    // TODO: Set up registry callbacks for service registration/deregistration events
    return nil
}

// Stop gracefully shuts down health checking.
func (hc *HealthChecker) Stop() error {
```

```
    // TODO: Check if not running and return early
    // TODO: Mark as not running under lock
    // TODO: Stop the scheduler and wait for completion
    // TODO: Close HTTP client resources
    // TODO: Clean up any background resources
    return nil
}

// CheckServiceHealth performs an immediate health check on a specific service instance.
// This is primarily used for testing and manual verification.
func (hc *HealthChecker) CheckServiceHealth(instance ServiceInstance) (HealthStatus, error)
{
    // TODO: Construct health endpoint URL from instance details
    // TODO: Use HTTP client to perform health check with timeout
    // TODO: Interpret HTTP response and construct HealthStatus
    // TODO: Update internal statistics with check results
    // TODO: Return health status and any errors encountered
    return HealthStatus{}, nil
}

// GetHealthStats returns current aggregate health checking statistics.
func (hc *HealthChecker) GetHealthStats() HealthStats {
    // TODO: Acquire read lock on stats
    // TODO: Return copy of current statistics
    return HealthStats{}
}

// UpdateCheckInterval modifies the health check frequency for a specific service.
```

```
func (hc *HealthChecker) UpdateCheckInterval(serviceName string, interval time.Duration)
error {

    // TODO: Validate interval is within reasonable bounds

    // TODO: Update scheduler configuration for the specified service

    // TODO: Reschedule pending checks for affected service instances

    return nil

}


// updateStats increments internal statistics based on health check results.

func (hc *HealthChecker) updateStats(success bool, responseTime time.Duration) {

    // TODO: Acquire write lock on stats

    // TODO: Increment appropriate counters (success/failure)

    // TODO: Update running average of response times

    // TODO: Update last check timestamp

    // TODO: Release lock

}
```

Milestone Checkpoint

After implementing the health checker component, verify the following behavior:

Unit Test Verification:

```
go test ./internal/health/... -v
```
BASH

Expected output should show all health checking functions working correctly with mock services.

Integration Test Verification:

1. **Start a test service** with a `/health` endpoint that returns HTTP 200
2. **Register the service** with your registry
3. **Start the health checker** and verify it begins checking the service
4. **Stop the test service** and verify the health checker marks it as unhealthy after the configured failure threshold
5. **Restart the test service** and verify it gets marked healthy again immediately

Manual Verification Steps:

1. Check health statistics show increasing check counts: `curl http://localhost:8080/stats`
2. Verify healthy services appear in lookup results: `curl http://localhost:8080/services/test-service`
3. Verify unhealthy services disappear from lookup results after failure threshold
4. Verify health checker logs show periodic check activity and status changes

Signs Something Is Wrong:

Symptom	Likely Cause	Check This
Health checks never execute	Scheduler not starting properly	Check for goroutine startup errors and timer initialization
All services marked unhealthy immediately	Health check timeout too short	Verify health endpoint response times vs configured timeout
Services flapping between healthy/unhealthy	Race condition in status updates	Check for proper locking around health status modifications
Memory usage growing continuously	Resource leaks in HTTP client	Verify HTTP connections are properly closed and pooled
Health checks hang indefinitely	Missing context timeouts	Ensure all HTTP requests use context with deadline

HTTP API Layer

> **Milestone(s):** Milestone 3 (HTTP API) - This section covers the REST API that exposes service registry operations over HTTP, building on the registry core (Milestone 1) and health checker (Milestone 2) to provide external access to service discovery functionality.

Mental Model: The Information Desk

Understanding the HTTP API layer becomes intuitive when you think of it as an **information desk** at a large corporate building or conference center. Just as visitors approach the information desk to find departments, get directions, or register for events, services and clients interact with our HTTP API to register themselves, discover other services, and check system health.

The information desk receptionist doesn't actually store the building directory or manage the departments themselves - they have access to the building's central directory system (our registry core) and can check with security about which departments are currently operational (our health checker). The receptionist's job is to

translate visitor requests into the right format, query the appropriate systems, and present the information in a way visitors can understand and use.

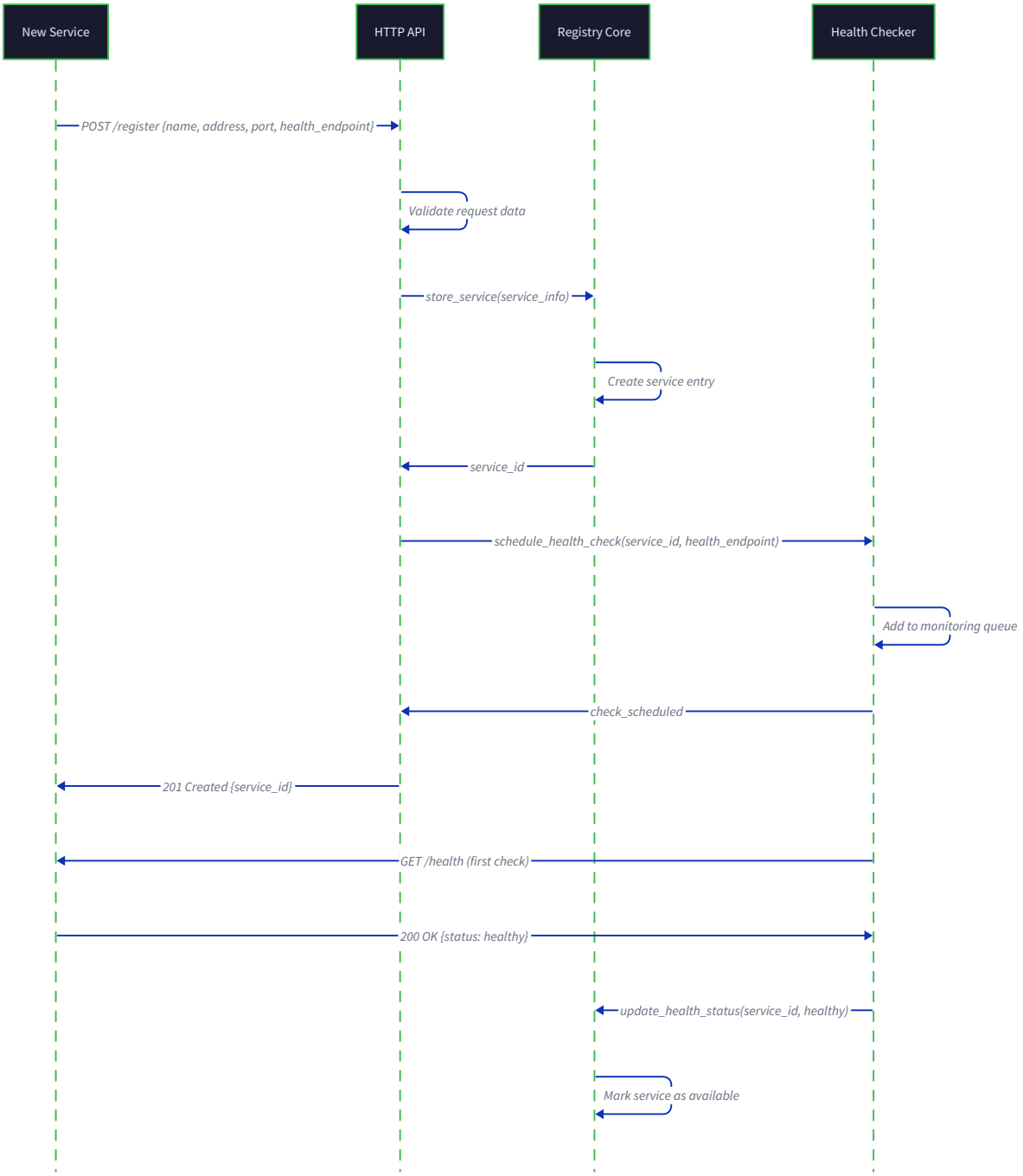
Similarly, our HTTP API layer serves as the **interface translator** between external clients and our internal registry system. It accepts HTTP requests in standardized formats, validates the input (just as a receptionist might verify visitor credentials), translates the requests into internal operations, executes them against the registry core, and formats the responses back into HTTP-friendly JSON.

The information desk analogy also highlights the **stateless nature** of our API - each visitor interaction is independent. The receptionist doesn't remember previous conversations with visitors, but can quickly look up any information needed from the central systems. This stateless design ensures our API can handle many concurrent requests efficiently and scale horizontally by adding more API servers behind a load balancer.

Just as an information desk provides standardized procedures for common requests (registration forms, visitor badges, direction cards), our HTTP API provides **consistent REST endpoints** with predictable request/response formats. This consistency allows different types of clients - whether they're microservices registering themselves, load balancers looking up healthy instances, or monitoring tools checking system health - to interact with the service discovery system in a uniform way.

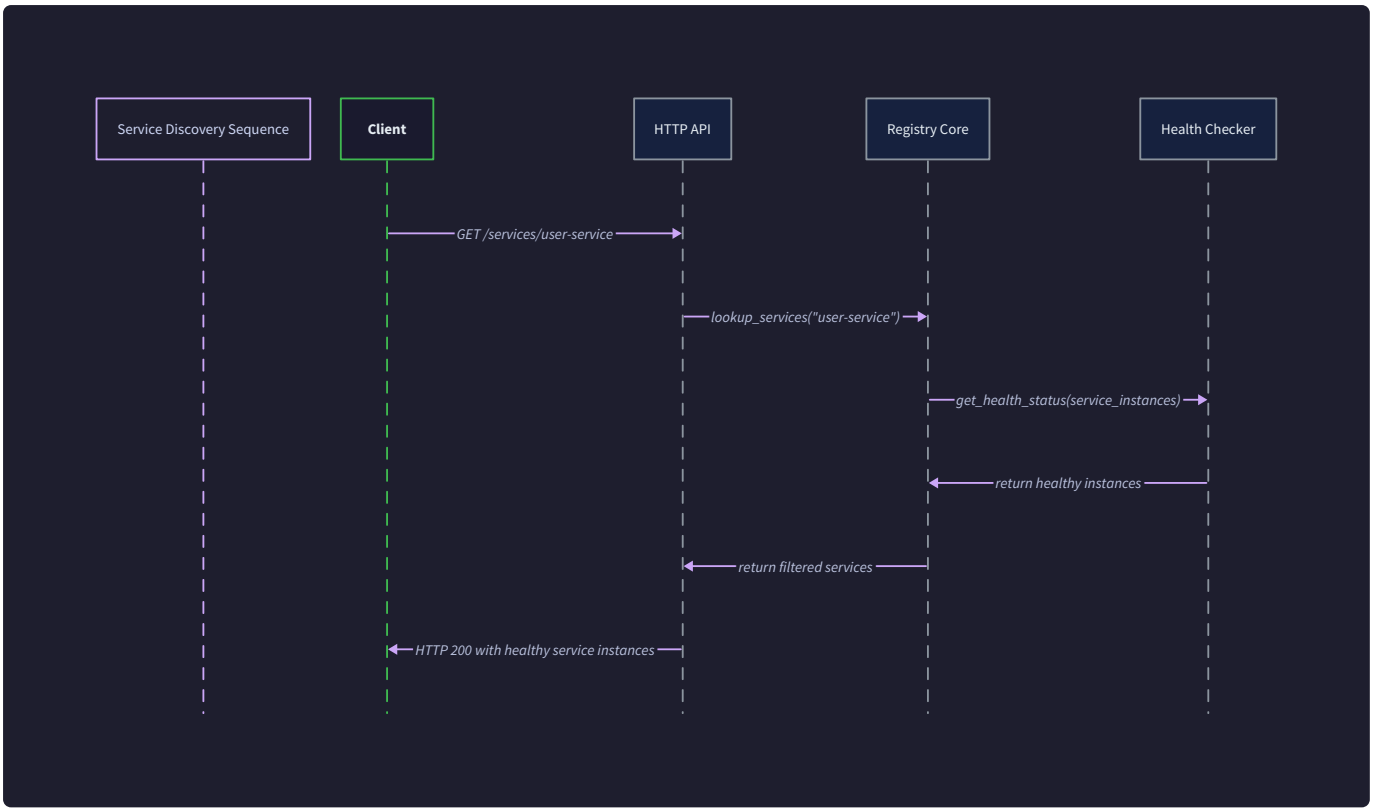
API Endpoints

The HTTP API exposes service registry operations through a carefully designed REST interface. Each endpoint serves a specific purpose in the service discovery workflow, from initial registration through ongoing health monitoring to final deregistration.



The API design follows REST principles with clear resource hierarchies and appropriate HTTP methods. The base path `/services` represents the collection of all registered services, while specific operations are accessed through sub-paths and different HTTP methods.

Endpoint	Method	Purpose	Request Body	Response
`/services`	POST	Register new service instance	`ServiceInstance`	`SuccessResponse` with instance ID
`/services`	GET	List all registered services	None	`ServiceList` array
`/services/{name}`	GET	Get healthy instances of service	None	`ServiceList` for specific service
`/services/{name}/{id}`	DELETE	Deregister specific instance	None	`SuccessResponse`
`/health`	GET	Get system health overview	None	`RegistryStats`
`/health/{name}`	GET	Get health status for service	None	Service health details
`/health/{name}/{id}`	GET	Get health status for instance	None	`HealthStatus`



Service Registration Endpoint

The POST `/services` endpoint accepts new service registrations and returns a unique instance identifier. The request body must contain a complete `ServiceInstance` object with all required fields validated before registration proceeds.

Request Format:

Field	Type	Required	Description
Name	string	Yes	Service name for lookup operations
Host	string	Yes	IP address or hostname where service runs
Port	int	Yes	Port number where service accepts connections
Tags	[]string	No	Metadata tags for filtering and categorization
HealthEndpoint	string	No	HTTP path for health checks (defaults to "/health")

Response Format:

Field	Type	Description
success	bool	Always true for successful registrations
data	object	Contains instanceID and registration timestamp
message	string	Human-readable success message

The registration endpoint performs several validation steps before accepting a service instance. It verifies that required fields are present and properly formatted, checks that the host:port combination is valid, and ensures the health endpoint path starts with a forward slash if provided.

Service Lookup Endpoints

The GET `/services` endpoint returns a complete listing of all registered services, while GET `/services/{name}` returns only healthy instances of a specific service. Both endpoints filter results to exclude instances marked as unhealthy by the health checker.

Service List Response Format:

Field	Type	Description
ServiceName	string	Name of the service (empty for all-services endpoint)
Instances	[]ServiceInstance	Array of healthy service instances
TotalCount	int	Total registered instances (including unhealthy)
HealthyCount	int	Number of instances currently healthy

The lookup operations are optimized for read performance since they represent the most frequent API operations. The registry core maintains pre-filtered lists of healthy instances to avoid computing health status during each lookup request.

Service Deregistration Endpoint

The DELETE `/services/{name}/{id}` endpoint removes a specific service instance from the registry. Both the service name and instance ID must match for the deregistration to proceed, providing protection against accidental removals.

Deregistration Response:

Field	Type	Description
success	bool	True if instance was found and removed
message	string	Confirmation message with instance details

Deregistration immediately removes the instance from active service lists and cancels any scheduled health checks. The operation is idempotent - attempting to deregister an already-removed instance returns success with an appropriate message.

Health Status Endpoints

The health endpoints provide visibility into the current state of the service discovery system. The GET `/health` endpoint returns overall system statistics, while service-specific endpoints provide detailed health information for monitoring and debugging purposes.

System Health Response (`RegistryStats`):

Field	Type	Description
TotalServices	int	Number of distinct service names registered
TotalInstances	int	Total service instances across all services
HealthyInstances	int	Number of instances passing health checks
HealthBreakdown	map[string]int	Health status counts by service name
LastCleanup	time.Time	Timestamp of last TTL cleanup operation

Instance Health Response (`HealthStatus`):

Field	Type	Description
Status	string	"healthy", "unhealthy", or "unknown"
LastCheck	time.Time	Timestamp of most recent health check
FailureCount	int	Consecutive failed health checks
ResponseTime	time.Duration	Response time of last successful check
LastError	string	Error message from most recent failed check

Error Response Format

All endpoints use a standardized error response format to provide consistent error handling for clients. Error responses include both machine-readable error codes and human-readable messages.

Error Response Format (`ErrorResponse`):

Field	Type	Description
error	string	Human-readable error message
error_code	string	Machine-readable error code for programmatic handling
details	map[string]interface{}	Additional context about the error

Common error codes include `VALIDATION_ERROR` for invalid request data, `NOT_FOUND` for non-existent services or instances, `INTERNAL_ERROR` for system failures, and `RATE_LIMITED` for excessive request rates.

Architecture Decision Records

The HTTP API design involves several critical decisions that affect usability, performance, and maintainability. Each decision represents a trade-off between competing concerns and requirements.

> **Decision: REST API Design Pattern**
>
> - **Context**: Need to choose API design pattern for exposing registry operations over HTTP
> - **Options Considered**: REST with JSON, GraphQL, gRPC with HTTP/2, Custom HTTP protocol
> - **Decision**: REST API with JSON request/response bodies
> - **Rationale**: REST provides the best balance of simplicity, tooling support, and developer familiarity. JSON is universally supported and human-readable. GraphQL adds unnecessary complexity for our straightforward CRUD operations. gRPC requires code generation and is less accessible for debugging and testing.
> - **Consequences**: Enables easy integration with any HTTP client, supports standard tooling (curl, Postman), but may have slightly higher overhead than binary protocols

Option	Pros	Cons
REST + JSON	Universal client support, human-readable, extensive tooling	Larger payload size, text parsing overhead
GraphQL	Flexible queries, single endpoint	Complex for simple operations, learning curve
gRPC + HTTP/2	Type safety, performance, streaming	Code generation required, debugging complexity

Decision: Resource Hierarchy Structure

- **Context**: Need to organize API endpoints in a logical, RESTful manner
- **Options Considered**: Flat namespace (/register, /lookup), Resource-based (/services, /health), Verb-based (/api/register-service)
- **Decision**: Resource-based hierarchy with `/services` as primary collection
- **Rationale**: Resource-based design follows REST principles, provides intuitive URL structure, and scales well as we add new resource types. HTTP methods (GET, POST, DELETE) clearly indicate operations.
- **Consequences**: Clean, predictable URLs that developers can guess, but requires understanding of REST conventions

Option	Pros	Cons
Resource-based	REST compliance, intuitive, scalable	Requires REST knowledge
Flat namespace	Simple, explicit	Doesn't scale, non-standard
Verb-based	Clear intent	Violates REST principles, verbose

Decision: Error Response Standardization

- **Context**: Need consistent error handling across all API endpoints
- **Options Considered**: HTTP status codes only, Custom error format, RFC 7807 Problem Details, Simple JSON errors
- **Decision**: Standardized JSON error format with error codes and details
- **Rationale**: Provides both human-readable messages and machine-readable error codes for programmatic handling. Includes details object for additional context. More accessible than RFC 7807 while still being structured.
- **Consequences**: Consistent error handling for clients, but requires discipline to use the format consistently across all endpoints

Option	Pros	Cons
Status codes only	Simple, HTTP-native	Limited error information
Custom JSON format	Flexible, informative	Must document format
RFC 7807	Standardized	Complex, less familiar

Decision: Instance Identity in URLs

- **Context**: Need to identify specific service instances for deregistration operations
- **Options Considered**: Auto-generated UUIDs, Host:port combinations, Sequential IDs, Client-provided IDs
- **Decision**: Auto-generated UUIDs returned during registration
- **Rationale**: UUIDs are globally unique, prevent collisions, and don't expose system internals. Host:port combinations can collide if services restart quickly. Sequential IDs expose registration order and volume.
- **Consequences**: Clients must store the returned instance ID for later deregistration, but we avoid all collision scenarios

Option	Pros	Cons
UUIDs	Globally unique, secure	Must be stored by client
Host:port	Intuitive, self-contained	Collision risk on restart
Sequential IDs	Simple, compact	Exposes system internals

Decision: Health Check Integration

- **Context**: Need to decide how health status affects API responses
- **Options Considered**: Always return all instances, Filter unhealthy automatically, Provide health flags, Separate health endpoints
- **Decision**: Filter unhealthy instances automatically in lookup operations, provide separate health endpoints for monitoring
- **Rationale**: Service lookup should return only usable instances by default. Monitoring tools need access to health details, but service clients want pre-filtered results for simplicity.
- **Consequences**: Lookup operations are immediately usable, but monitoring requires separate API calls

Option	Pros	Cons
Auto-filter unhealthy	Simple for clients	Less visibility
Include health flags	Full visibility	More complex responses
Separate health endpoints	Clean separation	Multiple API calls needed

Common Pitfalls

HTTP API implementations often suffer from common mistakes that affect reliability, security, and maintainability. Understanding these pitfalls helps avoid production issues and provides better developer experience.

⚠️ Pitfall: Insufficient Input Validation

Many implementations perform minimal validation on incoming requests, leading to data corruption, system crashes, or security vulnerabilities. For example, accepting empty service names allows invalid registrations that break lookup operations, while missing host validation can result in unreachable service instances being registered.

Why it's wrong: Invalid data in the registry corrupts the service discovery system. Services trying to connect to malformed addresses will fail, and empty names make services undiscoverable.

How to fix: Implement comprehensive validation for every field. Service names should match naming conventions (alphanumeric plus hyphens), hosts should be valid IP addresses or hostnames, ports should be in valid ranges (1-65535), and health endpoints should be proper HTTP paths starting with '/'.

⚠️ Pitfall: Inconsistent Error Responses

Different endpoints returning different error formats makes client error handling complex and unreliable. Some endpoints might return plain text errors, others JSON objects with different field names, leading to brittle client code that can't handle errors consistently.

Why it's wrong: Clients need to implement multiple error parsing strategies, increasing complexity and bug potential. Inconsistent errors also make debugging and logging more difficult.

How to fix: Use the standardized `ErrorResponse` format for all error conditions. Create helper functions like `WriteErrorResponse` that enforce consistent formatting. Document all possible error codes and ensure they're used consistently across endpoints.

⚠️ Pitfall: Race Conditions with Concurrent Requests

Multiple simultaneous requests can create race conditions, especially during registration and deregistration operations. For example, two services registering with the same name simultaneously might interfere with each other, or a deregistration might happen while a health check is updating the same instance.

Why it's wrong: Race conditions can corrupt registry state, cause instances to disappear unexpectedly, or create inconsistent health status. These issues are often intermittent and difficult to debug.

How to fix: Ensure the registry core handles concurrent access properly with appropriate locking. Design API operations to be atomic - either they succeed completely or fail completely with no partial state changes.

⚠️ Pitfall: Missing Content-Type Validation

Accepting requests without validating the Content-Type header can lead to parsing errors or security issues. For example, accepting `text/plain` data when expecting JSON can cause unmarshaling failures that crash

the handler.

Why it's wrong: Malformed requests can crash handlers, and missing content-type validation can be exploited for injection attacks or DOS scenarios.

How to fix: Validate that POST requests include `Content-Type: application/json` header. Reject requests with missing or incorrect content types using HTTP 400 Bad Request status code.

⚠️ Pitfall: Inadequate Error Status Codes

Using generic HTTP 500 errors for all failures provides no guidance to clients about whether requests should be retried, what went wrong, or how to fix problems. This makes debugging difficult and prevents intelligent client retry logic.

Why it's wrong: Clients can't distinguish between retryable errors (network issues) and permanent errors (validation failures), leading to unnecessary retries or missing legitimate retry opportunities.

How to fix: Use appropriate HTTP status codes: 400 for validation errors, 404 for not found, 409 for conflicts, 429 for rate limiting, 500 only for unexpected internal errors. Include detailed error information in the response body.

⚠️ Pitfall: Blocking Operations in Request Handlers

Performing long-running operations like health checks directly in HTTP request handlers can cause request timeouts and poor user experience. For example, triggering an immediate health check during registration blocks the response until the check completes.

Why it's wrong: HTTP clients expect reasonably fast responses. Blocking operations can cause timeouts, especially when network latency to health endpoints is high.

How to fix: Keep request handlers fast by deferring expensive operations to background processes. Registration should return immediately after storing the instance, with health checking happening asynchronously.

Implementation Guidance

The HTTP API layer requires careful integration of HTTP server components, request routing, input validation, and response formatting. The implementation should prioritize reliability, performance, and ease of maintenance while providing a developer-friendly interface.

Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Server	`net/http` standard library	`gin-gonic/gin` or `gorilla/mux` framework
JSON Handling	`encoding/json` standard library	`json-iterator/go` for performance
Input Validation	Manual validation with helper functions	`go-playground/validator` struct tags
Logging	`log` standard library	`logrus` or `zap` structured logging
Middleware	Custom middleware functions	Framework-provided middleware stack

The standard library approach provides excellent learning value and minimal dependencies, while the framework approach offers more convenience and advanced features for production use.

Recommended File Structure

```
internal/api/
  server.go              ← HTTP server setup and configuration
  handlers.go            ← Request handler functions
  middleware.go          ← Authentication, logging, CORS middleware
  validation.go          ← Input validation helper functions
  responses.go           ← Response formatting utilities
  routes.go              ← Route definitions and URL patterns
  server_test.go         ← Integration tests for HTTP endpoints
  handlers_test.go       ← Unit tests for individual handlers
```

This structure separates concerns clearly: server setup, request handling, validation, and response formatting each have dedicated files. The separation makes the codebase easier to navigate and test.

HTTP Server Infrastructure

The following complete HTTP server infrastructure handles common concerns like JSON marshaling, error responses, and request logging, allowing you to focus on the core handler logic.

```go
package api


import (

    "context"

    "encoding/json"

    "fmt"

    "log"

    "net/http"

    "time"

    "strings"

)


// Server wraps the HTTP server with registry dependencies

type Server struct {

    registry     *ServiceRegistry

    healthChecker *HealthChecker

    config       *Config

    httpServer   *http.Server

}


// NewServer creates a configured HTTP server with all routes

func NewServer(registry *ServiceRegistry, healthChecker *HealthChecker, config *Config)
*Server {

    server := &Server{

        registry:      registry,

        healthChecker: healthChecker,

        config:       config,

    }
```

```
    mux := http.NewServeMux()
    server.setupRoutes(mux)

    server.httpServer = &http.Server{
        Addr:         fmt.Sprintf("%s:%d", config.Host, config.Port),
        Handler:      server.loggingMiddleware(server.corsMiddleware(mux)),
        ReadTimeout:  10 * time.Second,
        WriteTimeout: 10 * time.Second,
    }

    return server
}

// Start begins serving HTTP requests
func (s *Server) Start() error {
    log.Printf("HTTP API server starting on %s", s.httpServer.Addr)
    return s.httpServer.ListenAndServe()
}

// Stop gracefully shuts down the HTTP server
func (s *Server) Stop(ctx context.Context) error {
    log.Println("HTTP API server stopping...")
    return s.httpServer.Shutdown(ctx)
}

// setupRoutes configures all API endpoints
func (s *Server) setupRoutes(mux *http.ServeMux) {
    mux.HandleFunc("/services", s.handleServices)
```

```
    mux.HandleFunc("/services/", s.handleServicesWithPath)
    mux.HandleFunc("/health", s.handleSystemHealth)
    mux.HandleFunc("/health/", s.handleServiceHealth)
}

// Middleware for request logging
func (s *Server) loggingMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        start := time.Now()
        next.ServeHTTP(w, r)
        log.Printf("%s %s %v", r.Method, r.URL.Path, time.Since(start))
    })
}

// Middleware for CORS headers
func (s *Server) corsMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Access-Control-Allow-Origin", "*")
        w.Header().Set("Access-Control-Allow-Methods", "GET, POST, DELETE, OPTIONS")
        w.Header().Set("Access-Control-Allow-Headers", "Content-Type")

        if r.Method == "OPTIONS" {
            w.WriteHeader(http.StatusOK)
            return
        }

        next.ServeHTTP(w, r)
    })
```

```
}
```

Response Formatting Utilities

These utilities ensure consistent response formatting across all endpoints and simplify error handling in request handlers.

GO

```go
// WriteJSONResponse sends a JSON response with the specified status code
func WriteJSONResponse(w http.ResponseWriter, statusCode int, data interface{}) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(statusCode)

    if err := json.NewEncoder(w).Encode(data); err != nil {
        log.Printf("Error encoding JSON response: %v", err)
        // Response already started, can't change status code
        return
    }
}

// WriteErrorResponse sends a standardized error response
func WriteErrorResponse(w http.ResponseWriter, statusCode int, message string, errorCode string) {
    response := ErrorResponse{
        error:      message,
        error_code: errorCode,
        details:    make(map[string]interface{}),
    }
    WriteJSONResponse(w, statusCode, response)
}

// WriteSuccessResponse sends a standardized success response
func WriteSuccessResponse(w http.ResponseWriter, data interface{}, message string) {
    response := SuccessResponse{
        success: true,
        data:    data,
```

```
        message: message,
    }
    WriteJSONResponse(w, http.StatusOK, response)
}

// WriteValidationError sends a 400 error with validation details
func WriteValidationError(w http.ResponseWriter, field string, problem string) {
    response := ErrorResponse{
        error:      fmt.Sprintf("Validation failed for field '%s': %s", field, problem),
        error_code: "VALIDATION_ERROR",
        details: map[string]interface{}{
            "field": field,
            "problem": problem,
        },
    }
    WriteJSONResponse(w, http.StatusBadRequest, response)
}
```

Input Validation Functions

Complete validation functions that check all service instance fields according to the requirements established in previous sections.

```go
// ValidateServiceInstance performs comprehensive validation of service instance data

func ValidateServiceInstance(instance ServiceInstance) error {

    if strings.TrimSpace(instance.Name) == "" {

        return fmt.Errorf("service name cannot be empty")

    }



    if !isValidServiceName(instance.Name) {

        return fmt.Errorf("service name must contain only alphanumeric characters and
hyphens")

    }



    if strings.TrimSpace(instance.Host) == "" {

        return fmt.Errorf("host cannot be empty")

    }



    if !isValidHost(instance.Host) {

        return fmt.Errorf("host must be a valid IP address or hostname")

    }



    if instance.Port <= 0 || instance.Port > 65535 {

        return fmt.Errorf("port must be between 1 and 65535")

    }



    if instance.HealthEndpoint != "" && !strings.HasPrefix(instance.HealthEndpoint, "/") {

        return fmt.Errorf("health endpoint must start with '/'")

    }
```

```
    return nil
}

// isValidServiceName checks service name format
func isValidServiceName(name string) bool {
    if len(name) == 0 || len(name) > 63 {
        return false
    }

    for _, char := range name {
        if !((char >= 'a' && char <= 'z') ||
             (char >= 'A' && char <= 'Z') ||
             (char >= '0' && char <= '9') ||
             char == '-') {
            return false
        }
    }

    return true
}

// isValidHost checks if host is valid IP or hostname
func isValidHost(host string) bool {
    // Simple validation - in production, use net.ParseIP and more sophisticated hostname validation
    return len(host) > 0 && len(host) < 255 && !strings.Contains(host, " ")
}
```

Core Handler Skeletons

The following handler skeletons provide the structure for implementing each endpoint while leaving the core logic for you to complete based on the algorithm descriptions from previous sections.

GO

```go
// handleServices routes between POST (register) and GET (list all)
func (s *Server) handleServices(w http.ResponseWriter, r *http.Request) {
    switch r.Method {
    case http.MethodPost:
        s.handleServiceRegistration(w, r)
    case http.MethodGet:
        s.handleListAllServices(w, r)
    default:
        WriteErrorResponse(w, http.StatusMethodNotAllowed, "Method not allowed",
"METHOD_NOT_ALLOWED")
    }
}

// handleServiceRegistration processes POST /services requests
func (s *Server) handleServiceRegistration(w http.ResponseWriter, r *http.Request) {
    // TODO 1: Validate Content-Type header is application/json
    // TODO 2: Parse JSON request body into ServiceInstance struct
    // TODO 3: Validate the service instance using ValidateServiceInstance()
    // TODO 4: Call s.registry.Register() to register the instance
    // TODO 5: Return success response with instance ID and registration time
    // Hint: Use json.NewDecoder(r.Body).Decode() for parsing
    // Hint: Handle parsing errors with appropriate HTTP status codes
}

// handleListAllServices processes GET /services requests
func (s *Server) handleListAllServices(w http.ResponseWriter, r *http.Request) {
    // TODO 1: Call s.registry.GetAllServices() to get all registered services
    // TODO 2: Filter out unhealthy instances from each service's instance list
```

```
    // TODO 3: Build response with service names, healthy instances, and counts
    // TODO 4: Return JSON response with service list
    // Hint: Create a slice of ServiceList objects, one per service name
    // Hint: Use the HealthyCount field to show filtering results
}

// handleServicesWithPath routes requests to /services/{name} and /services/{name}/{id}
func (s *Server) handleServicesWithPath(w http.ResponseWriter, r *http.Request) {
    pathParts := strings.Split(strings.Trim(r.URL.Path, "/"), "/")
    if len(pathParts) < 2 {
        WriteErrorResponse(w, http.StatusNotFound, "Invalid path", "INVALID_PATH")
        return
    }

    serviceName := pathParts[1]

    if len(pathParts) == 2 {
        // /services/{name}
        if r.Method == http.MethodGet {
            s.handleServiceLookup(w, r, serviceName)
        } else {
            WriteErrorResponse(w, http.StatusMethodNotAllowed, "Method not allowed",
"METHOD_NOT_ALLOWED")
        }
    } else if len(pathParts) == 3 {
        // /services/{name}/{id}
        instanceID := pathParts[2]
        if r.Method == http.MethodDelete {
```

```
            s.handleServiceDeregistration(w, r, serviceName, instanceID)

        } else {

            WriteErrorResponse(w, http.StatusMethodNotAllowed, "Method not allowed",
"METHOD_NOT_ALLOWED")

        }

    } else {

        WriteErrorResponse(w, http.StatusNotFound, "Invalid path", "INVALID_PATH")

    }

}


// handleServiceLookup processes GET /services/{name} requests

func (s *Server) handleServiceLookup(w http.ResponseWriter, r *http.Request, serviceName
string) {

    // TODO 1: Validate service name is not empty

    // TODO 2: Call s.registry.Lookup(serviceName) to get healthy instances

    // TODO 3: Get total count of all instances (healthy + unhealthy) for this service

    // TODO 4: Build ServiceList response with instances and counts

    // TODO 5: Return JSON response with service details

    // Hint: If no instances found, return empty list with 0 counts (not an error)

    // Hint: HealthyCount should match len(instances), TotalCount includes unhealthy

}


// handleServiceDeregistration processes DELETE /services/{name}/{id} requests

func (s *Server) handleServiceDeregistration(w http.ResponseWriter, r *http.Request,
serviceName string, instanceID string) {

    // TODO 1: Validate serviceName and instanceID are not empty

    // TODO 2: Call s.registry.Deregister(instanceID) to remove the instance

    // TODO 3: Handle case where instance ID is not found (return 404)

    // TODO 4: Return success response confirming deregistration
```

```
    // Hint: The registry should validate that the instance belongs to the specified
service

    // Hint: Deregistration should be idempotent (ok to deregister already-removed
instance)

}


// handleSystemHealth processes GET /health requests

func (s *Server) handleSystemHealth(w http.ResponseWriter, r *http.Request) {

    // TODO 1: Get overall registry statistics from s.registry

    // TODO 2: Get health checker statistics from s.healthChecker

    // TODO 3: Build RegistryStats response combining both data sources

    // TODO 4: Return JSON response with system health overview

    // Hint: Include breakdown by service name showing healthy vs total instances

    // Hint: This endpoint should never fail - return best available information

}
```

Language-Specific Implementation Tips

HTTP Server Management:

- Use `http.Server` struct with explicit timeouts to prevent resource leaks
- Implement graceful shutdown using `Shutdown(ctx)` method with context timeout
- Set reasonable `ReadTimeout` and `WriteTimeout` to prevent slow client attacks

JSON Handling:

- Use `json.NewDecoder(r.Body).Decode()` for request parsing to handle large payloads efficiently
- Always set `Content-Type: application/json` header before encoding responses
- Handle JSON marshaling errors gracefully - they indicate internal data corruption

Error Handling:

- Distinguish between client errors (4xx) and server errors (5xx) consistently
- Log internal errors but don't expose internal details to clients
- Use structured error responses to enable programmatic error handling

Concurrent Access:

- The registry core handles concurrency, but ensure handler functions don't hold locks

- Keep handlers stateless - don't store request data in handler structs
- Use context values for request-scoped data like request IDs

Milestone Checkpoint

After implementing the HTTP API layer, verify functionality with these checkpoints:

Basic Registration Test:

```bash
curl -X POST http://localhost:8080/services \
  -H "Content-Type: application/json" \
  -d '{"Name":"web-api","Host":"192.168.1.100","Port":8080,"HealthEndpoint":"/health"}'
```

Expected: JSON response with `success: true` and an `instanceID` in the data field.

Service Lookup Test:

```bash
curl http://localhost:8080/services/web-api
```

Expected: JSON response with `ServiceList` containing the registered instance and `HealthyCount: 1`.

All Services List Test:

```bash
curl http://localhost:8080/services
```

Expected: Array of `ServiceList` objects, including the registered service.

Health Status Test:

```bash
curl http://localhost:8080/health
```

Expected: `RegistryStats` showing `TotalServices: 1`, `TotalInstances: 1`, and appropriate health breakdown.

Deregistration Test:

```bash
curl -X DELETE http://localhost:8080/services/web-api/{instanceID}
```

Expected: Success response, followed by empty results from lookup endpoints.

Error Handling Tests:

- Send malformed JSON - should return 400 with validation error
- Look up non-existent service - should return empty list (not 404)
- Use unsupported HTTP methods - should return 405 Method Not Allowed

- Send requests without Content-Type - should return 400

Signs of correct implementation:

- All endpoints return consistent JSON format using `SuccessResponse` or `ErrorResponse`
- Validation errors include specific field names and problems
- Concurrent requests don't interfere with each other
- Health filtering works correctly (unhealthy instances don't appear in lookups)
- System health endpoint provides accurate statistics

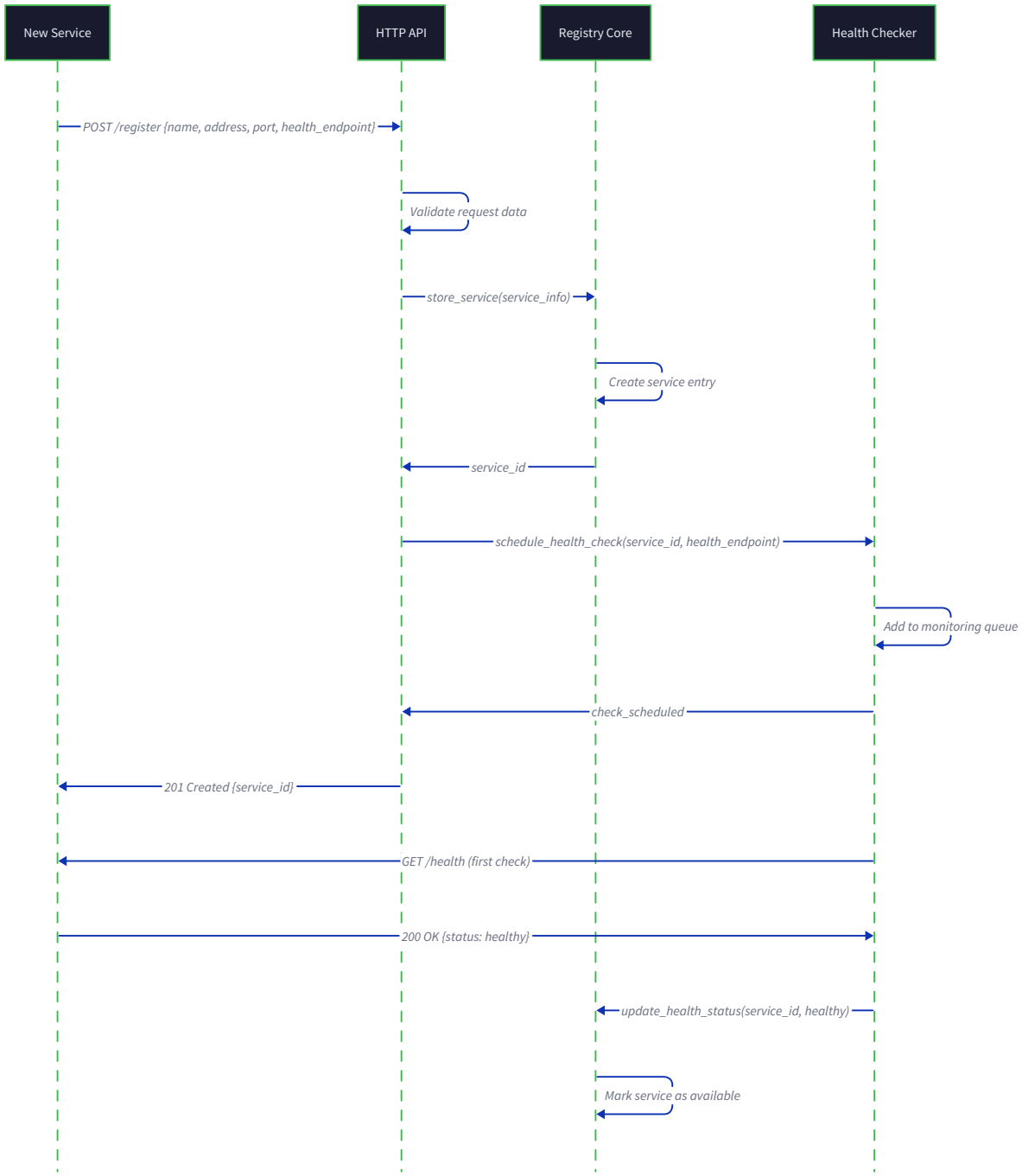
Interactions and Data Flow

> **Milestone(s):** All milestones (1-3) - This section shows how the components from service registration (Milestone 1), health checking (Milestone 2), and HTTP API (Milestone 3) work together to create a complete service discovery system.

Understanding how the three main components of our service discovery system interact is crucial for building a system that works reliably in production. While each component has distinct responsibilities, they must coordinate seamlessly to maintain an accurate, real-time view of service health across a distributed system. This section examines the key workflows that demonstrate these interactions: how services join the registry, how clients discover healthy services, and how the system continuously monitors and updates service health status.

The interactions between components follow well-defined patterns that ensure data consistency and system reliability. The registry core serves as the central authority for all service data, while the health checker operates as a background process that continuously validates service availability. The HTTP API layer acts as the interface between external services and the internal components, translating REST requests into registry operations and health status queries.

Service Registration Flow



The service registration flow represents the first interaction a new service has with the discovery system. This workflow establishes the service's identity, configures health monitoring, and makes the service available for discovery by other clients. Understanding this flow is essential because it sets up all the data structures and monitoring processes that will govern the service's lifecycle within the registry.

When a service starts up and wants to announce its availability, it initiates a carefully orchestrated sequence of operations that span all three components of the system. The process begins with an HTTP request to the API layer, flows through the registry core for data storage and validation, and culminates with the health checker beginning continuous monitoring of the new service instance.

Initial Registration Request

The registration process starts when a service sends an HTTP POST request to the `/services` endpoint. This request contains all the essential metadata that other services will need to communicate with the registering service. The HTTP API layer receives this request and immediately begins a series of validation and processing steps.

The API layer first extracts the service instance data from the JSON request body, parsing fields like service name, host, port, health endpoint, and any associated tags or metadata. This raw data undergoes comprehensive validation to ensure it meets all system requirements and constraints. The validation process checks for required fields, validates network addresses and port ranges, ensures the health endpoint URL is well-formed, and verifies that any provided tags conform to naming conventions.

Validation Check	Field	Criteria	Error Response
Required Fields	Name, Host, Port	Must be present and non-empty	400 Bad Request - Missing required field
Host Format	Host	Valid hostname or IP address	400 Bad Request - Invalid host format
Port Range	Port	Integer between 1-65535	400 Bad Request - Invalid port range
Health Endpoint	HealthEndpoint	Valid HTTP URL path	400 Bad Request - Invalid health endpoint
Service Name	Name	Alphanumeric with hyphens/underscores	400 Bad Request - Invalid service name
Tags Format	Tags	Array of valid tag strings	400 Bad Request - Invalid tag format

Registry Core Processing

Once validation succeeds, the API layer calls the registry core's `Register(instance)` method, passing the validated `ServiceInstance` structure. The registry core takes ownership of the registration process at this point, handling all the complex logic around instance identity, duplicate detection, and data storage.

The registry core first generates a unique instance identifier for the new service registration. This identifier serves as the primary key for all future operations involving this specific service instance. The generation

process combines the service name, host, and port with a timestamp and random component to ensure uniqueness even when services restart quickly or register multiple times.

> **Decision: Instance Identity Strategy**
>
> - **Context**: Need unique identifiers for service instances that can handle rapid restarts and duplicate registrations
> - **Options Considered**:
>   - Service-provided IDs (services generate their own identifiers)
>   - Host-port combinations (use network address as identifier)
>   - Registry-generated UUIDs (system creates unique identifiers)
> - **Decision**: Registry-generated UUIDs with deterministic fallback
> - **Rationale**: Registry-generated IDs prevent collisions and handle edge cases like rapid restarts, while deterministic fallback based on host-port ensures consistency for services that restart quickly
> - **Consequences**: Eliminates ID collision issues but requires registry to track more metadata; enables robust duplicate detection

The registry core then checks for existing registrations that might conflict with the new instance. This duplicate detection process examines both exact matches (same host and port) and potential conflicts (same service name but different metadata). When an exact match is found, the registry updates the existing entry rather than creating a duplicate, ensuring the system maintains a clean view of service instances.

Data Structure Creation and Storage

With a unique instance ID established, the registry core creates the complete data structures that will represent this service throughout its lifecycle. The process begins with creating a `RegistryEntry` that combines the service instance metadata with health status tracking and registration timestamps.

Registry Entry Component	Initial Value	Purpose
Instance	Provided ServiceInstance data	Service metadata and connection information
Health	Uninitialized HealthStatus	Placeholder for health monitoring data
RegisteredAt	Current timestamp	Track registration time for TTL calculations
InstanceID	Generated unique identifier	Primary key for all registry operations

The `HealthStatus` structure within the registry entry starts in an uninitialized state, with status set to "unknown" and all timing fields set to zero values. This reflects the fact that no health checks have been performed yet, and the service's actual availability is unknown until the first health check completes.

The registry core stores this new entry in its internal data structures, typically using the instance ID as the primary key and maintaining secondary indexes by service name for efficient lookup operations. The storage

operation must be thread-safe since multiple services might register simultaneously, requiring careful synchronization to prevent data corruption.

Health Check Initialization

After successful storage in the registry core, the registration process triggers initialization of health monitoring for the new service instance. The registry core notifies the health checker component about the new service through the `ScheduleHealthCheck(instanceID, instance)` method call.

The health checker receives this notification and creates a `ScheduledCheck` entry for the new service instance. This entry defines when the first health check should occur, what interval should be used for subsequent checks, and how many consecutive failures are allowed before marking the service as unhealthy.

Scheduled Check Field	Initial Value	Calculation Method
InstanceID	From registration	Direct copy from registry entry
ServiceName	From ServiceInstance	Used for service-specific health policies
NextCheck	Current time + initial delay	Prevents thundering herd on startup
Interval	From Config.HealthCheckInterval	Default interval, may be service-specific
FailureCount	0	No failures recorded yet

The health checker schedules the first health check with a small random delay to prevent all services from being checked simultaneously if many register at once. This jittering technique helps distribute the health check load and prevents system overload during bulk service registrations.

Response Generation and Client Notification

With all internal processing complete, the registration flow concludes by generating a response to the original HTTP client. The API layer constructs a success response that includes the generated instance ID and confirmation of successful registration.

The response follows the standard `SuccessResponse` format, providing the client with all information needed to manage its registration lifecycle. Most importantly, the response includes the instance ID that the service must use for any future deregistration requests.

Response Field	Value	Purpose
success	true	Indicates successful registration
data.instanceID	Generated unique ID	Required for future deregistration
data.healthEndpoint	Confirmed health endpoint	Shows what will be monitored
data.nextHealthCheck	Estimated first check time	When monitoring begins
message	"Service registered successfully"	Human-readable confirmation

Common Registration Edge Cases

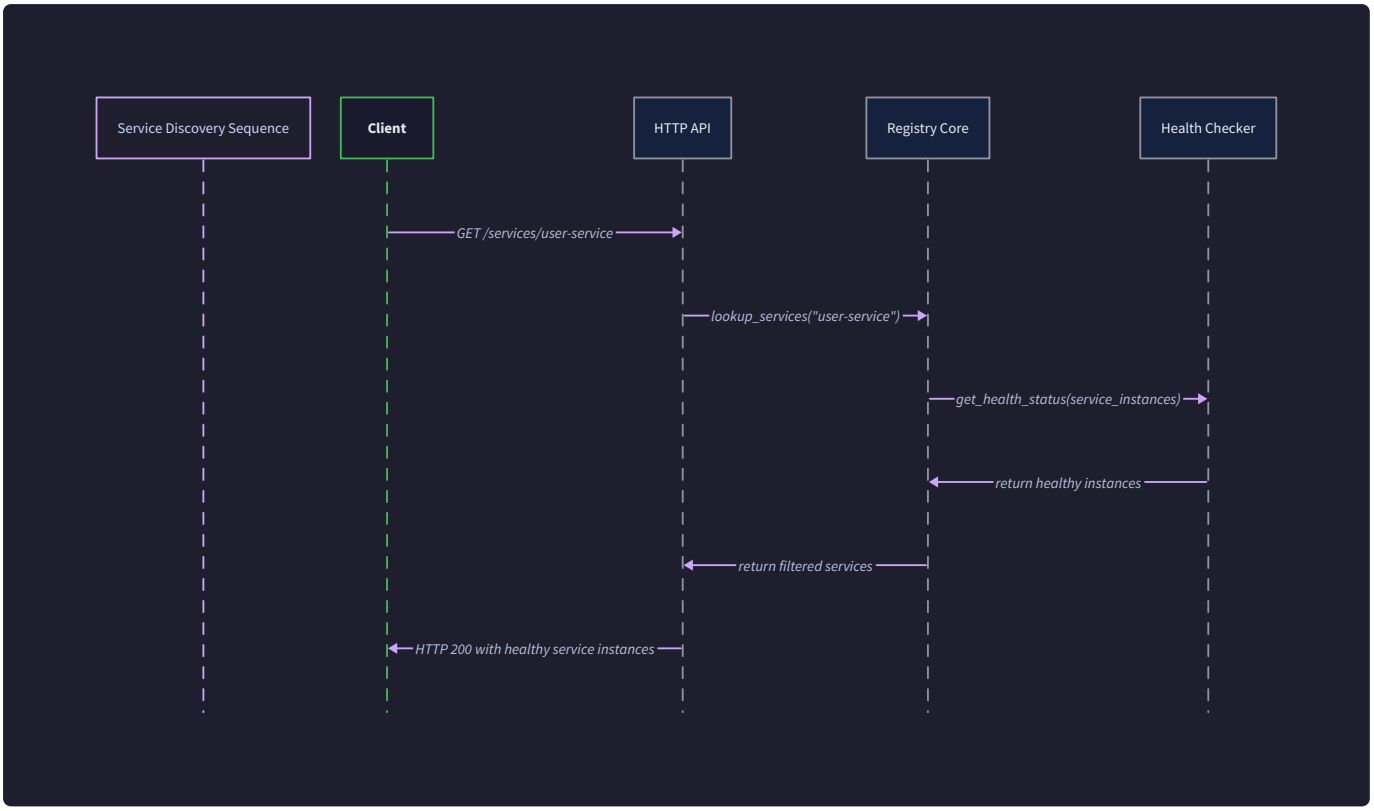
The registration flow must handle several edge cases that commonly occur in distributed systems. These scenarios require careful consideration to ensure the system behaves predictably and recovers gracefully from unusual conditions.

Rapid Re-registration: When a service restarts quickly and attempts to register before its previous registration has been cleaned up, the system must detect this scenario and update the existing entry rather than creating a duplicate. The registry core uses the host-port combination to identify potential duplicates and merges the new registration data with any existing health history that might be relevant.

Partial Failures: If the registration process fails after storing the service in the registry but before initializing health checks, the system must ensure consistency. The registry implements compensation logic that can detect and repair these partial states during the next cleanup cycle.

Network Issues During Registration: When network problems cause the registration HTTP request to fail or timeout, services must implement appropriate retry logic with exponential backoff. The registry API is designed to be idempotent, so repeated registration requests with the same service data will not create duplicate entries.

Service Lookup Flow



The service lookup flow represents how clients discover and retrieve information about available service instances. This workflow is typically the most frequent operation in a service discovery system, as services continuously need to find and communicate with their dependencies. The lookup flow demonstrates the coordination between the HTTP API layer and the registry core to provide clients with current, accurate information about healthy service instances.

Unlike the registration flow, which modifies system state, the lookup flow is primarily a read operation that filters and returns existing data. However, this apparent simplicity masks important complexity around data consistency, health status filtering, and performance optimization for high-frequency access patterns.

Client Lookup Request

The lookup process begins when a client sends an HTTP GET request to the `/services/{serviceName}` endpoint, where the service name identifies which type of service the client wants to discover. This request represents a client's need to find available instances of a particular service to establish communication.

The HTTP API layer receives the lookup request and extracts the service name from the URL path. The API layer performs basic validation on the service name to ensure it matches expected formatting rules and doesn't contain any characters that might indicate injection attacks or malformed requests.

Validation Check	Purpose	Error Response
Service name format	Prevent injection attacks	400 Bad Request - Invalid service name
Service name length	Prevent abuse	400 Bad Request - Service name too long
Special characters	Ensure URL safety	400 Bad Request - Invalid characters
Empty name	Basic requirement	400 Bad Request - Service name required

Registry Query Processing

After validating the request, the API layer calls the registry core's `Lookup(serviceName)` method to retrieve all instances of the requested service. This method triggers a multi-step process within the registry core that involves data retrieval, health filtering, and result preparation.

The registry core first queries its internal data structures to find all registered instances with the specified service name. This operation uses the secondary index maintained by service name to efficiently locate relevant entries without scanning the entire registry. The initial query returns all registered instances regardless of their current health status.

The query process must handle concurrent access carefully since lookup operations occur simultaneously with registrations, deregistrations, and health status updates. The registry core uses read-write locks to ensure that lookup operations can proceed concurrently with each other while being blocked only by write operations that modify the registry state.

Health Status Filtering

Once the registry core retrieves the raw list of service instances, it applies health status filtering to return only instances that are currently available for client connections. This filtering process examines the `HealthStatus` of each instance to determine whether it should be included in the response.

The health filtering logic evaluates several criteria to determine instance availability:

Health Criteria	Evaluation	Include in Results
Status field	Must equal "healthy"	Yes if healthy
Last check time	Within acceptable staleness window	No if too old
Failure count	Below maximum threshold	No if exceeded
Registration TTL	Not expired	No if expired
Health endpoint	Responding to checks	No if unreachable

The filtering process also considers the recency of health information to ensure clients don't receive stale data about service availability. If a service's health status hasn't been updated recently, it may be excluded from

results even if its last known status was healthy, since network partitions or system issues might have prevented recent health checks.

> **The critical insight here is that health filtering must balance freshness with availability - being too strict about health staleness can cause cascading failures when the health checker experiences temporary delays, while being too lenient can direct traffic to failed services.**

Load Balancing Considerations

Although the core service discovery system doesn't implement load balancing algorithms, the lookup flow prepares data in ways that support client-side load balancing decisions. The registry core can optionally shuffle the order of returned instances to prevent clients from always connecting to the first instance in the list.

The response includes metadata that helps clients make intelligent load balancing decisions, such as instance health scores, recent response times, and current failure counts. This information allows sophisticated clients to implement weighted load balancing or prefer instances with better health metrics.

Metadata Field	Purpose	Client Usage
ResponseTime	Average health check latency	Prefer faster instances
FailureCount	Recent failure history	Avoid recently failed instances
LastCheck	Health information freshness	Assess data reliability
RegisteredAt	Instance stability	Prefer stable instances

Response Construction and Optimization

With health filtering complete, the registry core constructs a response containing the list of healthy service instances. The response uses the `ServiceList` structure to provide both the instance data and useful metadata about the overall service health.

The API layer receives the filtered service list from the registry core and constructs an HTTP response using the standard `SuccessResponse` format. The response includes comprehensive information about each healthy instance, enabling clients to make connection decisions without requiring additional API calls.

Response Section	Contents	Client Benefit
Instances array	Healthy ServiceInstance objects	Connection details for each instance
TotalCount	All registered instances	Understand service scale
HealthyCount	Currently available instances	Assess service health
ServiceName	Requested service identifier	Confirm response accuracy

Caching and Performance Optimization

The lookup flow includes several performance optimizations that reduce latency and system load for frequently accessed services. The API layer can implement response caching with short TTL values to serve repeated requests for the same service without querying the registry core.

The caching strategy must balance performance with data freshness, ensuring clients receive reasonably current information about service availability. Cache invalidation occurs when services register, deregister, or experience health status changes, maintaining accuracy while providing performance benefits.

Cache Strategy Decision Table:

Cache Duration	Pros	Cons	Recommended Use
5-10 seconds	High performance, low registry load	Possible stale data	High-frequency lookups
1-2 seconds	Good performance, fresher data	Moderate registry load	Standard lookups
No caching	Always fresh data	High latency, high load	Critical health-sensitive lookups

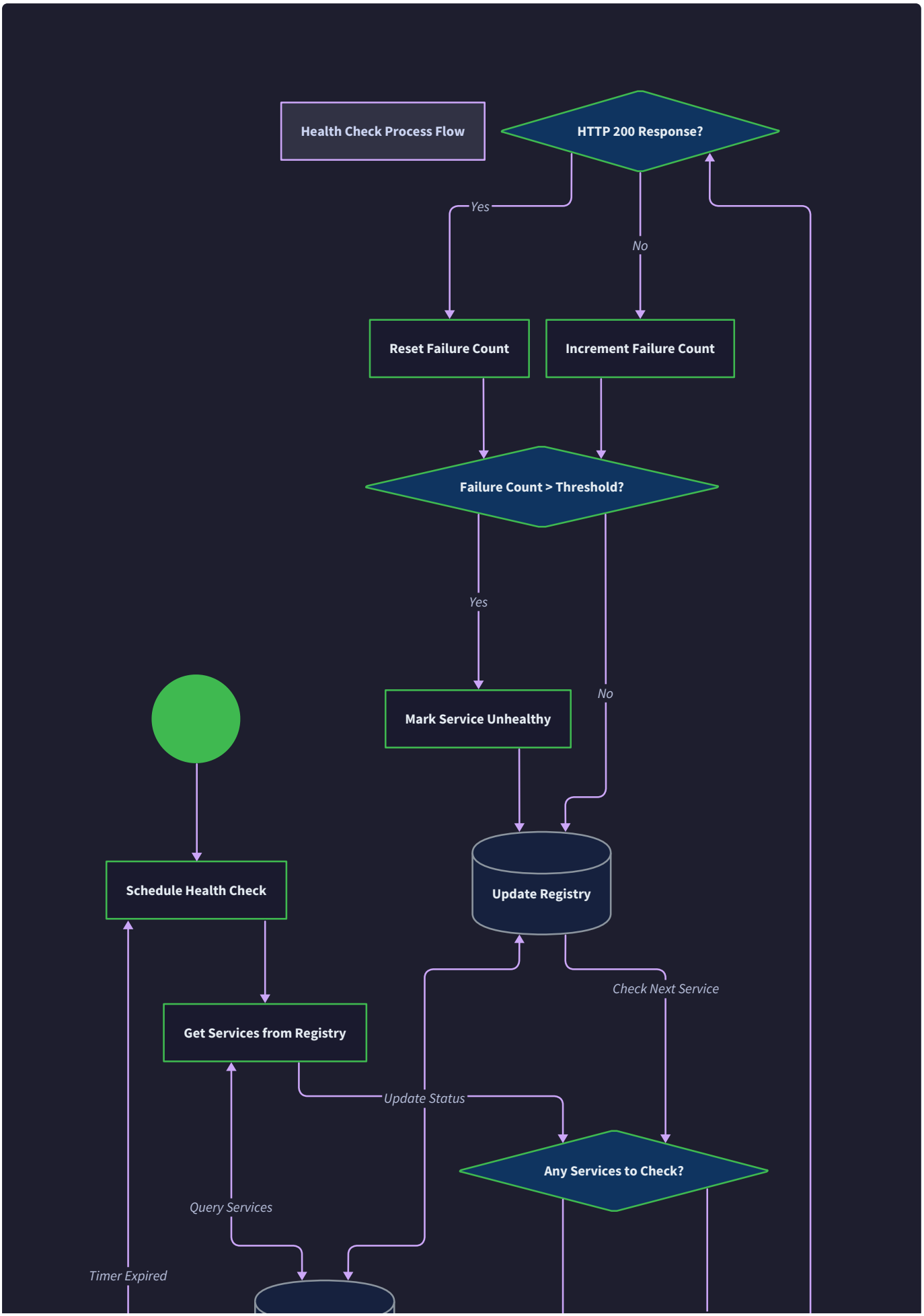
Error Handling and Fallback

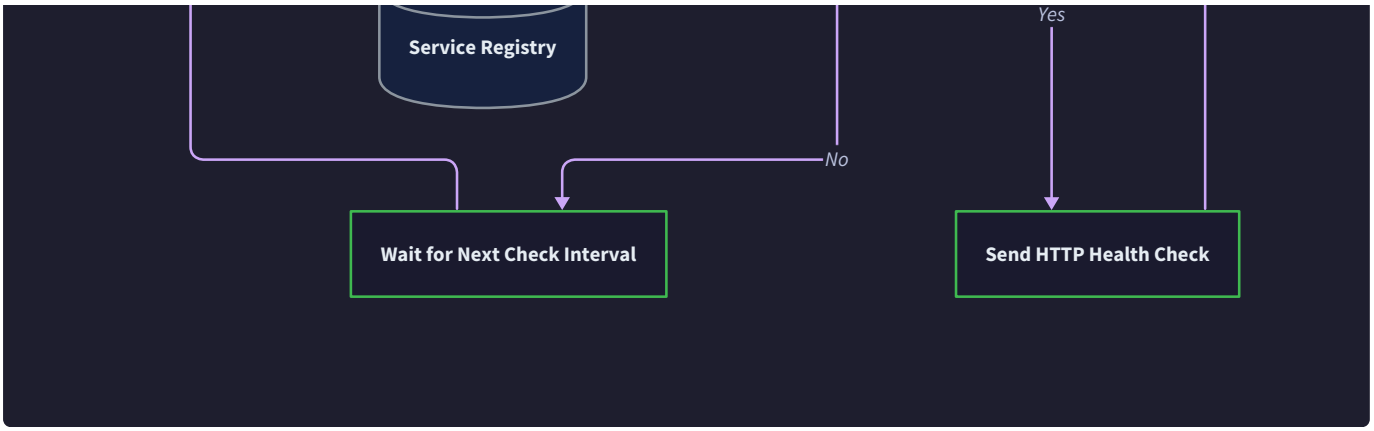
The lookup flow implements comprehensive error handling to ensure clients receive useful information even when parts of the system experience problems. When the registry core encounters issues, the API layer returns appropriate HTTP status codes with detailed error information.

Common error scenarios include service name not found, temporary registry unavailability, and health checker delays that prevent accurate health status determination. Each error condition requires different client-side handling strategies.

Error Condition	HTTP Status	Client Recommendation
Service not found	404 Not Found	Check service name, may not be registered
Registry unavailable	503 Service Unavailable	Retry with exponential backoff
All instances unhealthy	200 OK with empty list	Use cached instances or fallback service
Stale health data	200 OK with warning header	Accept results but retry soon

Health Check Flow


Health Check Process Flow
HTTP 200 Response?
Yes
No
Reset Failure Count
Increment Failure Count
Failure Count > Threshold?
Yes
No
Mark Service Unhealthy
Schedule Health Check
Update Registry
Check Next Service
Get Services from Registry
Update Status
Any Services to Check?
Query Services
Timer Expired



The health check flow represents the background process that continuously monitors service health and maintains the accuracy of the service registry. This workflow operates independently of client requests, running on a scheduled basis to verify that registered services are still available and responsive. The health check flow demonstrates the coordination between the health checker component and the registry core to maintain real-time service availability information.

The health check process is fundamentally different from registration and lookup flows because it operates autonomously in the background, driven by timers and schedules rather than external requests. This background operation must be robust and efficient, since it runs continuously and affects the availability information that clients receive during service discovery.

Health Check Scheduling and Coordination

The health check flow begins with the scheduler component within the health checker identifying which service instances require health verification. The scheduler maintains a priority queue of `ScheduledCheck` entries, ordered by the next check time for each service instance.

The scheduling algorithm runs in a continuous loop, examining the priority queue to identify checks that are due for execution. When a check's scheduled time arrives, the scheduler removes it from the queue and initiates the health verification process. After completing a check, the scheduler calculates the next check time based on the service's configured interval and the check result.

Scheduling Decision	Condition	Next Check Time	Rationale
Successful check	Health endpoint responds	Base interval	Maintain normal monitoring
First failure	Single check failure	Half interval	Increase monitoring frequency
Multiple failures	Consecutive failures	Quarter interval	Rapid failure detection
Recovery	Failed to healthy	Base interval	Return to normal monitoring
Service deregistered	Instance removed	Remove from queue	Stop unnecessary checks

The scheduler implements jittering to prevent synchronized health checks that could overwhelm either the health checker or the target services. Each check time includes a small random offset that spreads the load over time while maintaining the target check frequency.

Health Check Execution Process

When a scheduled check becomes due, the health checker initiates the actual health verification process by calling the `CheckServiceHealth(instance)` method. This method orchestrates the communication with the target service's health endpoint and interprets the response to determine service availability.

The health check execution follows a standardized protocol that works across different types of services and health endpoint implementations. The process begins by constructing an HTTP request to the service's health endpoint, setting appropriate timeouts and headers to ensure the check completes within a reasonable time frame.

HTTP Request Component	Value	Purpose
Method	GET	Standard health check protocol
URL	Service host + health endpoint path	Target the service's health endpoint
Timeout	Config.HealthTimeout (default 10s)	Prevent hanging checks
User-Agent	ServiceDiscovery/1.0 HealthChecker	Identify health check requests
Accept	application/json	Support structured health responses

The health checker sends the HTTP request and waits for a response within the configured timeout period. The interpretation of the response follows standard HTTP health check conventions, where 2xx status codes indicate healthy services and any other status codes or timeouts indicate service problems.

Response Interpretation and Status Determination

After receiving a response from the service's health endpoint, the health checker interprets the result to determine the service's current health status. This interpretation process considers both the HTTP status code and the response timing to make an accurate assessment of service availability.

The health status determination follows a clear decision tree that maps different response scenarios to health outcomes:

Response Scenario	HTTP Status	Response Time	Health Decision	Rationale
Normal response	200-299	Within timeout	Healthy	Service responding normally
Client error	400-499	Within timeout	Healthy	Health endpoint exists but has issues
Server error	500-599	Within timeout	Unhealthy	Service experiencing problems
Connection refused	N/A	Immediate	Unhealthy	Service not running or unreachable
Timeout	N/A	Exceeds limit	Unhealthy	Service too slow or unresponsive
Network error	N/A	Variable	Unhealthy	Network connectivity problems

Decision: Health Check Response Interpretation

- **Context**: Need consistent criteria for determining service health from HTTP responses
- **Options Considered**:
  - Strict interpretation (only 200 OK is healthy)
  - Lenient interpretation (any response is healthy)
  - Configurable interpretation (per-service health criteria)
- **Decision**: Moderate interpretation with 2xx as healthy, errors as unhealthy
- **Rationale**: Balances false positives from strict interpretation with false negatives from lenient approach; provides predictable behavior across different service implementations
- **Consequences**: Services must implement proper health endpoints that return appropriate status codes; reduces false unhealthy markings from minor issues

The health checker also records timing information from each check, including the total response time and any error details. This metadata helps with debugging service issues and provides valuable information for monitoring and alerting systems.

Health Status Update and Registry Integration

After determining the health status from the check response, the health checker updates the service's health information in the registry core through the `UpdateHealth(serviceName, instanceID, status)` method. This update process ensures that the latest health information is immediately available for service lookup operations.

The health status update includes comprehensive information about the check result, not just a simple healthy/unhealthy determination. The `HealthStatus` structure captures all relevant details that might be useful for debugging or monitoring purposes.

Health Status Field	Updated Value	Purpose
Status	"healthy" or "unhealthy"	Primary availability indicator
LastCheck	Current timestamp	Track when information was gathered
FailureCount	Increment on failure, reset on success	Count consecutive failures
ResponseTime	Actual response duration	Performance monitoring
LastError	Error description if applicable	Debugging information

The registry core receives the health update and immediately applies it to the stored `RegistryEntry` for the service instance. This update must be thread-safe since it occurs concurrently with lookup operations and other registry modifications.

Failure Count Management and Thresholds

The health check flow implements sophisticated failure counting logic that distinguishes between temporary service hiccups and persistent failures. This approach prevents single failed checks from immediately marking services as unhealthy while still detecting genuine service problems quickly.

The failure counting system tracks consecutive health check failures for each service instance. When a check succeeds, the failure count resets to zero. When a check fails, the failure count increments, and the system evaluates whether the service should be marked as unhealthy based on the configured failure threshold.

Failure Count	Health Status	Lookup Visibility	Rationale
0	Healthy	Visible	Service responding normally
1	Healthy	Visible	Single failure might be transient
2	Healthy	Visible	Two failures still might be temporary
3+ (at threshold)	Unhealthy	Hidden	Persistent failure detected

The failure threshold is configurable through the `Config.MaxFailures` setting, allowing operators to tune the sensitivity of failure detection based on their environment's characteristics. Networks with high latency or occasional packet loss might require higher thresholds, while reliable networks can use lower thresholds for faster failure detection.

Cleanup and Deregistration Integration

The health check flow coordinates with the registry's cleanup processes to handle service instances that should no longer be monitored. When services deregister explicitly or when TTL expiration removes them from the registry, the health checker must stop monitoring those instances to prevent resource waste.

The health checker maintains synchronization with the registry core through periodic reconciliation processes that identify orphaned health checks. These cleanup operations ensure that the health checker doesn't continue monitoring services that are no longer registered, preventing resource leaks and unnecessary network traffic.

Cleanup Trigger	Detection Method	Action Taken
Explicit deregistration	Registry notification	Remove from schedule immediately
TTL expiration	Registry cleanup notification	Remove from schedule after grace period
Health check failures	Failure count exceeds maximum	Continue checking but mark unhealthy
Network unreachability	Extended failure period	Reduce check frequency but continue

Performance Optimization and Resource Management

The health check flow implements several performance optimizations to ensure it can scale to monitor large numbers of service instances without overwhelming system resources. These optimizations focus on efficient scheduling, connection reuse, and intelligent batching of health checks.

The health checker uses HTTP connection pooling to reduce the overhead of establishing new connections for each health check. This optimization is particularly important when monitoring many instances of the same service, as it allows multiple health checks to reuse the same underlying TCP connections.

Resource Management Strategies:

Resource	Optimization Technique	Benefit
HTTP connections	Connection pooling with per-host limits	Reduce connection establishment overhead
Memory usage	Bounded queues and periodic cleanup	Prevent memory growth with large instance counts
CPU usage	Configurable concurrency limits	Prevent health checking from overwhelming the system
Network bandwidth	Intelligent check distribution	Spread network load over time

The health checker also implements backpressure mechanisms that slow down health checking when system resources become constrained. These mechanisms help maintain system stability during peak load periods or when monitoring very large numbers of service instances.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Client	net/http with default client	Custom http.Client with connection pooling
Concurrency	Basic goroutines with sync.Mutex	Worker pools with channels
Scheduling	time.Ticker with simple loops	Heap-based priority queue
Error Handling	Basic error returns	Structured errors with error wrapping
Logging	Standard log package	Structured logging with levels

Recommended File Structure

The interaction flows span multiple components, so organize the code to reflect these cross-cutting concerns:

```
project-root/
  internal/flows/              ← interaction coordination
    registration.go            ← service registration orchestration
    lookup.go                  ← service discovery coordination
    health_flow.go             ← health check coordination
    flow_test.go               ← integration tests for flows
  internal/registry/           ← registry core
    registry.go                ← main registry implementation
    lookup.go                  ← lookup-specific optimizations
  internal/health/             ← health checker
    scheduler.go               ← check scheduling logic
    checker.go                 ← health check execution
  internal/api/                ← HTTP API layer
    handlers.go                ← request handlers
    middleware.go              ← request processing middleware
```

Flow Coordination Infrastructure

```go
// FlowCoordinator manages the interaction between components during key workflows.
// It provides a centralized point for orchestrating complex operations that span
// multiple components while maintaining clear separation of concerns.
type FlowCoordinator struct {
    registry      *ServiceRegistry
    healthChecker *HealthChecker
    config        *Config
    logger        Logger
}

// NewFlowCoordinator creates a coordinator with all necessary component references.
func NewFlowCoordinator(registry *ServiceRegistry, healthChecker *HealthChecker, config
*Config) *FlowCoordinator {
    return &FlowCoordinator{
        registry:      registry,
        healthChecker: healthChecker,
        config:        config,
        logger:        NewLogger("flow-coordinator"),
    }
}

// FlowContext carries common data through workflow steps, providing a consistent
// way to pass request IDs, timing information, and error details between components.
type FlowContext struct {
    RequestID    string
    StartTime    time.Time
    ClientAddr   string
```

```
    Metadata     map[string]interface{}
    ErrorDetails []string
}

// NewFlowContext creates a new context for tracking a workflow through the system.
func NewFlowContext(requestID string) *FlowContext {
    return &FlowContext{
        RequestID:    requestID,
        StartTime:    time.Now(),
        Metadata:     make(map[string]interface{}),
        ErrorDetails: make([]string, 0),
    }
}
```

Registration Flow Implementation

```go
// ExecuteRegistrationFlow orchestrates the complete service registration process

// across all system components, ensuring proper error handling and rollback.

func (fc *FlowCoordinator) ExecuteRegistrationFlow(ctx *FlowContext, instance
ServiceInstance) (*RegistrationResult, error) {

    // TODO 1: Validate the service instance data using ValidateServiceInstance

    // TODO 2: Call registry.Register(instance) to store the service data

    // TODO 3: On successful registration, notify health checker with ScheduleHealthCheck

    // TODO 4: If health check scheduling fails, roll back the registration

    // TODO 5: Construct RegistrationResult with instance ID and next check time

    // TODO 6: Log the successful registration with timing information

    // Hint: Use defer for cleanup and rollback logic in case of failures

}


// RegistrationResult contains the outcome of a successful registration flow.

type RegistrationResult struct {

    InstanceID       string        `json:"instance_id"`

    HealthEndpoint   string        `json:"health_endpoint"`

    NextHealthCheck  time.Time     `json:"next_health_check"`

    RegistrationTime time.Time     `json:"registration_time"`

    TTLExpiration    time.Time     `json:"ttl_expiration"`

}


// RollbackRegistration removes a partially registered service when the flow fails

// partway through execution, ensuring the system remains in a consistent state.

func (fc *FlowCoordinator) RollbackRegistration(instanceID string) error {

    // TODO 1: Remove the instance from the registry using Deregister

    // TODO 2: Cancel any scheduled health checks for this instance
```

```
    // TODO 3: Log the rollback operation with the reason for failure

    // Hint: Ignore errors during rollback since the system might be in an inconsistent
state

}
```

Lookup Flow Implementation

```go
// ExecuteLookupFlow handles service discovery requests with caching and error handling
// to provide clients with the most current available service instance information.
func (fc *FlowCoordinator) ExecuteLookupFlow(ctx *FlowContext, serviceName string)
(*LookupResult, error) {
    // TODO 1: Validate the service name format and length
    // TODO 2: Check cache for recent lookup results (if caching enabled)
    // TODO 3: Call registry.Lookup(serviceName) to get current instances
    // TODO 4: Filter instances based on health status and TTL expiration
    // TODO 5: Apply any load balancing preparation (shuffle, sort by health)
    // TODO 6: Cache the results for future requests
    // TODO 7: Construct LookupResult with instances and metadata
    // Hint: Handle empty results gracefully - not finding instances isn't an error
}

// LookupResult contains service discovery information with metadata for client decisions.
type LookupResult struct {
    ServiceName     string            `json:"service_name"`
    Instances       []ServiceInstance `json:"instances"`
    TotalCount      int               `json:"total_count"`
    HealthyCount    int               `json:"healthy_count"`
    LastUpdated     time.Time         `json:"last_updated"`
    CacheHit        bool              `json:"cache_hit,omitempty"`
    LookupDuration  time.Duration     `json:"lookup_duration"`
}

// LookupCache provides simple caching for lookup results to reduce registry load
// and improve response times for frequently accessed services.
```

```go
type LookupCache struct {
    cache map[string]*CachedLookup
    mutex sync.RWMutex
    ttl   time.Duration
}

type CachedLookup struct {
    Result    *LookupResult
    ExpiresAt time.Time
}

// Get retrieves a cached lookup result if it's still valid.
func (lc *LookupCache) Get(serviceName string) (*LookupResult, bool) {
    // TODO 1: Acquire read lock for safe concurrent access
    // TODO 2: Check if entry exists in cache map
    // TODO 3: Verify that cached entry hasn't expired
    // TODO 4: Return copy of cached result to prevent modification
}

// Put stores a lookup result in the cache with expiration time.
func (lc *LookupCache) Put(serviceName string, result *LookupResult) {
    // TODO 1: Acquire write lock for cache modification
    // TODO 2: Create CachedLookup with current result and expiration time
    // TODO 3: Store in cache map under service name key
    // TODO 4: Optionally trigger cleanup of expired entries
}
```

Health Check Flow Implementation

```go
// ExecuteHealthCheckFlow runs a single health check cycle for all scheduled instances,

// managing the coordination between scheduling, checking, and status updates.

func (fc *FlowCoordinator) ExecuteHealthCheckFlow(ctx *FlowContext)
(*HealthCheckCycleResult, error) {

    // TODO 1: Get list of checks due for execution from scheduler

    // TODO 2: Execute health checks concurrently with worker pool

    // TODO 3: Collect all check results and update registry for each

    // TODO 4: Reschedule next checks based on results and intervals

    // TODO 5: Update health statistics and cleanup any orphaned checks

    // TODO 6: Return summary of cycle execution with counts and timing

    // Hint: Use limited concurrency to prevent overwhelming target services

}


// HealthCheckCycleResult summarizes the results of a complete health check cycle.

type HealthCheckCycleResult struct {

    ChecksPerformed  int           `json:"checks_performed"`

    SuccessfulChecks int           `json:"successful_checks"`

    FailedChecks     int           `json:"failed_checks"`

    AverageLatency   time.Duration `json:"average_latency"`

    CycleDuration    time.Duration `json:"cycle_duration"`

    ErrorDetails     []string      `json:"error_details,omitempty"`

}


// HealthCheckWorker processes health checks from a work queue, allowing controlled

// concurrency for health check execution without overwhelming the system.

type HealthCheckWorker struct {

    id       int
```

```
    workChan chan *ScheduledCheck
    registry *ServiceRegistry
    checker  *HealthCheckClient
    results  chan *HealthCheckResult
}

// Start begins processing health checks from the work channel.
func (hw *HealthCheckWorker) Start(ctx context.Context) {
    // TODO 1: Listen for ScheduledCheck items on workChan
    // TODO 2: For each check, call checker.CheckServiceHealth
    // TODO 3: Send HealthCheckResult to results channel
    // TODO 4: Handle context cancellation for graceful shutdown
    // TODO 5: Log worker startup and shutdown events
    // Hint: Use select statement to handle both work items and context cancellation
}

// HealthCheckResult contains the outcome of checking a single service instance.
type HealthCheckResult struct {
    InstanceID    string
    ServiceName   string
    Success       bool
    ResponseTime  time.Duration
    StatusCode    int
    Error         error
    CheckTime     time.Time
}
```

Milestone Checkpoints

After Milestone 1 (Service Registry):

- Test registration flow: `curl -X POST localhost:8080/services -d '{"name":"test","host":"localhost","port":3000,"health_endpoint":"/health"}'`
- Expected: JSON response with instance_id
- Test lookup flow: `curl localhost:8080/services/test`
- Expected: Array with registered instance
- Verify: Instance appears in registry data structures

After Milestone 2 (Health Checking):

- Register service and wait 30 seconds
- Check health status: `curl localhost:8080/services/test`
- Expected: Instance marked as healthy or unhealthy based on health endpoint
- Stop test service and wait for failure threshold
- Expected: Instance disappears from lookup results

After Milestone 3 (HTTP API):

- Test complete registration/lookup/deregistration cycle
- Verify all HTTP status codes are appropriate
- Test concurrent registrations and lookups
- Expected: No race conditions or data corruption
- Monitor health check background process
- Expected: Continuous health status updates

Common Flow Issues and Debugging

Symptom	Likely Cause	Debugging Steps
Registration succeeds but lookup fails	Health check not scheduled	Check health checker logs for scheduling errors
Intermittent lookup results	Race condition in health updates	Add logging to health status update path
Health checks not running	Scheduler not started	Verify health checker Start() method called
Memory growth over time	Health checks not cleaned up	Check for orphaned ScheduledCheck entries
Slow lookup responses	No caching or inefficient registry queries	Profile lookup performance and add caching

Error Handling and Edge Cases

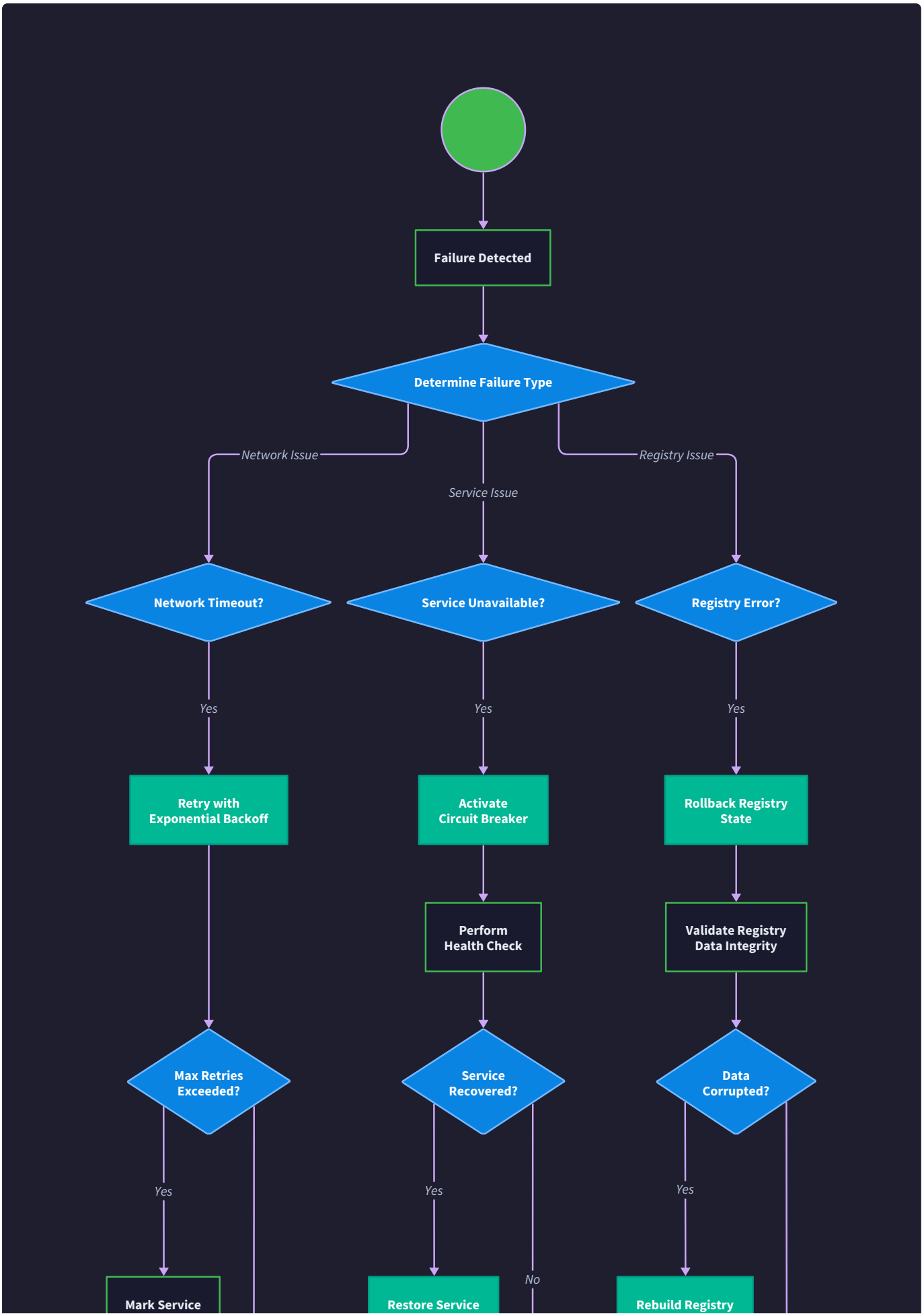
> **Milestone(s):** All milestones (1-3) - Error handling spans across service registration (Milestone 1), health checking (Milestone 2), and HTTP API operations (Milestone 3), with each milestone introducing new failure modes.

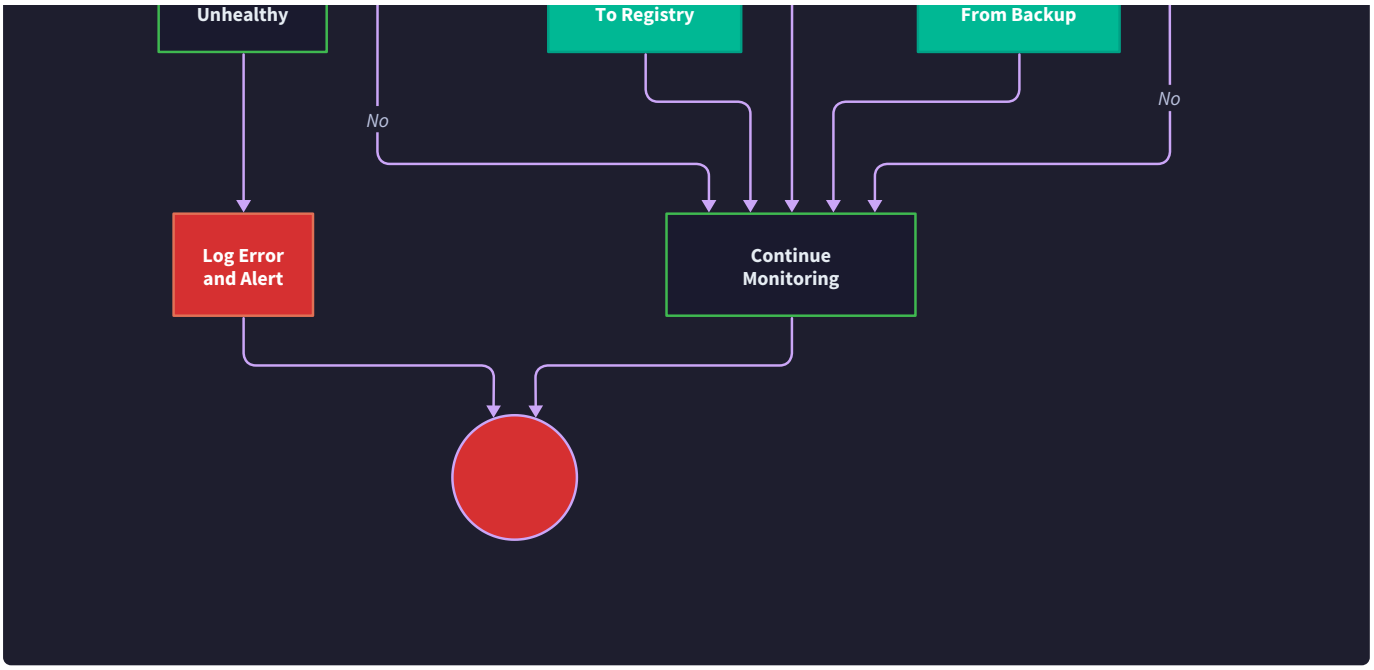
Mental Model: The Hospital Emergency Response

Think of our service discovery system like a hospital managing patient admissions and monitoring. The hospital (service registry) needs to handle three types of emergencies: communication failures (network issues), patient crises (service failures), and overcrowding (system overload). Just as a hospital has protocols for handling ambulance radio failures, distinguishing between minor ailments and critical conditions, and managing capacity during disasters, our system needs comprehensive error handling strategies.

The triage nurse (error detection) must quickly identify the severity of each situation, the emergency response team (recovery mechanisms) must know exactly how to respond to each type of crisis, and the hospital administration (system protection) must prevent any single emergency from overwhelming the entire facility.

Robust error handling in distributed systems requires understanding that failures are not exceptional cases—they are normal operating conditions that must be handled gracefully. Network partitions will occur, services will crash unexpectedly, and traffic spikes will strain system resources. The quality of a service discovery system is measured not by its performance during perfect conditions, but by its behavior when everything goes wrong simultaneously.


Failure Detected
Determine Failure Type
Network Issue
Service Issue
Registry Issue
Network Timeout?
Service Unavailable?
Registry Error?
Yes
Yes
Yes
Retry with Exponential Backoff
Activate Circuit Breaker
Rollback Registry State
Perform Health Check
Validate Registry Data Integrity
Max Retries Exceeded?
Service Recovered?
Data Corrupted?
Yes
Yes
Yes
No
Mark Service
Restore Service
Rebuild Registry



The fundamental principle underlying all error handling in service discovery is **failure isolation**—ensuring that problems with individual services, network segments, or system components do not cascade into system-wide outages. This requires careful design of failure detection mechanisms, recovery procedures, and protective measures that maintain system stability even when multiple components are failing.

Network Failure Scenarios

Network failures represent the most common and challenging class of problems in distributed systems. Unlike service failures, which affect individual components, network failures can partition the system and make it impossible to distinguish between a failed service and a healthy service that cannot be reached.

> **Critical Design Principle**: Network failures must be treated as temporary by default, with escalating responses based on failure duration and pattern analysis.

Connection Timeout Handling

Connection timeouts occur when the health checker cannot establish a connection to a service's health endpoint within the configured timeout period. These failures can indicate network congestion, service overload, or actual service failure, making them particularly challenging to interpret correctly.

The health checking system must implement **progressive timeout handling** that distinguishes between transient network issues and persistent service problems. When a connection timeout occurs, the system should not immediately mark the service as unhealthy, but instead implement a graduated response based on the pattern of failures.

Timeout Scenario	Detection Method	Response Strategy	Recovery Action
Single timeout	Connection establishment > `HealthTimeout`	Increment failure count, schedule immediate retry	No status change if under `MaxFailures`
Intermittent timeouts	Pattern of timeout/success alternating	Extend check interval, reduce timeout sensitivity	Monitor for pattern stabilization
Sustained timeouts	Multiple consecutive timeouts > `MaxFailures`	Mark instance unhealthy, remove from active pool	Continue background checking for recovery
Network partition	All services in subnet timing out simultaneously	Isolate affected network segment, maintain service state	Restore services when connectivity returns

The timeout detection algorithm follows this decision process:

1. **Timeout Detection**: The `HealthCheckClient` monitors connection establishment time and request completion time separately
2. **Pattern Analysis**: The system maintains a sliding window of recent check results to identify timeout patterns
3. **Escalation Logic**: Based on timeout frequency and duration, the system chooses between immediate retry, delayed retry, or service marking
4. **Recovery Detection**: Background checks continue even for timed-out services to detect when connectivity is restored

> **Architecture Decision: Timeout Escalation Strategy**
>
> - **Context**: Network timeouts can indicate temporary congestion or permanent service failure, requiring different responses
> - **Options Considered**: Immediate marking, fixed backoff, adaptive backoff with pattern analysis
> - **Decision**: Progressive escalation with pattern analysis and background recovery checks
> - **Rationale**: Prevents false positives from transient network issues while ensuring failed services are detected promptly
> - **Consequences**: More complex timeout logic but significantly reduced false positive rate and faster recovery

Network Partition Recovery

Network partitions occur when network connectivity is lost between the service registry and groups of services, creating "split-brain" scenarios where services remain healthy but appear failed to the health checker. Recovery from network partitions requires careful orchestration to prevent service registry thrashing.

When connectivity is restored after a partition, the system must **gradually reintegrate** services rather than immediately marking all previously unreachable services as healthy. This prevents overwhelming recovered services with sudden traffic surges and allows time to verify actual service health.

The partition recovery process involves several stages:

1. **Partition Detection**: Identify when multiple services in the same network segment fail simultaneously
2. **State Preservation**: Maintain service registration data for partitioned services rather than immediately removing them
3. **Connectivity Restoration**: Detect when network connectivity to the partitioned segment is restored
4. **Gradual Reintegration**: Slowly increase health check frequency and verify service health before marking services available
5. **Traffic Ramping**: Coordinate with load balancing to gradually increase traffic to recovered services

Partition Phase	System Behavior	Health Check Strategy	Service Availability
Active partition	Mark segment as partitioned, maintain registrations	Suspend health checks for partitioned segment	Remove partitioned services from active pool
Recovery detection	Detect first successful health check in segment	Resume health checks with extended intervals	Keep services unavailable pending verification
Verification	Verify multiple services in segment are responsive	Standard health check interval	Mark verified services as healthy
Full recovery	All segment services verified or expired	Normal health checking resumes	Full service availability restored

Connection Pool Management

The health checking system maintains connection pools to service health endpoints to avoid the overhead of establishing new connections for each health check. However, connection pools can become a source of cascading failures if not managed properly during network issues.

Connection pool exhaustion occurs when many health checks fail with long timeouts, tying up connections and preventing new health checks from being performed. The system must implement **adaptive connection management** that adjusts pool sizes and connection timeouts based on current network conditions.

The connection management strategy includes:

- **Dynamic Pool Sizing**: Adjust connection pool sizes based on health check success rates and response times
- **Connection Health**: Monitor connection staleness and proactively close connections that may be broken
- **Failure Isolation**: Isolate failing endpoints to prevent them from consuming excessive connection pool resources

- **Recovery Optimization**: Prioritize connection pool resources for services showing signs of recovery

Service Failure Detection

Service failures differ from network failures in that they represent actual problems with service functionality rather than connectivity issues. Accurate service failure detection requires distinguishing between different types of service problems and implementing appropriate responses for each.

Temporary vs Permanent Service Failures

The most critical distinction in service failure detection is between **temporary failures** that will resolve automatically and **permanent failures** that require intervention. Temporary failures include resource exhaustion, dependency unavailability, or transient configuration issues. Permanent failures include application crashes, critical resource failures, or configuration errors that prevent service startup.

> **Design Insight**: The key to accurate failure classification is observing failure patterns over time rather than making decisions based on individual health check results.

Failure Type	Characteristics	Detection Pattern	Response Strategy
Temporary resource exhaustion	High response times, occasional timeouts	Gradual degradation followed by recovery	Maintain registration, mark temporarily unhealthy
Dependency failure	Consistent error responses with specific codes	Immediate failure, consistent error pattern	Mark unhealthy, increase check interval
Application crash	Connection refused, no response	Immediate connection failure	Mark unhealthy, attempt restart notification
Configuration error	Consistent HTTP 500 errors	Immediate failure, consistent error response	Mark permanently failed, require manual intervention
Resource leak	Progressively slower responses	Gradual response time increase	Alert for intervention, prepare for restart

The service failure classification algorithm analyzes multiple dimensions of health check results:

1. **Response Time Trends**: Monitor whether response times are increasing, stable, or improving over time
2. **Error Code Patterns**: Analyze HTTP status codes and error messages to identify specific failure modes
3. **Failure Consistency**: Determine whether failures are consistent or intermittent
4. **Recovery Indicators**: Look for signs that a service is attempting to recover or has stabilized in a failed state

Health Endpoint Response Analysis

Service health endpoints can provide detailed information about service status beyond simple success/failure indicators. Sophisticated health checking analyzes response content, timing patterns, and metadata to make more accurate health determinations.

The `HealthCheckResult` structure captures comprehensive information about each health check attempt:

Field	Type	Purpose	Analysis Usage
Success	bool	Basic success/failure indicator	Primary health determination
ResponseTime	time.Duration	Request completion time	Performance trend analysis
StatusCode	int	HTTP response status code	Error classification and pattern detection
Error	string	Detailed error information	Failure mode identification
CheckTime	time.Time	When check was performed	Temporal pattern analysis

Advanced health endpoint analysis includes:

- **Response Content Parsing**: Parse JSON health responses to extract detailed service status information
- **Performance Correlation**: Correlate response times with service load and resource utilization
- **Error Message Classification**: Categorize error messages to identify specific failure modes
- **Trend Analysis**: Monitor health metric trends to predict service failures before they occur

Cascade Failure Prevention

One of the most dangerous scenarios in service discovery is **cascade failure**, where the failure of one service triggers failures in dependent services, ultimately leading to system-wide outages. The service registry must implement protective mechanisms to prevent cascade failures from propagating through the system.

Cascade failure prevention strategies include:

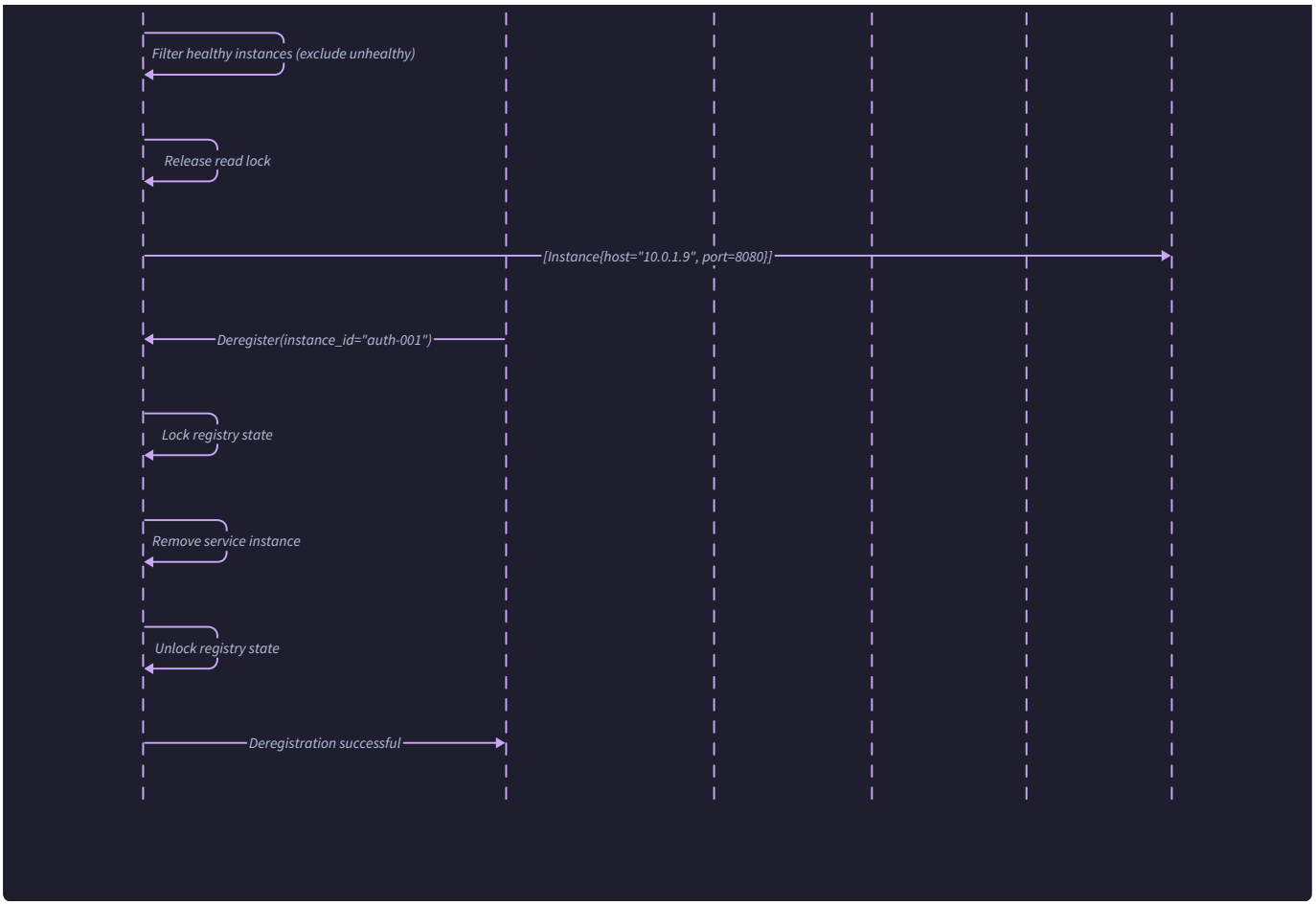
1. **Circuit Breaker Integration**: Coordinate with circuit breakers in client services to prevent overwhelming failing services
2. **Gradual Degradation**: Remove failing services from the registry gradually rather than immediately
3. **Load Shedding Coordination**: Communicate with load balancers to redirect traffic away from struggling services
4. **Dependency Analysis**: Track service dependencies to predict cascade failure propagation paths

Architecture Decision: Cascade Failure Prevention

- **Context**: Service failures can trigger cascade failures that overwhelm the entire system
- **Options Considered**: Immediate removal, gradual degradation, dependency-aware removal
- **Decision**: Gradual degradation with dependency analysis and load coordination
- **Rationale**: Prevents sudden traffic shifts while maintaining system stability during failures
- **Consequences**: More complex failure handling logic but significantly reduced cascade failure risk


Registry
Service A
Service B
Service C
Health Checker
Client
Register(name="auth", host="10.0.1.5", port=8080)
Lock registry state
Validate registration data
Store service instance
Unlock registry state
Registration successful (instance_id="auth-001")
Register(name="user", host="10.0.1.7", port=8081)
Lock registry state
Store service instance
Unlock registry state
Registration successful (instance_id="user-001")
Lookup(service_name="auth")
Read lock registry state
Filter healthy instances
Release read lock
[Instance{host="10.0.1.5", port=8080}]
UpdateHealth(instance_id="auth-001", status=HEALTHY)


Lock registry state
Update health status
Unlock registry state
Register(name="auth", host="10.0.1.9", port=8080)
Lock registry state
Store service instance
Unlock registry state
Registration successful (instance_id="auth-002")
Lookup(service_name="auth")
Read lock registry state
Filter healthy instances
Release read lock
[Instance{host="10.0.1.5"}, Instance{host="10.0.1.9"}]
UpdateHealth(instance_id="auth-001", status=UNHEALTHY)
Lock registry state
Update health status
Unlock registry state
Lookup(service_name="auth")
Read lock registry state



System Overload Protection

System overload occurs when the service registry itself becomes overwhelmed by the volume of registration requests, health checks, or service lookups. Unlike service failures, system overload affects the registry's ability to function correctly, potentially causing it to become a single point of failure for the entire system.

Request Rate Limiting

The HTTP API layer must implement **adaptive rate limiting** that protects the registry from excessive request volumes while maintaining service availability for legitimate clients. Rate limiting must be sophisticated enough to distinguish between normal traffic spikes and potential abuse or misconfiguration.

Rate Limiting Strategy	Trigger Condition	Protection Mechanism	Client Impact
Per-client limiting	Individual client exceeds request threshold	Return HTTP 429 with retry-after header	Specific client must back off
Global rate limiting	Total system request rate exceeds capacity	Apply global request throttling	All clients experience slight delays
Endpoint-specific limiting	Specific API endpoints overwhelmed	Limit access to expensive operations	Registration/deregistration delayed
Emergency throttling	System resources critically low	Aggressive request rejection	Temporary service degradation

The rate limiting implementation uses a **token bucket algorithm** with adaptive refill rates based on system performance metrics. When system response times increase or resource utilization approaches critical levels, the token refill rate decreases automatically, providing back-pressure to clients.

Rate limiting configuration includes:

- **Base Request Rate**: Normal operating request rate per client and globally
- **Burst Allowance**: Short-term burst capacity for handling traffic spikes
- **Adaptive Thresholds**: Dynamic rate adjustments based on system performance
- **Priority Classes**: Different rate limits for different types of operations (registration vs lookup)

Memory Management and Cleanup

The service registry maintains in-memory data structures for service instances, health status, and historical information. Without proper memory management, the registry can experience **memory pressure** that degrades performance and potentially causes crashes.

Memory management strategies include:

1. **TTL-Based Cleanup**: Automatically remove expired service registrations and health check history
2. **LRU Eviction**: Remove least recently used service instances when memory pressure increases
3. **Batch Cleanup Operations**: Perform memory cleanup in batches to minimize performance impact
4. **Memory Pressure Monitoring**: Monitor memory utilization and trigger cleanup when thresholds are exceeded

The cleanup process is coordinated by the `FlowCoordinator` component, which schedules regular cleanup cycles and triggers emergency cleanup when memory pressure is detected:

Cleanup Type	Trigger Condition	Cleanup Scope	Performance Impact
Regular TTL cleanup	Scheduled interval (`CleanupInterval`)	Expired registrations and health data	Minimal, performed during low-traffic periods
Emergency cleanup	Memory utilization > 85%	Aggressive cleanup of all expired data	Moderate, may cause brief response delays
Historical data pruning	Health history size > threshold	Remove old health check results	Low, performed in background
Connection cleanup	Connection pool exhaustion	Close idle and stale connections	Low, improves subsequent performance

Graceful Degradation Strategies

When the service registry approaches resource limits, it must implement **graceful degradation** that maintains core functionality while temporarily reducing service quality. Graceful degradation prevents complete system failure by selectively disabling non-essential features.

The degradation strategy prioritizes service registry operations in order of importance:

1. **Critical Operations**: Service lookup requests for existing healthy services (highest priority)
2. **Important Operations**: Service registration and health status updates
3. **Background Operations**: Health checking and cleanup processes
4. **Optional Operations**: Statistical reporting and historical data collection

During resource pressure, the system progressively disables lower-priority operations while maintaining higher-priority functionality:

Degradation Level	Resource Utilization	Disabled Features	Maintained Functionality
Level 1	70-80%	Historical data collection, detailed statistics	All core operations continue normally
Level 2	80-90%	Background cleanup, non-essential health checks	Core registry and essential health checking
Level 3	90-95%	New service registration, detailed health analysis	Service lookup for existing services only
Emergency	>95%	All non-lookup operations	Read-only service lookup with cached data

⚠️ **Pitfall: Aggressive Resource Protection** Many implementations make the mistake of implementing overly aggressive resource protection that degrades service quality unnecessarily. The system should tolerate

brief resource spikes rather than immediately entering degradation mode. Use smoothed resource metrics over time windows rather than instantaneous measurements, and implement hysteresis in degradation thresholds (different thresholds for entering and exiting degradation modes) to prevent oscillation.

Error Recovery Coordination

When multiple components of the service registry experience errors simultaneously, the system must coordinate recovery efforts to prevent conflicting recovery actions and resource competition. **Recovery coordination** ensures that system recovery is orderly and efficient.

The `FlowCoordinator` manages system-wide error recovery through several mechanisms:

1. **Recovery Prioritization**: Establish the order in which components should be recovered
2. **Resource Allocation**: Ensure recovery operations do not overwhelm available system resources
3. **State Synchronization**: Coordinate state recovery between registry core, health checker, and API layer
4. **Progress Monitoring**: Track recovery progress and adjust strategy if recovery stalls

Recovery coordination follows this priority sequence:

1. **Registry Core Recovery**: Restore service registration data from persistent storage or rebuild from known good state
2. **Health Checker Recovery**: Resume health checking for known services, starting with previously healthy instances
3. **API Layer Recovery**: Restore HTTP API functionality and begin accepting client requests
4. **Background Process Recovery**: Resume cleanup, statistics collection, and other non-essential operations

Common Pitfalls

⚠️ **Pitfall: False Positive Health Failures** One of the most common mistakes in health checking implementation is creating systems that are too sensitive to transient network issues, resulting in healthy services being marked as failed due to temporary connectivity problems. This typically occurs when timeout values are set too low or when a single failed health check immediately marks a service as unhealthy. The solution is implementing failure thresholds that require multiple consecutive failures before marking a service unhealthy, and using appropriate timeout values that account for normal network latency variations.

⚠️ **Pitfall: Inconsistent Error Response Formats** Many implementations fail to maintain consistent error response formats across different failure scenarios, making it difficult for clients to handle errors programmatically. This leads to brittle client implementations that break when new error conditions are encountered. Always use the standardized `ErrorResponse` structure for all error conditions, ensuring that error codes and messages follow consistent patterns that clients can reliably parse and handle.

⚠️ **Pitfall: Resource Leak During Error Conditions** Error handling paths often contain resource leaks because developers focus on the happy path and neglect proper cleanup in error scenarios. This is particularly problematic for network connections, file handles, and memory allocations that are created during

failed operations. Implement comprehensive resource cleanup using defer statements or try-finally blocks, and ensure that all error paths release resources properly.

⚠️ **Pitfall: Cascade Failure Through Health Checking** Aggressive health checking can inadvertently contribute to cascade failures by overwhelming struggling services with health check requests when they are already under stress. This occurs when health check intervals are too frequent or when multiple health checkers probe the same service simultaneously. Implement exponential backoff for health check intervals when services are struggling, and coordinate health checking across multiple registry instances to prevent overlapping checks.

⚠️ **Pitfall: Blocking Error Recovery** Many error recovery implementations use blocking operations that prevent the system from handling new requests while recovery is in progress. This extends outage duration unnecessarily and can make the system appear completely unavailable during recovery periods. Design error recovery as background operations that allow the system to continue serving requests with potentially degraded functionality rather than complete unavailability.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Error Classification	Simple error type enumeration	Machine learning-based failure pattern recognition
Rate Limiting	Token bucket with fixed rates	Adaptive rate limiting based on system metrics
Circuit Breaking	Simple failure count threshold	Hystrix-style circuit breaker with metrics
Monitoring	Basic error counting and logging	Comprehensive observability with distributed tracing
Recovery Coordination	Sequential component restart	Parallel recovery with dependency management

Error Handling Infrastructure

```go
package errors

import (
    "fmt"
    "net/http"
    "time"
)

// ErrorClassifier categorizes different types of failures for appropriate handling
type ErrorClassifier struct {
    networkTimeouts   map[string]int
    serviceFailures   map[string]FailurePattern
    systemOverloads   []OverloadEvent
}

type FailurePattern struct {
    FailureType    string
    Count          int
    FirstOccurred  time.Time
    LastOccurred   time.Time
    RecoveryTime   time.Duration
}

type OverloadEvent struct {
    Timestamp    time.Time
    ResourceType string
    Utilization  float64
    Duration     time.Duration
```

```go
}

// NewErrorClassifier creates an initialized error classifier
func NewErrorClassifier() *ErrorClassifier {
    return &ErrorClassifier{
        networkTimeouts: make(map[string]int),
        serviceFailures: make(map[string]FailurePattern),
        systemOverloads: make([]OverloadEvent, 0),
    }
}

// RateLimiter implements adaptive rate limiting with token bucket algorithm
type RateLimiter struct {
    tokens        float64
    capacity      float64
    refillRate    float64
    lastRefill    time.Time
    requestCounts map[string]int
    globalLimit   int
}

// NewRateLimiter creates a rate limiter with specified capacity and refill rate
func NewRateLimiter(capacity, refillRate float64, globalLimit int) *RateLimiter {
    return &RateLimiter{
        tokens:        capacity,
        capacity:      capacity,
        refillRate:    refillRate,
        lastRefill:    time.Now(),
```

```
        requestCounts: make(map[string]int),
        globalLimit:   globalLimit,
    }
}

// AllowRequest checks if a request should be allowed based on current rate limits
func (rl *RateLimiter) AllowRequest(clientID string) bool {
    // TODO 1: Refill tokens based on time elapsed since last refill
    // TODO 2: Check if client has exceeded individual rate limit
    // TODO 3: Check if global request limit has been exceeded
    // TODO 4: If allowed, consume one token and increment client counter
    // TODO 5: Return whether request should be allowed
    return false
}

// SystemMonitor tracks system resources and triggers protective measures
type SystemMonitor struct {
    cpuThreshold    float64
    memoryThreshold float64
    degradationLevel int
    lastCheck       time.Time
}

// NewSystemMonitor creates a system monitor with specified thresholds
func NewSystemMonitor(cpuThreshold, memoryThreshold float64) *SystemMonitor {
    return &SystemMonitor{
        cpuThreshold:    cpuThreshold,
        memoryThreshold: memoryThreshold,
```

```
        degradationLevel: 0,
        lastCheck:       time.Now(),
    }
}

// CheckSystemHealth evaluates current system health and returns degradation level
func (sm *SystemMonitor) CheckSystemHealth() int {
    // TODO 1: Get current CPU utilization percentage
    // TODO 2: Get current memory utilization percentage
    // TODO 3: Compare against thresholds to determine degradation level
    // TODO 4: Update degradation level with hysteresis to prevent oscillation
    // TODO 5: Return current degradation level (0-3)
    return 0
}
```

Error Recovery Coordination

```go
package recovery

import (
    "context"
    "sync"
    "time"
)

// RecoveryCoordinator manages system-wide error recovery operations
type RecoveryCoordinator struct {
    registry      ServiceRegistry
    healthChecker HealthChecker
    server        Server
    recoveryState RecoveryState
    mutex         sync.RWMutex
}

type RecoveryState struct {
    Phase          string
    StartTime      time.Time
    ComponentStates map[string]ComponentState
    OverallProgress float64
}

type ComponentState struct {
    Status      string
    Progress    float64
    LastUpdated time.Time
```

```
    ErrorCount  int
}

// NewRecoveryCoordinator creates a recovery coordinator for system components
func NewRecoveryCoordinator(registry ServiceRegistry, healthChecker HealthChecker, server
Server) *RecoveryCoordinator {
    return &RecoveryCoordinator{
        registry:      registry,
        healthChecker: healthChecker,
        server:        server,
        recoveryState: RecoveryState{
            Phase:           "healthy",
            ComponentStates: make(map[string]ComponentState),
        },
    }
}

// ExecuteRecovery coordinates recovery of all system components
func (rc *RecoveryCoordinator) ExecuteRecovery(ctx context.Context) error {
    // TODO 1: Assess current system state and determine recovery requirements
    // TODO 2: Create recovery plan based on component failure states
    // TODO 3: Execute registry core recovery with data restoration
    // TODO 4: Resume health checking with graduated service reintegration
    // TODO 5: Restore HTTP API functionality and client request handling
    // TODO 6: Monitor recovery progress and adjust strategy if needed
    // TODO 7: Return to normal operation when all components are healthy
    return nil
}
```

```go
// NetworkFailureHandler specifically handles network-related failures
type NetworkFailureHandler struct {
    timeoutThresholds map[string]time.Duration
    partitionedSegments map[string]PartitionInfo
    connectionPools   map[string]*ConnectionPool
}

type PartitionInfo struct {
    SegmentID     string
    Services      []string
    PartitionTime time.Time
    RecoveryTime  *time.Time
}

type ConnectionPool struct {
    activeConns   int
    maxConns      int
    idleConns     []Connection
    healthyConns  []Connection
}

// HandleNetworkPartition manages service registry state during network partitions
func (nfh *NetworkFailureHandler) HandleNetworkPartition(segmentID string, affectedServices
[]string) {
    // TODO 1: Identify all services in the partitioned network segment
    // TODO 2: Mark segment as partitioned and preserve service registration data
    // TODO 3: Suspend health checking for services in partitioned segment
    // TODO 4: Remove partitioned services from active service pool
    // TODO 5: Set up monitoring for partition recovery detection
```

```
    // TODO 6: Log partition event for operational awareness
}

// DetectPartitionRecovery monitors for network partition recovery
func (nfh *NetworkFailureHandler) DetectPartitionRecovery(segmentID string) bool {
    // TODO 1: Attempt health checks on sample services from partitioned segment
    // TODO 2: Verify successful responses from multiple services in segment
    // TODO 3: Confirm network connectivity is stable over time window
    // TODO 4: Return true if partition has been resolved
    return false
}
```

Milestone Checkpoints

Milestone 1 Error Handling Checkpoint: After implementing basic error handling for service registration, verify:

- Registration requests with invalid data return appropriate HTTP 400 errors with detailed error messages
- Duplicate registration attempts are handled gracefully without corrupting registry state
- Concurrent registration and deregistration operations don't cause race conditions
- Memory usage remains stable even with high registration/deregistration rates

Test command: `curl -X POST localhost:8080/services -d '{"invalid":"data"}' -H "Content-Type: application/json"` Expected: HTTP 400 response with structured error message explaining validation failure

Milestone 2 Error Handling Checkpoint: After implementing health check error handling, verify:

- Services with failing health checks are gradually removed from registry after configured failure threshold
- Network timeouts don't immediately mark healthy services as failed
- Health check system recovers gracefully from temporary network partitions
- System continues operating normally even when majority of services are failing health checks

Test command: Start service, stop service process, observe health check failure progression Expected: Service marked unhealthy after `MaxFailures` consecutive failures, background checking continues

Milestone 3 Error Handling Checkpoint: After implementing HTTP API error handling, verify:

- API returns consistent error response format for all error conditions

- Rate limiting prevents API overload during high request volumes
- System gracefully degrades functionality during resource pressure
- Error recovery restores full functionality after temporary failures

Test command: `ab -n 1000 -c 10 localhost:8080/services` (load testing) Expected: API remains responsive, rate limiting activates if needed, no crashes or data corruption

The comprehensive error handling implementation ensures that the service discovery system remains stable and functional even when facing multiple simultaneous failure modes, providing the reliability required for production microservice environments.

Testing Strategy

> **Milestone(s):** All milestones (1-3) - This section provides comprehensive testing approaches that validate service registration (Milestone 1), health checking (Milestone 2), and HTTP API operations (Milestone 3).

Mental Model: The Quality Control Inspector

Think of testing a service discovery system like a quality control inspector in a manufacturing plant with multiple production lines. The inspector has three levels of scrutiny: **component testing** (examining individual parts in isolation), **integration testing** (verifying that assembled components work together), and **milestone validation** (confirming that completed product stages meet specifications).

Just as the inspector uses different tools and techniques for each level—magnifying glass for component defects, functional tests for assembled units, and full performance trials for finished products—our testing strategy employs unit tests with mocks for isolated components, integration tests for component interactions, and milestone checkpoints for end-to-end validation.

The inspector must also verify that the system gracefully handles anomalies: what happens when a component fails, when supply chains are disrupted, or when demand exceeds capacity? Similarly, our testing strategy must validate error scenarios, network failures, and system overload conditions to ensure robust behavior under stress.

Unit Testing Approach

Unit testing in service discovery focuses on testing individual components in complete isolation using test doubles and controlled data. Each component—`ServiceRegistry`, `HealthChecker`, and `Server`—has distinct responsibilities that can be verified independently without external dependencies or side effects.

Component Isolation Strategy

The unit testing approach treats each component as a black box with well-defined interfaces. The `ServiceRegistry` component is tested by providing controlled `ServiceInstance` data and verifying that registration, deregistration, and lookup operations maintain correct internal state. Health checking logic is isolated by mocking the HTTP client and time-based scheduling, allowing tests to verify failure counting and threshold logic without actual network calls or waiting for real time intervals.

The HTTP API layer requires mocking the underlying `ServiceRegistry` and `HealthChecker` dependencies, enabling tests to focus purely on request parsing, response formatting, and error handling. This isolation ensures that API tests remain fast and deterministic, independent of the complex state management and background processing in lower layers.

> **Critical Testing Principle**: Unit tests must be completely deterministic and fast. Any test that depends on external systems, real network calls, actual time passage, or file system state is not a unit test—it's an integration test and belongs in a different test suite.

Test Data Management

Unit tests require carefully constructed test data that covers both typical scenarios and edge cases. Test data should include valid `ServiceInstance` configurations with various combinations of tags, health endpoints, and metadata. Edge case data must test boundary conditions: empty service names, invalid port numbers, malformed health endpoints, and extremely long tag lists.

Mock data for health checking should simulate various response scenarios: successful HTTP 200 responses, timeout conditions, connection refused errors, and malformed response bodies. The test data should also include time-based scenarios where health checks succeed initially but begin failing after a specific number of attempts, allowing verification of failure counting and threshold logic.

Mock and Test Double Strategy

Effective unit testing requires sophisticated mocking of external dependencies. The HTTP client used by `HealthChecker` must be replaced with a mock that can simulate network conditions: successful responses with configurable latency, various HTTP error codes, connection timeouts, and DNS resolution failures. The mock should track call patterns to verify that health checks are scheduled and executed according to configuration.

Time-based operations require special attention in unit testing. Rather than using real time, tests should inject a controllable time source that allows advancing time instantly and verifying time-based behavior like TTL expiration, health check scheduling, and cleanup operations. This approach makes tests both faster and more reliable.

The registry's background cleanup operations present particular testing challenges. Tests must verify that expired registrations are removed, that cleanup operations don't interfere with active registrations, and that cleanup frequency matches configuration. This requires mocking the cleanup scheduler and providing controlled time advancement.

Core Testing Patterns

Unit tests should follow consistent patterns that make them easy to understand and maintain. Each test should follow the Arrange-Act-Assert pattern: set up test data and mocks, execute the operation being tested, and verify expected outcomes. Setup and teardown should be handled by test fixtures to ensure clean state between test runs.

State verification requires comprehensive assertions that check not only the direct return values but also the internal state changes. When testing `Register`, verify not only that the operation returns success but also that the service appears in subsequent `Lookup` calls, that health checking is initiated, and that the registration timestamp is recorded correctly.

Error condition testing requires verifying both error return values and system state. When registration fails due to validation errors, verify that no partial state is left in the registry, that no health checks are scheduled for the invalid service, and that the error response contains appropriate diagnostic information.

Test Category	Focus Area	Mock Requirements	Verification Points
Registry Core	Registration/lookup logic	Time source, cleanup scheduler	Service state, lookup results, expiration
Health Checker	Failure detection logic	HTTP client, time source	Failure counts, status updates, scheduling
HTTP API	Request/response handling	Registry, health checker	Status codes, response format, validation
Configuration	Setting validation	File system, environment	Default values, override logic, validation
Error Handling	Failure scenarios	Various failure modes	Error messages, state consistency, recovery

Test Coverage Requirements

Unit test coverage should focus on decision points and state transitions rather than raw line coverage. Every conditional branch in the registration logic should have tests that exercise both the true and false paths. State machines like health status transitions require tests for every valid state change and verification that invalid transitions are properly rejected.

Boundary condition testing is particularly important for service discovery systems. Tests should verify behavior at configuration limits: maximum number of registered services, health check timeout boundaries, TTL expiration edge cases, and rate limiting thresholds. These boundary tests often reveal off-by-one errors and race conditions that are difficult to detect in normal operation.

Error Injection Testing

Unit tests must systematically verify error handling by injecting failures at every external dependency point. Mock the HTTP client to return various error conditions and verify that the health checker responds appropriately: temporary failures increment failure counts without immediate deregistration, while permanent failures trigger immediate removal from the active service list.

Configuration error testing should verify that invalid configurations are detected early and result in clear error messages. Test scenarios should include missing required fields, invalid port ranges, negative timeout values, and conflicting settings. The system should fail fast with diagnostic information rather than attempting to operate with invalid configuration.

Integration Testing Scenarios

Integration testing verifies that components work correctly together in realistic scenarios without mocks or test doubles. These tests validate the complete data flow from HTTP API requests through registry operations to health checking and back to API responses. Integration tests catch issues that unit tests miss: timing dependencies, protocol mismatches, and state synchronization problems.

End-to-End Workflow Validation

Integration tests must verify complete workflows from start to finish. The service registration workflow test begins with an HTTP POST request to register a service, verifies that the registry stores the service data correctly, confirms that health checking begins automatically, waits for the first health check to complete, and validates that the service appears in lookup results with correct health status.

The service deregistration workflow test registers a service, confirms it's healthy and discoverable, sends a DELETE request to remove the service, verifies that health checking stops immediately, and confirms that the service no longer appears in lookup results. This test must also verify that any cached data is invalidated and that cleanup operations complete successfully.

The health status change workflow test registers a service, waits for it to become healthy, stops the mock service to trigger health check failures, monitors the failure count progression, verifies that the service becomes unavailable after reaching the failure threshold, restarts the mock service, and confirms that the service recovers to healthy status.

Component Interaction Testing

Integration tests must verify that components communicate correctly through their defined interfaces. The registry-to-health-checker interaction test verifies that newly registered services are automatically added to the health check schedule, that configuration changes (like modified check intervals) are propagated correctly, and that deregistered services are removed from the checking schedule immediately.

The health-checker-to-registry interaction test verifies that health status updates are applied correctly to registry state, that health check results trigger appropriate state transitions, and that registry queries return services with current health status. This test must also verify that health check operations don't interfere with concurrent registration and lookup operations.

The API-to-registry interaction test verifies that HTTP requests are correctly translated to registry operations, that registry responses are properly formatted as HTTP responses, and that error conditions in the registry layer result in appropriate HTTP error codes and error messages.

Concurrent Operation Testing

Service discovery systems must handle concurrent operations correctly since multiple services may register, deregister, or be looked up simultaneously while health checking runs continuously in the background. Integration tests must verify that these concurrent operations don't corrupt shared state or cause deadlocks.

The concurrent registration test launches multiple goroutines that simultaneously register different services and verifies that all registrations succeed, that no service data is lost or corrupted, and that health checking begins for all registered services. This test should also verify that the final registry state matches expectations regardless of the order in which registrations complete.

The concurrent lookup test registers multiple services, then launches multiple goroutines that simultaneously perform service lookups while background processes modify service health status. The test verifies that lookup results are always consistent—no partially updated service information is returned—and that lookup operations don't interfere with health status updates.

The registration-during-health-check test verifies that new service registrations work correctly even while health checking is actively running. This test should register an initial service, wait for health checking to begin, register additional services during active health checking, and verify that health checking coverage expands correctly to include the new services.

Integration Test Scenario	Components Involved	Success Criteria	Failure Modes Tested
Complete Service Lifecycle	API + Registry + Health Checker	Registration → Health → Discovery → Deregistration	Network failures during registration
Concurrent Service Operations	API + Registry	Multiple simultaneous operations complete correctly	Race conditions in shared state
Health Status Transitions	Registry + Health Checker	Healthy → Unhealthy → Healthy transitions work	Health check timeouts and recovery
API Error Handling	API + Registry + Health Checker	Proper error codes and messages	Invalid requests and system failures
Configuration Changes	All Components	Runtime configuration updates apply correctly	Configuration conflicts and validation

Network Failure Simulation

Integration tests must verify behavior under realistic network conditions. The network partition test starts multiple mock services, allows them to become healthy, simulates network connectivity loss to some services

while maintaining connectivity to others, verifies that affected services are marked unhealthy while unaffected services remain available, restores network connectivity, and confirms that previously affected services recover to healthy status.

The health check timeout test configures short health check timeouts, starts mock services that respond slowly, verifies that slow-responding services are marked unhealthy due to timeouts, improves mock service response times, and confirms that services recover when response times improve. This test validates that timeout handling works correctly and doesn't cause false positives.

The DNS resolution failure test registers services using hostnames rather than IP addresses, simulates DNS resolution failures for some hostnames, verifies that affected services are marked unhealthy due to DNS failures, restores DNS resolution, and confirms service recovery. This test is crucial since many real deployments use service names rather than IP addresses.

Load and Stress Testing

Integration tests should verify system behavior under load conditions that might occur in production. The high registration volume test rapidly registers large numbers of services and verifies that all registrations succeed, that health checking scales appropriately to handle all registered services, that lookup performance remains acceptable, and that memory usage remains bounded.

The high lookup volume test registers a moderate number of services and then performs a high volume of concurrent lookup operations while health checking continues in the background. This test verifies that lookup operations remain fast and consistent, that health checking isn't disrupted by lookup volume, and that the system doesn't degrade under query load.

The configuration stress test repeatedly modifies configuration settings while the system is under moderate load, verifying that configuration changes are applied correctly, that existing operations continue to work during configuration updates, and that invalid configuration changes are rejected without affecting system operation.

Data Consistency Validation

Integration tests must verify that data remains consistent across all system components even under failure conditions. The consistency test performs a complex sequence of operations: register multiple services, wait for health status to stabilize, force some services to fail health checks, perform lookup operations to verify consistency, restart failed services, wait for recovery, and perform final validation that all data matches expected state.

The recovery consistency test simulates system restart scenarios by stopping the registry service while maintaining mock services, restarting the registry, and verifying that the system recovers to a consistent state: previously registered services are restored from persistent state, health checking resumes correctly, and lookup operations return accurate results.

Milestone Validation Checkpoints

Each project milestone requires specific validation checkpoints that verify the implementation meets acceptance criteria and functions correctly in realistic scenarios. These checkpoints serve as comprehensive integration tests that validate not just individual features but the complete milestone functionality working together.

Milestone 1: Service Registry Validation

The Milestone 1 checkpoint validates the core service registry functionality through a comprehensive test scenario. The validation begins by starting the registry service and verifying that it accepts HTTP requests on the configured port. The test then registers multiple service instances with different configurations: services with different names, multiple instances of the same service, services with various tag combinations, and services with different health endpoint configurations.

Registration validation verifies that each service registration returns a unique instance ID, that the instance ID can be used for subsequent operations, that registered services appear in the complete service list, and that service metadata is stored and returned accurately. The test should also verify that registration timestamps are recorded correctly and that TTL expiration works as expected.

Lookup validation tests service discovery functionality by querying for services by name and verifying that all registered instances are returned with correct metadata. The test should verify lookup behavior for non-existent services (empty result set), lookup behavior for services with multiple instances (all instances returned), and lookup performance under various query patterns.

Deregistration validation verifies that services can be removed from the registry using their instance IDs, that deregistered services no longer appear in lookup results, that deregistered services are removed from the complete service list, and that deregistration of one instance doesn't affect other instances of the same service.

Validation Check	Expected Behavior	Test Method	Success Criteria
Service Registration	Unique ID returned, service discoverable	POST /services with valid data	HTTP 201, valid instance ID, service in lookup results
Service Deregistration	Service removed from registry	DELETE /services/{name}/{id}	HTTP 200, service not in lookup results
Service Lookup	All healthy instances returned	GET /services/{name}	HTTP 200, correct instance list
Complete Service List	All services visible	GET /services	HTTP 200, comprehensive service list
TTL Expiration	Expired services auto-removed	Wait beyond TTL period	Expired services absent from lookup

Milestone 2: Health Checking Validation

The Milestone 2 checkpoint builds on Milestone 1 by adding health checking functionality. The validation starts multiple mock services that implement health endpoints returning HTTP 200 for healthy status. The test registers these services with the registry and verifies that health checking begins automatically with configured intervals.

Health endpoint validation verifies that the health checker makes HTTP requests to the correct health endpoints, that successful health checks mark services as healthy, that healthy services appear in lookup results, and that health check timing follows configuration settings. The test should monitor actual HTTP requests to mock services to verify that health checking is working correctly.

Failure detection validation tests the system's response to service failures by stopping mock services or configuring them to return error responses. The test verifies that consecutive health check failures increment the failure count, that services are marked unhealthy after reaching the failure threshold, that unhealthy services are excluded from lookup results, and that failure detection timing matches configuration.

Recovery validation tests service recovery by restarting failed mock services and verifying that health checking detects the recovery, that failure counts reset after successful health checks, that services are marked healthy again, and that recovered services reappear in lookup results.

Configuration validation tests health checking behavior with different configuration settings: various health check intervals, different failure thresholds, and different health check timeout values. The test should verify that configuration changes take effect correctly and that health checking behavior matches the configured parameters.

Milestone 3: HTTP API Validation

The Milestone 3 checkpoint validates the complete HTTP API functionality building on the registry core and health checking from previous milestones. The validation tests all API endpoints with various request types and verifies that the API correctly handles both success and error scenarios.

API endpoint validation tests each endpoint with valid requests and verifies correct responses: POST /services successfully registers services and returns instance IDs, GET /services returns the complete list of services, GET /services/{name} returns instances for the specified service, DELETE /services/{name}/{id} successfully deregisters services, and GET /health returns system health status.

Request validation testing verifies that the API properly validates input data and returns appropriate error responses for invalid requests: malformed JSON in registration requests, missing required fields, invalid port numbers, malformed URLs, and unsupported HTTP methods. The test should verify that error responses include helpful diagnostic information and use correct HTTP status codes.

Response format validation verifies that all API responses use consistent JSON formatting, that success responses include expected data fields, that error responses follow a standard error format, and that HTTP status codes are used correctly throughout the API.

Content negotiation validation tests that the API handles HTTP headers correctly: Content-Type verification for request bodies, Accept header handling for responses, and proper HTTP method handling for each endpoint.

Cross-Milestone Integration Validation

The final validation checkpoint tests the complete system functionality with all milestones integrated together. This comprehensive test simulates realistic usage patterns that exercise service registration, health checking, and API operations simultaneously.

The multi-service scenario registers multiple services of different types, waits for all services to become healthy, performs service discovery operations to verify all services are discoverable, simulates failure of some services, verifies that failed services are detected and excluded from discovery, recovers failed services, and verifies complete system recovery.

The load testing scenario performs high-volume operations across all system components: rapid service registrations and deregistrations, high-frequency service lookups, concurrent API operations, and sustained operation under load. This test verifies that system performance remains acceptable under realistic load conditions.

The persistence and recovery scenario stops and restarts the registry service while maintaining external mock services, verifies that the system recovers correctly after restart, checks that service states are restored accurately, and confirms that health checking resumes correctly after recovery.

Milestone	Core Validation	Performance Check	Error Handling Check
Milestone 1	Registration/lookup accuracy	Lookup latency under load	Invalid registration handling
Milestone 2	Health checking accuracy	Health check scalability	Network failure detection
Milestone 3	API completeness	API response times	Request validation and errors
Integrated	End-to-end workflows	System-wide performance	Complete failure recovery

Automated Validation Procedures

Each milestone checkpoint should be implementable as automated test suites that can be run repeatedly during development. The validation procedures should include setup scripts that start necessary mock services, configuration files that define test parameters, test execution scripts that run the validation scenarios, and cleanup procedures that restore the system to a clean state.

The validation automation should produce clear pass/fail results for each checkpoint, detailed logs of test execution for debugging failed validations, performance metrics that can be tracked over time, and summary reports that highlight any regressions or improvements.

Continuous integration should run these validation checkpoints automatically on code changes, ensuring that new development doesn't break existing milestone functionality. The checkpoints should also be runnable locally by developers to validate their implementations before committing changes.

> **Key Insight**: Milestone validation checkpoints serve as both acceptance criteria verification and regression test suites. They provide concrete evidence that each milestone's functionality works correctly and continues to work as the system evolves.

Implementation Guidance

A. Testing Framework Recommendations

Component	Simple Option	Advanced Option
Unit Testing	Go standard testing package	Testify for assertions and mocks
HTTP Testing	httptest package	Ginkgo/Gomega BDD framework
Mock Generation	Manual mocks	GoMock for automatic mock generation
Test Data	JSON fixtures	Factory pattern with Faker library
Integration	Docker containers	Testcontainers for service dependencies

B. Recommended Test File Structure

```
project-root/
  internal/registry/
    registry.go
    registry_test.go              ← unit tests for registry core
    registry_integration_test.go  ← integration tests
  internal/health/
    checker.go
    checker_test.go               ← unit tests for health checker
    checker_integration_test.go   ← integration tests
  internal/server/
    server.go
    server_test.go                ← unit tests for HTTP API
    server_integration_test.go    ← integration tests
  test/
    fixtures/                     ← test data files
      services.json
      health_responses.json
    helpers/                      ← test utilities
      mock_services.go
      test_registry.go
    integration/                  ← end-to-end integration tests
      milestone1_test.go
      milestone2_test.go
      milestone3_test.go
    testdata/                     ← test configuration files
      test_config.yaml
```

C. Test Infrastructure Starter Code

```go
// test/helpers/mock_services.go - Complete mock service infrastructure

package helpers

import (
    "encoding/json"
    "net/http"
    "net/http/httptest"
    "sync"
    "time"
)

// MockService represents a controllable mock service for testing
type MockService struct {
    server     *httptest.Server
    healthy    bool
    delay      time.Duration
    statusCode int
    mutex      sync.RWMutex
}

// NewMockService creates a new mock service with health endpoint
func NewMockService() *MockService {
    ms := &MockService{
        healthy:    true,
        statusCode: http.StatusOK,
    }

    mux := http.NewServeMux()
```

```
    mux.HandleFunc("/health", ms.handleHealth)
    ms.server = httptest.NewServer(mux)

    return ms
}

func (ms *MockService) handleHealth(w http.ResponseWriter, r *http.Request) {
    ms.mutex.RLock()
    defer ms.mutex.RUnlock()

    if ms.delay > 0 {
        time.Sleep(ms.delay)
    }

    if !ms.healthy {
        w.WriteHeader(http.StatusServiceUnavailable)
        json.NewEncoder(w).Encode(map[string]string{"status": "unhealthy"})
        return
    }

    w.WriteHeader(ms.statusCode)
    json.NewEncoder(w).Encode(map[string]string{"status": "healthy"})
}

// URL returns the base URL of the mock service
func (ms *MockService) URL() string {
    return ms.server.URL
}
```

```go

// SetHealthy controls the health status returned by the mock service
func (ms *MockService) SetHealthy(healthy bool) {
    ms.mutex.Lock()
    defer ms.mutex.Unlock()
    ms.healthy = healthy
}

// SetDelay adds artificial delay to health check responses
func (ms *MockService) SetDelay(delay time.Duration) {
    ms.mutex.Lock()
    defer ms.mutex.Unlock()
    ms.delay = delay
}

// Close shuts down the mock service
func (ms *MockService) Close() {
    ms.server.Close()
}
```

```go
// test/helpers/test_registry.go - Registry test utilities

package helpers

import (
    "testing"
    "time"
    "your-project/internal/registry"
)

// TestConfig returns configuration suitable for testing
func TestConfig() *registry.Config {
    return &registry.Config{
        Port: 0, // Use random port for testing
        Host: "localhost",
        HealthCheckInterval: 100 * time.Millisecond, // Fast checks for testing
        HealthTimeout: 50 * time.Millisecond,
        MaxFailures: 2,
        TTL: 5 * time.Second,
        CleanupInterval: 1 * time.Second,
    }
}

// WaitForHealthCheck waits for health checking to complete a cycle
func WaitForHealthCheck(interval time.Duration) {
    time.Sleep(interval * 3) // Wait for 3 check cycles
}

// CreateTestServiceInstance returns a valid ServiceInstance for testing
func CreateTestServiceInstance(name, host string, port int) *registry.ServiceInstance {
```

```
    return &registry.ServiceInstance{
        Name: name,
        Host: host,
        Port: port,
        Tags: []string{"test", "mock"},
        HealthEndpoint: "/health",
    }
}
```

D. Core Unit Test Skeleton

```go
// internal/registry/registry_test.go - Unit test skeleton

package registry


import (

    "testing"

    "time"

    "your-project/test/helpers"

)


// TestServiceRegistration verifies service registration logic

func TestServiceRegistration(t *testing.T) {

    // TODO 1: Create registry with test configuration

    // TODO 2: Create test ServiceInstance with valid data

    // TODO 3: Call Register method and verify success

    // TODO 4: Verify instance ID is returned and non-empty

    // TODO 5: Verify service appears in GetAllServices result

    // TODO 6: Verify service appears in Lookup result

    // Hint: Use helpers.CreateTestServiceInstance for test data

}


// TestServiceDeregistration verifies service removal logic

func TestServiceDeregistration(t *testing.T) {

    // TODO 1: Create registry and register a test service

    // TODO 2: Verify service is discoverable after registration

    // TODO 3: Call Deregister with the instance ID

    // TODO 4: Verify deregistration returns success

    // TODO 5: Verify service no longer appears in Lookup results

    // TODO 6: Verify service removed from GetAllServices results

```

```
}


// TestConcurrentRegistration verifies thread safety of registration
func TestConcurrentRegistration(t *testing.T) {
    // TODO 1: Create registry with test configuration
    // TODO 2: Launch multiple goroutines registering different services
    // TODO 3: Wait for all registrations to complete
    // TODO 4: Verify all services were registered successfully
    // TODO 5: Verify no service data was corrupted
    // TODO 6: Verify all services are discoverable
    // Hint: Use sync.WaitGroup to coordinate goroutines
}
```

E. Integration Test Skeleton

```go
// test/integration/milestone1_test.go - Milestone 1 validation

package integration

import (
    "bytes"
    "encoding/json"
    "net/http"
    "testing"
    "time"
    "your-project/internal/registry"
    "your-project/test/helpers"
)

// TestMilestone1Complete validates complete Milestone 1 functionality
func TestMilestone1Complete(t *testing.T) {
    // TODO 1: Start registry server with test configuration
    // TODO 2: Register multiple services via HTTP API
    // TODO 3: Verify each registration returns unique instance ID
    // TODO 4: Test service lookup via GET /services/{name}
    // TODO 5: Test complete service list via GET /services
    // TODO 6: Test service deregistration via DELETE
    // TODO 7: Verify deregistered services not in lookup results
    // TODO 8: Clean up test server

    t.Run("service_registration", func(t *testing.T) {
        // TODO: Implement registration test steps
    })
```

```
    t.Run("service_lookup", func(t *testing.T) {
        // TODO: Implement lookup test steps
    })

    t.Run("service_deregistration", func(t *testing.T) {
        // TODO: Implement deregistration test steps
    })
}
```

F. Language-Specific Testing Tips

- Use `httptest.NewServer()` to create test HTTP servers for integration tests
- Use `t.Parallel()` for tests that can run concurrently to speed up test execution
- Use table-driven tests for testing multiple input scenarios: `tests := []struct{name, input, expected}{...}`
- Use `testing.Short()` to skip slow integration tests during rapid development
- Use build tags like `// +build integration` to separate unit and integration tests
- Create test helpers for common setup/teardown operations to reduce duplication

G. Milestone Checkpoint Commands

Milestone 1 Validation:

```bash
# Run unit tests for registry core
go test ./internal/registry/... -v

# Run Milestone 1 integration test
go test ./test/integration/milestone1_test.go -v

# Manual verification - start server and test registration
go run cmd/server/main.go &
curl -X POST http://localhost:8080/services -d '{"name":"test-service","host":"localhost","port":9000,"tags":["test"],"healthEndpoint":"/health"}'
curl http://localhost:8080/services/test-service
```

Expected Output:

- Unit tests: All registry operations pass, no race conditions detected
- Integration test: Complete registration/lookup/deregistration workflow succeeds
- Manual test: Registration returns instance ID, lookup returns registered service

Signs of Problems:

- Registration fails with validation errors → Check ServiceInstance field validation
- Lookup returns empty results → Verify service storage in registry map
- Race condition warnings → Add proper mutex locking around shared state

H. Debugging Test Failures

Symptom	Likely Cause	Diagnosis Steps	Fix
Unit tests timeout	Infinite loop or deadlock	Add debug logging, check mutex usage	Fix locking logic, add timeouts
Integration tests fail intermittently	Race conditions or timing issues	Run tests with `-race` flag	Add proper synchronization
Mock services not responding	Port conflicts or server startup issues	Check server.URL() output, verify ports	Use httptest.NewServer for automatic ports
Health checks never complete	Health checker not started or configured incorrectly	Verify health checker Start() called	Check background goroutine startup
API tests return wrong status codes	Request formatting or handler errors	Log request/response bodies	Fix JSON marshaling or handler logic

Debugging Guide

> **Milestone(s):** All milestones (1-3) - This section provides systematic approaches to diagnose and resolve issues across service registration (Milestone 1), health checking (Milestone 2), and HTTP API operations (Milestone 3).

Mental Model: The Detective Investigation

Think of debugging a service discovery system like a detective investigating a crime scene. Just as a detective follows clues systematically - examining physical evidence, interviewing witnesses, establishing timelines, and testing theories - debugging requires a methodical approach to gather symptoms, trace root causes, and verify solutions.

The service discovery system leaves "fingerprints" everywhere: in logs, metrics, network traces, and registry state. Like a detective, you need to know where to look for these clues, how to interpret them, and how to piece together the story of what went wrong. Some problems are obvious "smoking guns" (server crashes, network failures), while others require careful investigation of subtle patterns (gradual memory leaks, race conditions, cascading failures).

Common Symptoms and Diagnosis

The first step in debugging is recognizing symptoms and mapping them to likely root causes. Service discovery systems exhibit predictable failure patterns, and understanding these patterns helps you quickly narrow down the investigation scope.

The following diagnostic approach treats each symptom as a starting point for systematic investigation. Rather than randomly checking different components, this approach guides you through logical elimination of causes based on what you observe.

Service Registration Issues

Service registration problems manifest in several distinct ways, each pointing to different underlying causes. The key is distinguishing between client-side issues (service can't register), server-side issues (registry can't process registrations), and infrastructure issues (network or resource problems).

Symptom	Likely Root Cause	Diagnostic Steps	Expected Findings
Services can't register (HTTP 500 errors)	Registry internal error, database corruption, or resource exhaustion	Check registry logs for panic/error messages, verify memory/disk usage, test registry internal state consistency	Error logs showing specific failure, high memory usage, or corrupted registry data structures
Services register but disappear immediately	TTL too short, aggressive cleanup process, or immediate deregistration	Check `Config.TTL` and `Config.CleanupInterval` settings, examine cleanup logs for premature removals	TTL expiration logs shortly after registration, cleanup process removing recent entries
Duplicate service registrations accepted	Missing instance identity validation or concurrent registration race condition	Test registering same service details multiple times, check for proper `InstanceID` generation and validation	Multiple `RegistryEntry` records with identical service details but different `InstanceID` values
Registration requests timeout	Network issues, registry overload, or blocking operations in registration path	Check network connectivity, measure registration response times, examine registry performance metrics	High response times, network packet loss, or registry CPU/memory spikes during registration
Services register with wrong metadata	Client-side configuration errors or API request construction issues	Compare actual registration requests (via logs/traces) with expected service configuration	Discrepancies between intended service configuration and actual `ServiceInstance` fields in registry

Service Discovery Issues

Service discovery problems typically involve clients unable to find services or receiving stale/incorrect service information. These issues often indicate problems with the lookup logic, health filtering, or data consistency.

Symptom	Likely Root Cause	Diagnostic Steps	Expected Findings
Clients receive empty service lists	Health checker marking all instances unhealthy, incorrect service name filtering, or registry data loss	Check health status of registered instances, verify service name matching logic, confirm registry contains expected services	All instances showing unhealthy status, service name mismatches, or empty registry state
Clients receive unhealthy service instances	Health filtering not working, stale health data, or health checker not running	Examine `HealthStatus` records for registered instances, verify health checker is active and updating status	Unhealthy instances not filtered from lookup results, outdated `LastCheck` timestamps
Service lookup returns inconsistent results	Race conditions between health updates and lookups, or concurrent modification of registry state	Test concurrent lookup requests, examine locking mechanisms around registry access, check for data races	Different results from simultaneous lookup calls, missing synchronization in registry operations
Service discovery extremely slow	Inefficient lookup algorithms, registry size issues, or blocking operations during lookup	Measure lookup response times across different registry sizes, profile lookup operations for performance bottlenecks	Lookup times increasing with registry size, blocking operations in lookup path
Clients find services that don't exist	Stale registry data, failed deregistrations, or cleanup process not running	Check for orphaned registry entries, verify deregistration is working correctly, examine cleanup process status	Registry contains entries for services that are no longer running

Health Checking Issues

Health checking problems affect service availability detection and can cause healthy services to be marked as failed or failed services to remain in the registry. These issues require examining both the health checking logic and the services being checked.

Symptom	Likely Root Cause	Diagnostic Steps	Expected Findings
Healthy services marked as unhealthy	Health check timeout too short, network latency issues, or service temporarily overloaded	Measure actual response times from health endpoints, compare with `HealthTimeout` configuration, check service load	Health endpoint response times exceeding timeout threshold, high service CPU/memory usage
Unhealthy services not removed	Health checker not running, failure threshold too high, or health check interval too long	Verify health checker background process is active, examine `MaxFailures` and `HEALTH_CHECK_INTERVAL` settings	Health checker stopped/crashed, failure counts not reaching threshold, infrequent health checks
Health checks failing with connection errors	Service health endpoints not responding, incorrect health endpoint URLs, or network connectivity issues	Test health endpoints manually (curl/wget), verify `HealthEndpoint` URLs in service registrations	Connection refused errors, invalid URLs, or network unreachability
Health checking consuming too many resources	Health check interval too aggressive, too many concurrent checks, or inefficient health check implementation	Monitor health checker CPU/memory usage, examine check frequency and concurrency settings	High resource usage correlating with health check activity, excessive concurrent HTTP requests
Health status not updating	Health checker database update failures, locking issues preventing status updates, or health checker crash/restart loop	Check health checker logs for update errors, examine registry locking mechanisms, verify health checker process stability	Database write errors, deadlocks in registry updates, frequent health checker restarts

HTTP API Issues

API problems typically manifest as request failures, incorrect responses, or performance issues. These problems require examining both the HTTP layer implementation and the underlying registry/health checker integration.

Symptom	Likely Root Cause	Diagnostic Steps	Expected Findings
API requests return HTTP 500 errors	Unhandled exceptions in request handlers, underlying registry errors, or resource exhaustion	Examine server error logs, check registry/health checker status, monitor server resource usage	Specific error messages in logs, registry operation failures, high server resource consumption
API requests hang or timeout	Blocking operations in request handlers, database deadlocks, or insufficient server concurrency	Profile request handling performance, check for blocking operations, examine server concurrency settings	Long-running operations in request path, database lock waits, server thread pool exhaustion
API returns malformed JSON responses	Serialization errors, null pointer dereferences in response construction, or data corruption	Test API responses for JSON validity, check response construction code for null handling	Invalid JSON syntax, missing fields in responses, or serialization exceptions
API authentication/validation failures	Input validation logic errors, malformed request bodies, or missing required fields	Send test requests with various input combinations, examine request validation code	Validation rejecting valid requests or accepting invalid requests
API performance degrades with load	Inefficient request handling, lack of caching, or resource contention under concurrent load	Load test API endpoints, profile performance under various request volumes	Response times increasing with concurrent requests, resource bottlenecks under load

Debugging Techniques

Effective debugging requires both reactive techniques (diagnosing problems when they occur) and proactive techniques (building observability into the system). The service discovery system has multiple layers where problems can occur, so debugging techniques must address each layer systematically.

Registry State Inspection

The registry core maintains the authoritative state of all services and their health status. When debugging service discovery issues, examining registry state provides ground truth about what the system believes to be true.

Registry Snapshot Analysis: Create diagnostic endpoints that expose complete registry state for inspection. This should include not just the service instances, but also metadata like registration timestamps, health check history, and internal counters.

Component	Inspection Method	Key Information	Warning Signs
`ServiceRegistry.services`	Export complete registry map	Service count, instance distribution, registration age	Empty registry, uneven distribution, very old or very new timestamps
`RegistryEntry` records	List all entries with full details	Instance metadata, health status, registration/update times	Missing health data, stale timestamps, inconsistent metadata
Health status data	Export `HealthStatus` for all instances	Check results, failure counts, response times	High failure counts, very old `LastCheck` times, inconsistent status
Internal metrics	Expose `RegistryStats` and counters	Registration/deregistration rates, cleanup activity	Unusual activity patterns, growing counters without corresponding business activity

State Consistency Verification: Implement validation functions that check registry state consistency. These should verify that all references are valid, timestamps are reasonable, and health data aligns with registration data.

Historical State Tracking: Maintain a circular buffer of recent registry changes to help diagnose timing issues and understand the sequence of events leading to a problem.

Health Check Tracing

Health checking involves complex interactions between the health checker, external services, and the registry. Tracing health check operations helps identify where the process breaks down.

Health Check Execution Tracing: Log detailed information about each health check operation, including timing, network operations, and decision points.

Trace Point	Information to Capture	Purpose
Check initiation	Instance ID, scheduled time, check type	Verify checks are being scheduled and initiated correctly
HTTP request	Request URL, headers, timeout settings	Confirm correct endpoints and request configuration
HTTP response	Status code, response time, response body (truncated)	Diagnose service-side issues and response validation
Health decision	Pass/fail decision, failure count updates, registry update actions	Understand how check results translate to health status
Error conditions	Network errors, timeout details, retry logic execution	Identify infrastructure and transient failure patterns

Health Check Timing Analysis: Track timing patterns to identify scheduling issues, cascade failures, or resource contention in health checking.

Synthetic Health Checks: Implement test health checks against known services to validate that the health checking mechanism itself is working correctly.

Network and HTTP Debugging

Network issues are common in distributed systems and can manifest as various service discovery problems. Network debugging techniques help isolate connectivity, performance, and protocol issues.

HTTP Request/Response Logging: Implement detailed HTTP request and response logging for both incoming API requests and outgoing health check requests. This should include headers, timing information, and error details.

Connection Pool Monitoring: Track HTTP connection pool status, including active connections, connection reuse rates, and connection errors.

Metric	Normal Range	Problem Indicators	Likely Causes
Active connections	Steady, reasonable number	Rapidly growing, very high numbers	Connection leaks, services not closing connections
Connection reuse rate	High percentage for repeated requests	Low reuse, frequent new connections	Connection pool configuration issues, network instability
Connection errors	Low, occasional errors	Frequent connection refused/timeout errors	Network issues, service overload, firewall problems
Response times	Consistent, within expected ranges	High variability, increasing trends	Network latency, service performance issues

Network Connectivity Testing: Implement diagnostic tools that test basic network connectivity to registered services independent of health checking.

Concurrent Operation Analysis

Service discovery systems must handle multiple concurrent operations safely. Race conditions, deadlocks, and resource contention can cause subtle, intermittent problems.

Concurrency Pattern Detection: Log concurrent operations with correlation IDs to identify patterns that lead to race conditions or resource contention.

Lock Contention Analysis: Monitor locking patterns in the registry to identify bottlenecks where multiple operations compete for the same resources.

Resource Usage Patterns: Track resource usage (memory, CPU, file descriptors) over time to identify leaks or resource exhaustion patterns.

Logging and Observability

Effective logging and observability are essential for both debugging problems and preventing them. The service discovery system should provide comprehensive visibility into its operations while avoiding log spam that makes debugging harder.

Structured Logging Strategy

Structured logging uses consistent formats and fields that enable automated analysis and correlation of log events across system components.

Core Logging Fields: Every log entry should include standard fields that enable correlation and filtering.

Field	Type	Purpose	Example Values
`timestamp`	ISO8601	Precise event timing	`2024-01-15T14:30:25.123Z`
`level`	String	Log severity	`INFO`, `WARN`, `ERROR`, `DEBUG`
`component`	String	Source component	`registry-core`, `health-checker`, `http-api`
`operation`	String	Operation being performed	`service-registration`, `health-check`, `service-lookup`
`instance_id`	String	Service instance identifier	`svc-web-001-uuid`
`service_name`	String	Service name being operated on	`user-service`, `payment-api`
`request_id`	String	Request correlation ID	`req-12345-abcde`
`client_addr`	String	Client IP address	`192.168.1.100:45678`

Operation-Specific Logging: Each major operation should log its lifecycle with sufficient detail for debugging.

Service Registration Logging:

- Registration initiation: Log complete `ServiceInstance` details being registered
- Validation results: Log validation failures with specific field errors
- Registration success: Log assigned `InstanceID` and next health check schedule
- Registration failure: Log specific error reasons and any partial state created

Health Check Logging:

- Check scheduling: Log when checks are scheduled and their target times
- Check execution: Log HTTP request details, response status, and timing
- Status changes: Log when service health status transitions between healthy/unhealthy
- Check failures: Log detailed error information including network errors and timeouts

Service Lookup Logging:

- Lookup requests: Log service name and any filtering criteria
- Lookup results: Log number of total vs. healthy instances returned
- Lookup performance: Log query execution time and cache hit/miss status
- Lookup errors: Log specific errors in registry access or filtering logic

Observability Metrics

Metrics provide quantitative data about system behavior that complements qualitative log information. Service discovery metrics should cover both operational health and business functionality.

System Health Metrics: Track overall system performance and resource usage.

Metric	Type	Description	Alert Thresholds
`registry_total_services`	Gauge	Number of distinct service names	Sudden drops (service outages)
`registry_total_instances`	Gauge	Number of registered service instances	Large changes (mass failures/deployments)
`registry_healthy_instances`	Gauge	Number of instances passing health checks	Drop below critical service levels
`health_check_duration_seconds`	Histogram	Time to complete health check cycles	P99 > health check interval
`api_request_duration_seconds`	Histogram	HTTP API request processing time	P95 > acceptable response time
`registry_memory_usage_bytes`	Gauge	Memory consumed by registry data	Approaching system memory limits

Business Logic Metrics: Track metrics that reflect the core functionality and user experience.

Metric	Type	Description	Business Impact
`service_registrations_total`	Counter	Total service registrations processed	Service deployment activity
`service_deregistrations_total`	Counter	Total service deregistrations processed	Service shutdown activity
`health_checks_total`	Counter	Total health checks performed	Health monitoring coverage
`failed_health_checks_total`	Counter	Health checks that failed	Service reliability issues
`service_lookup_requests_total`	Counter	Service discovery requests	Service discovery usage
`service_lookup_empty_responses_total`	Counter	Lookups returning no healthy instances	Service availability problems

Distributed Tracing Integration

When service discovery is part of a larger microservice ecosystem, distributed tracing helps understand how service discovery operations affect overall request flows.

Trace Span Creation: Create trace spans for major service discovery operations that can be correlated with application request traces.

Registry Operation Spans: Each registry operation (register, deregister, lookup) should create a trace span with relevant attributes like service name, instance count, and operation result.

Health Check Spans: Health check operations should create spans that show the relationship between health checking activity and service availability in application traces.

Cross-Service Correlation: Include trace context in health check requests so that health check failures can be correlated with application request failures.

Log Aggregation and Analysis

Effective debugging requires the ability to search, filter, and correlate log events across the entire system.

Log Correlation Patterns: Design logging to support common debugging queries.

Debugging Question	Required Log Correlation	Search Strategy
"What happened to service X?"	All operations affecting a specific service name	Filter by `service_name` field across all components
"Why did this registration fail?"	Registration attempt through completion/failure	Follow `request_id` through registration pipeline
"When did this service become unhealthy?"	Health check history for specific instance	Filter by `instance_id` in health checker logs
"What's causing API slowdowns?"	API request timing and concurrent operations	Correlate `api_request_duration` with `operation` and timestamp
"Why are services disappearing?"	Deregistration and cleanup activity	Search for deregistration events and cleanup operations

Alerting Integration: Configure alerts based on log patterns that indicate serious problems requiring immediate attention.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Structured Logging	Standard library with JSON formatting (`encoding/json`)	Structured logging library (logrus, zap)
Metrics Collection	Simple counters with periodic export	Prometheus metrics with automatic scraping
Health Tracing	Basic logging with timing information	OpenTelemetry tracing with span creation
Log Aggregation	File-based logs with log rotation	Centralized logging (ELK stack, Loki)
Debugging Interface	HTTP endpoints exposing internal state	Admin dashboard with real-time data

Debugging Infrastructure

Registry State Inspection Utilities:

```go
package debug

import (
    "encoding/json"
    "fmt"
    "net/http"
    "time"
)

// RegistryDebugInfo contains comprehensive registry state for debugging
type RegistryDebugInfo struct {
    Timestamp       time.Time                `json:"timestamp"`
    TotalServices   int                      `json:"total_services"`
    TotalInstances  int                      `json:"total_instances"`
    HealthyInstances int                     `json:"healthy_instances"`
    Services        map[string][]ServiceInfo `json:"services"`
    SystemStats     SystemStats              `json:"system_stats"`
}

type ServiceInfo struct {
    InstanceID      string        `json:"instance_id"`
    Host            string        `json:"host"`
    Port            int           `json:"port"`
    HealthStatus    string        `json:"health_status"`
    LastCheck       time.Time     `json:"last_check"`
    FailureCount    int           `json:"failure_count"`
    RegisteredAt    time.Time     `json:"registered_at"`
    ResponseTime    time.Duration `json:"response_time"`
```

```
}

type SystemStats struct {
    UptimeSeconds       int64   `json:"uptime_seconds"`
    MemoryUsageMB       float64 `json:"memory_usage_mb"`
    GoroutineCount      int     `json:"goroutine_count"`
    LastCleanupTime     time.Time `json:"last_cleanup_time"`
}

// NewDebugServer creates HTTP server for debugging endpoints
func NewDebugServer(registry *ServiceRegistry, healthChecker *HealthChecker, port int)
*http.Server {
    mux := http.NewServeMux()

    // Registry state inspection
    mux.HandleFunc("/debug/registry", func(w http.ResponseWriter, r *http.Request) {
        // TODO: Implement registry state export
        // TODO: Collect all RegistryEntry records from registry.services
        // TODO: Transform to ServiceInfo format with health status
        // TODO: Add system statistics and timing information
        // TODO: Return as JSON response
    })

    // Health checker status
    mux.HandleFunc("/debug/health-checker", func(w http.ResponseWriter, r *http.Request) {
        // TODO: Export health checker internal state
        // TODO: Include scheduled checks, recent results, timing stats
        // TODO: Show check intervals and failure thresholds
```

```
    })


    // Performance metrics
    mux.HandleFunc("/debug/metrics", func(w http.ResponseWriter, r *http.Request) {
        // TODO: Export performance counters and timing histograms
        // TODO: Include API request metrics, health check timing
        // TODO: Registry operation counts and error rates
    })


    return &http.Server{
        Addr:    fmt.Sprintf(":%d", port),
        Handler: mux,
    }
}
```

Health Check Tracing Utilities:

```go
package healthcheck


import (

    "context"

    "log/slog"

    "time"

)


// HealthCheckTracer provides detailed tracing for health check operations

type HealthCheckTracer struct {

    logger *slog.Logger

}


// TraceHealthCheck logs comprehensive information about health check execution

func (t *HealthCheckTracer) TraceHealthCheck(ctx context.Context, instanceID string,
serviceInstance ServiceInstance) *HealthCheckResult {

    startTime := time.Now()



    // TODO: Log health check initiation with instance details

    // TODO: Perform HTTP request to health endpoint with timeout

    // TODO: Measure response time and capture response details

    // TODO: Evaluate health based on response status and timing

    // TODO: Log health decision with failure count updates

    // TODO: Return HealthCheckResult with complete trace information



    return &HealthCheckResult{

        InstanceID:   instanceID,

        ServiceName:  serviceInstance.Name,

        Success:      false, // TODO: Set based on actual check result
```

```
        ResponseTime: time.Since(startTime),
        CheckTime:    startTime,
    }
}

// LogHealthTransition logs when service health status changes
func (t *HealthCheckTracer) LogHealthTransition(instanceID, serviceName string, oldStatus,
newStatus string, failureCount int) {
    t.logger.Info("service health status changed",
        slog.String("instance_id", instanceID),
        slog.String("service_name", serviceName),
        slog.String("old_status", oldStatus),
        slog.String("new_status", newStatus),
        slog.Int("failure_count", failureCount),
        slog.Time("transition_time", time.Now()),
    )
}
```

Structured Logging Implementation

```go
package logging

import (
    "context"
    "log/slog"
    "os"
    "time"
)

// ServiceDiscoveryLogger provides structured logging for all components
type ServiceDiscoveryLogger struct {
    logger *slog.Logger
}

// NewServiceDiscoveryLogger creates logger with service discovery specific configuration
func NewServiceDiscoveryLogger() *ServiceDiscoveryLogger {
    // Configure structured logging with JSON output
    handler := slog.NewJSONHandler(os.Stdout, &slog.HandlerOptions{
        Level: slog.LevelDebug,
        AddSource: true,
    })

    logger := slog.New(handler)
    return &ServiceDiscoveryLogger{logger: logger}
}

// LogServiceRegistration logs service registration with comprehensive details
```

```go
func (l *ServiceDiscoveryLogger) LogServiceRegistration(ctx context.Context, instance
ServiceInstance, result RegistrationResult, success bool) {

    fields := []any{

        slog.String("operation", "service-registration"),

        slog.String("service_name", instance.Name),

        slog.String("host", instance.Host),

        slog.Int("port", instance.Port),

        slog.String("health_endpoint", instance.HealthEndpoint),

        slog.Bool("success", success),

        slog.Time("timestamp", time.Now()),

    }



    if success {

        fields = append(fields,

            slog.String("instance_id", result.InstanceID),

            slog.Time("next_health_check", result.NextHealthCheck),

            slog.Time("ttl_expiration", result.TTLExpiration),

        )

        l.logger.Info("service registered successfully", fields...)

    } else {

        // TODO: Add error details from registration failure

        l.logger.Error("service registration failed", fields...)

    }

}


// LogHealthCheck logs health check execution with timing and result details

func (l *ServiceDiscoveryLogger) LogHealthCheck(instanceID string, result
HealthCheckResult) {
```

```
    l.logger.Info("health check completed",
        slog.String("operation", "health-check"),
        slog.String("instance_id", instanceID),
        slog.String("service_name", result.ServiceName),
        slog.Bool("success", result.Success),
        slog.Duration("response_time", result.ResponseTime),
        slog.Int("status_code", result.StatusCode),
        slog.Time("check_time", result.CheckTime),
        slog.String("error", result.Error),
    )
}

// LogServiceLookup logs service discovery requests and results
func (l *ServiceDiscoveryLogger) LogServiceLookup(serviceName string, result LookupResult)
{
    l.logger.Info("service lookup completed",
        slog.String("operation", "service-lookup"),
        slog.String("service_name", serviceName),
        slog.Int("total_instances", result.TotalCount),
        slog.Int("healthy_instances", result.HealthyCount),
        slog.Bool("cache_hit", result.CacheHit),
        slog.Duration("lookup_duration", result.LookupDuration),
        slog.Time("last_updated", result.LastUpdated),
    )
}
```

Milestone Checkpoint: Debugging Verification

After implementing debugging infrastructure:

Verify Registry State Inspection:

1. Start the service registry with debug endpoints enabled
2. Register several test services using the HTTP API
3. Access `GET /debug/registry` - should return complete registry state as JSON
4. Verify all registered services appear with correct metadata
5. Register an unhealthy service - verify it appears in debug output with health status

Verify Health Check Tracing:

1. Register services with valid and invalid health endpoints
2. Wait for health check cycles to complete
3. Check logs for structured health check trace entries
4. Verify health transitions are logged when services become unhealthy
5. Confirm health check timing and error details are captured

Verify Structured Logging:

1. Perform various operations (register, lookup, deregister services)
2. Verify all operations produce structured JSON log entries
3. Confirm log entries include correlation fields (request_id, instance_id)
4. Test log filtering by component and operation type
5. Verify error conditions produce appropriate error-level log entries

Signs that debugging infrastructure is working correctly:

- Debug endpoints return comprehensive, accurate system state
- All major operations produce structured log entries with correlation IDs
- Health check failures include detailed error information and timing
- Log entries can be easily filtered and searched by service, operation, or time range
- System metrics show accurate counts of services, instances, and operations

Future Extensions

> **Milestone(s):** All milestones (1-3) - This section outlines potential enhancements that build upon the service registry foundation (Milestone 1), health checking capabilities (Milestone 2), and HTTP API (Milestone 3).

Mental Model: The Growing City

Think of your service registry as a small town that's experiencing rapid growth. Right now, you have the basic infrastructure: a town hall (registry core) where residents register, a security patrol (health checker) that

checks on everyone, and an information desk (HTTP API) that helps people find what they need. As the town grows into a city, you'll need additional infrastructure: multiple town halls working together (clustering), security systems (authentication and authorization), traffic management (load balancing), and specialized services (advanced health checks).

Just as a city planner designs infrastructure that can be added incrementally without rebuilding the entire town, your current service registry architecture provides extension points that accommodate these enhancements without requiring fundamental rewrites. Each future extension represents a new city department that integrates with your existing infrastructure while adding specialized capabilities.

The current design's modularity means these extensions can be added incrementally - you might start with simple authentication, then add clustering support, and later introduce advanced health checking protocols. Each enhancement builds upon the solid foundation you've established while extending the system's capabilities to handle larger scale and more sophisticated requirements.

Registry Clustering and High Availability

The most significant limitation of the current single-node design is that it represents a single point of failure. In production environments, service discovery systems must remain available even when individual nodes fail, network partitions occur, or maintenance activities require taking nodes offline.

A clustered registry extension would distribute service registration data across multiple nodes, providing both high availability and horizontal scalability. This extension builds naturally on the current `ServiceRegistry` design by adding a clustering layer that manages data replication and consensus decisions across multiple registry instances.

Registry Cluster Architecture

The clustering extension introduces several new components that integrate with your existing registry core. A `ClusterManager` component would coordinate communication between registry nodes, while a `ConsensusEngine` ensures all nodes agree on service registration changes. The existing `ServiceRegistry` becomes a cluster member that participates in distributed decision-making while maintaining its current interface for local operations.

Data consistency across cluster nodes requires implementing a consensus protocol like Raft or using distributed key-value stores like etcd or Consul as the backing storage. This extension point exists because your current registry design already abstracts service data through clear interfaces - adding clustering means implementing a new storage backend that happens to be distributed rather than local.

The clustering extension would introduce these additional data structures:

Structure Name	Fields	Purpose
`ClusterNode`	NodeID string, Address string, Role string, Status string, LastSeen time.Time	Represents a registry node in the cluster
`ClusterConfig`	Nodes []ClusterNode, ReplicationFactor int, ConsensusTimeout time.Duration	Configuration for cluster behavior
`ReplicationEntry`	Operation string, Data interface{}, Timestamp time.Time, NodeID string	Represents a change to replicate across nodes
`ConsensusResult`	Success bool, LeaderID string, Term int64, CommittedIndex int64	Result of consensus operation

Cluster Operations and Consensus

Service registration in a clustered environment requires consensus among nodes to ensure all cluster members have consistent views of available services. When a service registers with any cluster node, that node proposes the registration to other cluster members through the consensus protocol.

The consensus process follows these steps:

1. A client sends a service registration request to any cluster node
2. The receiving node validates the registration request locally
3. The node proposes the registration operation to the cluster leader
4. The leader replicates the operation to a majority of follower nodes
5. Once a majority confirms the operation, the leader commits it
6. All nodes apply the committed registration to their local registries
7. The original node responds to the client with success

This approach ensures that service registrations are consistent across the cluster even if individual nodes fail during the registration process. The existing `Register` method signature remains unchanged - clustering becomes an internal implementation detail that doesn't affect the public API.

Partition Tolerance and Split-Brain Prevention

Network partitions represent one of the most challenging aspects of distributed systems. When cluster nodes cannot communicate with each other, they must decide whether to continue accepting registrations (risking inconsistency) or reject them (sacrificing availability).

The clustering extension would implement partition tolerance through quorum-based decision making. A cluster partition can continue operating only if it contains a majority of nodes, preventing split-brain scenarios where multiple partitions each believe they're authoritative.

During a network partition, the minority partition would enter read-only mode, allowing service lookups but rejecting new registrations. This design choice prioritizes consistency over availability during partition events,

which aligns with service discovery requirements where returning stale data is generally acceptable, but accepting inconsistent registrations causes serious problems.

> **Design Insight**: Service discovery systems typically favor consistency over availability during network partitions because returning slightly stale service information is much less problematic than having different parts of your infrastructure believe different services are available.

Health Checking in Clustered Environments

Clustering introduces complexity to health checking because multiple nodes might attempt to health check the same service instances. The extension would implement distributed health checking coordination to avoid overwhelming services with redundant health checks while ensuring comprehensive coverage.

A `DistributedHealthChecker` component would extend the existing `HealthChecker` by adding coordination logic that divides health checking responsibilities among cluster nodes. This could be implemented through consistent hashing, where each service instance is assigned to specific cluster nodes for health checking, or through leader-based coordination where a single node schedules all health checks but distributes the actual checking work.

Security and Authentication

The current HTTP API accepts all requests without authentication or authorization, which is acceptable for development environments but inadequate for production deployments. A security extension would add multiple layers of protection while maintaining the existing API structure.

Service Authentication and Authorization

The security extension introduces authentication for both service registration and service lookup operations. Services registering themselves would need to prove their identity, while clients looking up services would need appropriate permissions to access service information.

The extension would add these security-related data structures:

Structure Name	Fields	Purpose
`ServiceIdentity`	ServiceName string, Credentials string, Permissions []string, ExpiresAt time.Time	Authentication information for a service
`AccessToken`	TokenID string, ServiceID string, Permissions []string, IssuedAt time.Time, ExpiresAt time.Time	Bearer token for API access
`SecurityPolicy`	AllowedServices []string, RequiredPermissions map[string] []string, TokenTTL time.Duration	Security configuration
`AuditLogEntry`	Timestamp time.Time, ServiceID string, Operation string, Resource string, Success bool, ClientIP string	Security audit trail

Authentication could be implemented through multiple mechanisms: API keys for simple deployments, mutual TLS for high-security environments, or integration with external identity providers like OAuth 2.0 or service mesh identity systems.

The security extension would modify existing HTTP API endpoints to require authentication headers and validate permissions before processing requests. Service registration would require "register" permission for the specific service name, while service lookup might require "discover" permission for the target service.

Mutual TLS and Certificate Management

For high-security environments, the extension would support mutual TLS (mTLS) authentication where both the registry and connecting services verify each other's certificates. This provides strong cryptographic identity verification and encrypted communication channels.

The mTLS extension would add certificate management capabilities to automatically issue, renew, and revoke certificates for registered services. This could integrate with certificate authorities like HashiCorp Vault or cloud provider certificate services to provide a complete PKI solution.

Audit Logging and Compliance

Security extensions require comprehensive audit logging to track who accessed what services when. The extension would maintain detailed audit trails of all registry operations, including failed authentication attempts, successful service lookups, and administrative operations.

Audit logs would capture sufficient information for compliance requirements and security incident investigation while being structured for automated analysis and alerting on suspicious patterns.

Advanced Health Checking Protocols

The current health checking implementation supports only HTTP endpoints with basic success/failure detection. Production environments often require more sophisticated health checking that can integrate with various protocols and provide nuanced health status information.

Multi-Protocol Health Checking

The advanced health checking extension would support multiple protocols beyond HTTP, including TCP socket checks, gRPC health checking protocol, database connection verification, and custom script-based checks.

Each protocol would implement a common `HealthCheckProtocol` interface while providing protocol-specific configuration options:

Protocol Type	Configuration Fields	Use Cases
HTTP	Path string, Headers map[string]string, ExpectedStatus []int, Timeout time.Duration	REST APIs, web services
TCP	Port int, Timeout time.Duration	Database connections, message queues
gRPC	ServiceName string, Timeout time.Duration	gRPC microservices
Script	Command string, Args []string, Timeout time.Duration, Environment map[string]string	Custom application logic

The extension would maintain the existing `HealthChecker` interface while adding protocol-specific implementations that can be configured per service instance. A service might use HTTP health checks for its API endpoint while also performing TCP checks on its database connection.

Composite Health Status

Advanced health checking introduces the concept of composite health status where a service instance reports multiple health dimensions. Instead of simple healthy/unhealthy status, services could report separate status for different subsystems.

For example, a web service might report:

- API endpoint: healthy
- Database connection: degraded
- Cache connection: unhealthy
- Disk space: healthy

The composite status extension would allow services to continue receiving traffic for operations that don't require unhealthy subsystems while being removed from routing for operations that do require them.

Adaptive Health Checking

Instead of fixed check intervals, adaptive health checking adjusts check frequency based on service behavior patterns. Services that consistently pass health checks are checked less frequently, while services showing instability receive more frequent monitoring.

The adaptive extension would implement algorithms that:

1. Increase check frequency when services show signs of instability
2. Reduce check frequency for consistently healthy services
3. Implement exponential backoff for services that are clearly failed
4. Provide burst checking during recovery periods

This approach reduces system overhead while providing faster detection of service issues when they occur.

Health Check Circuit Breakers

When services become overloaded, aggressive health checking can make the situation worse by adding additional load. The circuit breaker extension would automatically reduce health check frequency or temporarily disable checks for services showing signs of overload.

The circuit breaker monitors health check response times and error rates, opening the circuit when thresholds are exceeded. During circuit-open periods, the registry assumes the service is unhealthy without performing actual checks, allowing the service to recover from overload conditions.

Load Balancing and Traffic Management

The current registry returns all healthy service instances without providing guidance on how clients should distribute traffic among them. A traffic management extension would add intelligent load balancing recommendations and traffic steering capabilities.

Weighted Load Balancing

Services instances often have different capacities - some might run on larger machines or have better network connectivity. The load balancing extension would allow services to register with weight information that influences traffic distribution.

The extension would add weight-aware service lookup that returns instances along with recommended traffic distribution ratios. Clients could use this information to implement weighted random selection or other load balancing algorithms that respect instance capacities.

Geographic and Zone-Aware Routing

In distributed deployments, services should prefer communicating with nearby instances to reduce latency and improve reliability. The geographic extension would add location awareness to service registration and intelligent proximity-based routing.

Services would register with zone, region, and availability zone information. Client lookups could specify their own location and receive service instances sorted by proximity, with fallback to remote instances if local ones are unavailable.

Traffic Splitting and Canary Deployments

The traffic management extension would support sophisticated deployment patterns like canary releases and A/B testing by allowing services to register with version information and traffic splitting rules.

A service deployment might register both v1.0 and v1.1 instances, with routing rules that direct 90% of traffic to v1.0 and 10% to v1.1. The registry would return instances according to these ratios, enabling gradual rollouts with automatic traffic distribution.

Service Mesh Integration

Modern microservice deployments increasingly use service mesh technologies like Istio, Linkerd, or Consul Connect. A service mesh integration extension would allow your registry to interoperate with these systems while providing additional service discovery capabilities.

Sidecar Proxy Integration

Service mesh deployments typically include sidecar proxies that handle service-to-service communication. The integration extension would support registering services through their sidecar proxies and coordinating with the service mesh control plane for traffic routing decisions.

The extension would add support for service mesh protocols like Envoy's xDS (discovery service) protocol, allowing the registry to act as a data source for service mesh control planes. This enables hybrid deployments where some services use direct registry integration while others integrate through service mesh infrastructure.

Policy Integration and Enforcement

Service meshes provide sophisticated traffic policies for security, reliability, and observability. The integration extension would respect service mesh policies when returning service discovery information, ensuring that registry responses align with mesh-level traffic management rules.

For example, if the service mesh has configured circuit breakers or rate limits for specific service pairs, the registry integration would reflect this in service discovery responses, potentially filtering out instances that would violate mesh policies.

Observability and Metrics

Production service discovery systems require comprehensive observability for monitoring, alerting, and capacity planning. The observability extension would add detailed metrics, distributed tracing, and operational dashboards.

Metrics and Monitoring

The extension would expose detailed metrics about registry operations, health check performance, and service topology changes. These metrics would integrate with monitoring systems like Prometheus, providing insights into:

- Service registration and deregistration rates
- Health check success rates and response times
- API request patterns and error rates
- Registry resource utilization and performance

The metrics would enable operational teams to understand registry behavior, detect performance issues, and plan capacity requirements for growing service deployments.

Distributed Tracing

For complex service interactions, distributed tracing provides insights into how service discovery operations contribute to overall request latency. The tracing extension would add trace spans for registry operations, allowing correlation with application traces.

This enables debugging scenarios where application performance issues might be related to service discovery latency, health check delays, or registry overload conditions.

Configuration Management and Dynamic Updates

The current system uses static configuration loaded at startup. A dynamic configuration extension would allow runtime updates to registry behavior without requiring restarts, enabling operational flexibility and rapid response to changing conditions.

Hot Configuration Updates

The extension would support updating health check intervals, timeout values, cleanup frequencies, and other operational parameters without service interruption. This capability is essential for production systems where configuration changes might be needed to respond to traffic patterns or performance issues.

Configuration updates would be validated before application and provide rollback capabilities if updates cause problems. The system would maintain configuration history and support gradual rollout of configuration changes across clustered deployments.

Service-Specific Configuration

Different services often have different requirements for health checking frequency, timeout values, and failure thresholds. The extension would support service-specific configuration overrides that allow fine-tuning registry behavior for individual services or service categories.

For example, critical services might have more aggressive health checking, while batch processing services might use longer timeouts and higher failure thresholds before being marked unhealthy.

Implementation Roadmap

These extensions can be implemented incrementally, with each building upon the current architecture foundation. The recommended implementation order considers dependencies and operational value:

Phase 1: Security and Observability

- Basic authentication and authorization
- Comprehensive metrics and monitoring
- Audit logging capabilities

Phase 2: Advanced Health Checking

- Multi-protocol health checking
- Adaptive check intervals
- Composite health status

Phase 3: High Availability

- Registry clustering
- Distributed health checking
- Partition tolerance

Phase 4: Traffic Management

- Load balancing recommendations
- Geographic routing
- Service mesh integration

Each phase provides immediate operational value while preparing the foundation for subsequent enhancements. The modular architecture ensures that extensions can be developed and deployed independently without disrupting existing functionality.

Architecture Compatibility

The current design's emphasis on clear interfaces and component separation makes it particularly well-suited for these extensions. The `ServiceRegistry` interface can accommodate clustered implementations, the `HealthChecker` component can support multiple protocols, and the HTTP API layer can add authentication middleware without changing core logic.

Most importantly, the extension points preserve backward compatibility, allowing existing clients to continue working while new clients take advantage of enhanced capabilities. This approach supports gradual migration and mixed-environment deployments where some services use basic features while others require advanced functionality.

The modular design also supports partial deployment of extensions - an organization might implement security features immediately while deferring clustering until scale requirements justify the additional complexity. Each extension stands alone while integrating seamlessly with the core system and other extensions.

Glossary

> **Milestone(s):** All milestones (1-3) - This glossary provides definitions for terminology used across service registration (Milestone 1), health checking (Milestone 2), and HTTP API operations (Milestone 3).

Mental Model: The Reference Manual

Think of this glossary as a reference manual for a complex piece of machinery. When you're working with an industrial system like a manufacturing robot, you need a comprehensive manual that defines every component, every operation, and every technical term. The manual doesn't just list parts—it explains how they work, why they're designed that way, and how they interact with other components. Similarly, this glossary

serves as your reference manual for the service discovery system, providing not just definitions but context for how each concept fits into the larger architecture.

Core System Components

The service discovery system consists of several fundamental components that work together to maintain an accurate view of service availability across a distributed system.

Component	Definition	Responsibilities
Service Registry	Central database that maintains information about all available service instances	Stores service metadata, tracks instance identity, manages registration lifecycles, provides lookup operations
Registry Core	The central component managing service data storage and retrieval operations	Handles concurrent access to service data, enforces TTL policies, manages instance identity mapping
Health Checker	Background component that continuously monitors service availability and responsiveness	Schedules periodic health verifications, processes health check results, updates registry state based on service health
HTTP API Layer	REST interface that exposes registry operations to services and clients over HTTP	Validates incoming requests, serializes responses, handles concurrent requests, provides standardized error responses
Flow Coordinator	Component that orchestrates complex multi-step operations across system components	Manages service registration workflows, coordinates health check cycles, handles rollback operations

Service Discovery Fundamentals

Service discovery encompasses the mechanisms and patterns that enable microservices to locate and communicate with each other dynamically in distributed environments.

Term	Definition	Key Characteristics
Service Discovery	Mechanism for services to find and communicate with each other without hardcoded network locations	Dynamic resolution, automatic updates, health-aware routing, location transparency
Service Instance	Single running instance of a service that can handle requests	Has unique identity, network endpoint, health status, metadata tags for routing decisions
Instance Identity	Unique identifier assigned to each service registration that distinguishes it from other instances	Persistent across health status changes, used for deregistration, enables targeted operations
Service Registry Pattern	Architectural pattern where services register themselves in a central database for others to discover	Centralized service metadata, dynamic registration/deregistration, health-aware lookups
Lookup By Name	Operation to find all healthy instances of a service using its logical name	Returns filtered list of instances, includes load balancing metadata, excludes unhealthy instances

Health Monitoring and Status Management

Health checking is the continuous process of verifying service availability and removing failed instances from active service routing.

Term	Definition	Implementation Details
Health Checking	Process of monitoring service availability and responsiveness through periodic verification	Uses HTTP, TCP, or gRPC protocols, configurable intervals, failure thresholds, timeout handling
Health Status	Current availability state of a service instance	States: healthy, unhealthy, unknown; includes failure count, last check time, response metrics
Health Endpoint	HTTP endpoint that reports service health status, typically `/health` or `/status`	Returns status codes indicating health, may include detailed diagnostics, responds within timeout limits
Failure Detection	Process of identifying when a service instance is no longer able to handle requests properly	Based on consecutive failures, network timeouts, HTTP error codes, connection refusals
Failure Threshold	Number of consecutive failed health checks before marking an instance as unhealthy	Configurable per service, prevents false positives from transient failures, triggers automatic deregistration
Check Interval	Time between health verification attempts for a service instance	Configurable per service type, balances freshness with system load, may use adaptive scheduling
Check Scheduling	Timing coordination of health verification operations across all registered instances	Prevents thundering herd effects, distributes load evenly, handles dynamic instance populations
Failure Count	Number of consecutive failed health checks for a service instance	Reset on successful check, used to determine unhealthy status, tracked per instance

Data Structures and Types

The system uses specific data structures to represent services, health information, and system state consistently across all components.

Type	Purpose	Key Fields
ServiceInstance	Represents a single service instance with its network location and metadata	Name (service type), Host (IP/hostname), Port (network port), Tags (routing metadata), HealthEndpoint (monitoring URL)
HealthStatus	Tracks current health state and historical information for an instance	Status (current state), LastCheck (timestamp), FailureCount (consecutive failures), ResponseTime (latency), LastError (failure details)
RegistryEntry	Complete registry record combining instance data with health information and metadata	Instance (service data), Health (status info), RegisteredAt (timestamp), InstanceID (unique identifier)
ServiceList	Response format for service lookup operations containing filtered results	ServiceName, Instances (healthy only), TotalCount (all instances), HealthyCount (available instances)
Config	System configuration containing operational parameters and thresholds	Port, Host, HealthCheckInterval, HealthTimeout, MaxFailures, TTL, CleanupInterval

Registration and Lifecycle Management

Service registration encompasses the processes by which services announce their availability and manage their presence in the registry.

Term	Definition	Lifecycle Phase
Service Registration	Process by which a service instance announces its availability to the registry	Initialization: instance provides metadata, receives unique ID, begins health monitoring
Service Deregistration	Process of removing a service instance from the registry, either explicitly or automatically	Cleanup: instance removed from lookup results, health checking stopped, resources freed
TTL (Time-To-Live)	Maximum duration a service registration remains valid without renewal	Automatic cleanup: prevents stale entries, requires periodic heartbeats, configurable per service
TTL Handling	Automatic cleanup mechanism that removes expired service registrations	Background process: scans for expired entries, removes from active registry, logs cleanup actions
Registration Result	Response provided when a service successfully registers with the registry	Contains: InstanceID, HealthEndpoint, NextHealthCheck time, RegistrationTime, TTLExpiration
Heartbeat Timeout	Maximum time between service heartbeats before automatic deregistration	Configurable threshold: balances responsiveness with tolerance for temporary issues

Network Operations and Protocols

The system supports multiple protocols and handles various network conditions that affect service communication and monitoring.

Term	Definition	Protocol Support
HTTP Health Check	Health verification using HTTP requests to service endpoints	GET/POST requests, status code evaluation, timeout handling, custom headers support
TCP Health Check	Basic connectivity verification using TCP socket connections	Port connectivity, connection establishment, minimal overhead, network-level verification
gRPC Health Check	Health verification using gRPC health check protocol standard	Structured health reporting, service-specific status, streaming support, rich error information
Network Partition	Loss of network connectivity between system components	Split-brain scenarios: isolated segments, conflicting state, partition detection, recovery coordination
Network Timeout	Failure to complete network operations within specified time limits	Connection timeouts, read timeouts, configurable per operation, affects health status determination
Connection Pool	Managed collection of reusable network connections for health checking	Resource optimization: connection reuse, concurrent limits, idle connection management, health monitoring

API Operations and HTTP Interface

The HTTP API provides standardized operations for service registration, discovery, and management through REST endpoints.

Term	Definition	HTTP Details
REST API (Representational State Transfer API)	HTTP-based interface following REST principles for resource manipulation	Stateless operations, resource-based URLs, standard HTTP methods, JSON content type
JSON Serialization	Converting data structures to JavaScript Object Notation format for HTTP transport	Request/response bodies, standardized field names, error response formats, content-type headers
Request Validation	Verifying input data meets requirements before processing	Field presence checks, type validation, range constraints, format verification
Error Response	Standardized format for communicating operation failures to clients	HTTP status codes, error messages, error codes, additional context details
Success Response	Standardized format for communicating successful operation results	Success flag, response data, confirmation messages, operation metadata
Middleware	Software components that process requests before reaching main handlers	Cross-cutting concerns: logging, authentication, rate limiting, request validation

Concurrency and Thread Safety

The system handles multiple simultaneous operations safely through proper synchronization and concurrency control mechanisms.

Term	Definition	Implementation Approach
Concurrent Access	Multiple operations accessing shared data simultaneously without conflicts	Read-write locks, atomic operations, synchronized access patterns, data isolation
Concurrent Requests	Multiple simultaneous API operations being processed by the system	Request isolation, resource sharing, response ordering, error propagation
Race Conditions	Timing-dependent bugs in concurrent operations where results depend on execution order	Prevention through locks, atomic operations, immutable data structures, careful state management
Thread Safety	Guarantee that shared data structures can be accessed safely from multiple threads	Synchronization primitives, lock-free algorithms, immutable data, isolated state

Error Handling and Resilience Patterns

The system implements comprehensive error handling and resilience patterns to maintain availability despite various failure modes.

Term	Definition	Implementation Strategy
Error Handling	Managing and responding to failure conditions throughout the system	Structured error types, recovery procedures, client communication, operational logging
Graceful Degradation	Selective reduction of service quality to maintain core functionality during stress	Feature prioritization, resource management, partial functionality, user communication
Failure Isolation	Preventing problems from spreading to other system components	Component boundaries, timeout limits, circuit breakers, resource isolation
Circuit Breaker	Pattern for preventing calls to failing services to avoid cascading failures	Failure counting, automatic tripping, recovery detection, fallback responses
Cascade Failure	Failure of one service triggering failures in dependent services	Prevention through isolation, timeout limits, graceful degradation, monitoring
False Positive	Incorrectly identifying healthy service as failed due to monitoring issues	Network issues, temporary overload, check timeout too short, monitoring system problems
Progressive Timeout Handling	Graduated response to timeouts based on failure patterns and system state	Adaptive timeouts, escalating delays, failure pattern recognition, recovery coordination

Load Management and System Protection

The system includes mechanisms to protect itself from overload and manage resource consumption effectively.

Term	Definition	Protection Mechanism
Rate Limiting	Controlling the frequency of requests to prevent system overload	Token bucket algorithm, per-client limits, global limits, burst handling
System Overload	Condition where system resources are overwhelmed by demand	CPU/memory thresholds, connection limits, queue depths, response time degradation
Token Bucket Algorithm	Rate limiting technique using token consumption model for request throttling	Token generation rate, bucket capacity, burst allowance, deficit handling
Adaptive Rate Limiting	Dynamic request limiting based on current system performance and health	Performance monitoring, dynamic threshold adjustment, client prioritization, feedback loops
Backpressure	Mechanism to slow down request rates when downstream systems are overwhelmed	Queue management, flow control, client notification, resource allocation

Monitoring and Observability

Comprehensive monitoring and observability enable understanding system behavior and diagnosing issues effectively.

Term	Definition	Implementation Details
Structured Logging	Logging with consistent formats and fields for automated analysis	JSON format, standard fields, correlation IDs, log levels, automated parsing
Distributed Tracing	Tracking request flows across multiple services to understand system behavior	Trace IDs, span relationships, timing information, error propagation
Correlation ID	Unique identifier linking related operations across components for troubleshooting	Request tracking, log aggregation, performance analysis, error correlation
Log Aggregation	Collecting and centralizing log data from multiple sources for analysis	Centralized storage, search capabilities, alerting, dashboard creation
Observability	Ability to understand system internal state from external outputs	Metrics, logs, traces, health indicators, performance data
Health Check Tracing	Detailed logging of health verification operations for debugging and monitoring	Check timing, failure reasons, network details, service responses

Testing and Quality Assurance

Comprehensive testing strategies ensure system reliability and correct behavior under various conditions.

Term	Definition	Testing Scope
Unit Testing	Testing individual components in isolation with mocks and test data	Component functionality, error conditions, boundary cases, interface contracts
Integration Testing	End-to-end workflows and component interaction testing	Multi-component flows, real network operations, external service integration
Milestone Validation	Comprehensive checkpoints verifying complete milestone functionality	Feature completeness, acceptance criteria, integration points, performance requirements
Test Doubles	Mock objects and stubs used to replace dependencies in testing	Controllable behavior, predictable responses, error injection, isolation support
Component Isolation	Testing strategy focusing on single components without external dependencies	Dependency mocking, controlled inputs, predictable environment, focused validation
Mock Service	Controllable test service that simulates real service behavior for testing	Health status control, response delays, error injection, realistic endpoints
Load Testing	Verification of system performance under high volume conditions	Concurrent requests, resource utilization, response times, failure rates
Failure Injection	Deliberately causing failures to test error handling and recovery mechanisms	Network failures, service unavailability, resource exhaustion, timeout conditions

Advanced Concepts and Extensions

These concepts represent potential system extensions and advanced features that build upon the core service discovery functionality.

Term	Definition	Extension Area
Clustering	Distributing registry data across multiple nodes for high availability	Multi-node deployment, data replication, consensus protocols, failover handling
Consensus Protocol	Algorithm ensuring distributed nodes agree on data changes	Distributed consistency, leader election, log replication, split-brain prevention
Service Mesh	Infrastructure layer handling service communication with advanced features	Traffic management, security policies, observability, protocol translation
Sidecar Proxy	Auxiliary process deployed alongside main service for communication handling	Service abstraction, policy enforcement, metrics collection, traffic routing
Mutual TLS	Authentication where both parties verify each other's certificates	Service identity, encrypted communication, certificate management, trust establishment
Weighted Load Balancing	Traffic distribution based on instance capacity and performance characteristics	Instance capabilities, traffic distribution, performance-based routing, capacity awareness
Canary Deployment	Gradual rollout strategy using traffic splitting between service versions	Version management, traffic splitting, rollback capabilities, risk mitigation
Hot Configuration Updates	Runtime configuration changes without service restart	Dynamic reconfiguration, configuration validation, rollback support, change notification

Performance and Scalability Terms

These terms relate to system performance characteristics and scaling behavior under various load conditions.

Term	Definition	Performance Impact
Response Time	Time taken to complete an operation from request to response	User experience, system capacity, resource utilization, scalability limits
Throughput	Number of operations completed per unit time	System capacity, resource efficiency, scaling characteristics, bottleneck identification
Latency	Delay between operation initiation and first response	Real-time requirements, user perception, cascade effects, system responsiveness
Cache Hit Ratio	Percentage of requests served from cache rather than original source	Performance optimization, resource usage, data freshness, system load
Resource Utilization	Percentage of available system resources currently in use	Capacity planning, performance optimization, scaling decisions, efficiency measurement
Scalability	System's ability to handle increased load through resource addition	Horizontal scaling, vertical scaling, bottleneck identification, capacity planning

Security and Access Control

Security-related terms that apply to protecting the service registry and controlling access to registry operations.

Term	Definition	Security Aspect
Service Identity	Cryptographic or token-based identification of services for access control	Authentication, authorization, service verification, trust establishment
Access Token	Credential that grants specific permissions for registry operations	Authorization, permission scoping, expiration handling, token management
Security Policy	Rules defining allowed operations and access patterns for different service types	Access control, permission management, policy enforcement, compliance requirements
Audit Log	Record of all security-relevant operations for compliance and investigation	Security monitoring, compliance verification, incident investigation, access tracking

Implementation Guidance

This subsection provides practical implementation support for developers working with the service discovery system terminology and concepts.

Technology Recommendations

Component	Simple Option	Advanced Option
Documentation	Inline comments + README	Automated API docs with Swagger/OpenAPI
Terminology Management	Static glossary file	Interactive documentation with search
Code Documentation	Standard language docs	Rich documentation with examples and links

Recommended File Structure

```
project-root/
  docs/
    glossary.md              ← comprehensive terminology reference
    architecture.md          ← system overview with terminology
    api-reference.md         ← endpoint documentation with consistent terms
  internal/
    types/
      service.go            ← ServiceInstance, HealthStatus definitions
      registry.go           ← RegistryEntry, ServiceList definitions
      api.go               ← ErrorResponse, SuccessResponse definitions
      config.go            ← Config, system parameter definitions
    glossary/
      terms.go             ← programmatic access to terminology
      validation.go        ← terminology consistency checking
```

Terminology Consistency Utilities

```go
// Package glossary provides programmatic access to system terminology
package glossary

// TermDefinition represents a single glossary entry
type TermDefinition struct {
    Term        string   `json:"term"`
    Definition  string   `json:"definition"`
    Category    string   `json:"category"`
    Aliases     []string `json:"aliases"`
    SeeAlso     []string `json:"see_also"`
    Examples    []string `json:"examples"`
}

// GlossaryManager provides terminology validation and lookup
type GlossaryManager struct {
    terms map[string]TermDefinition
    categories map[string][]string
}

// ValidateTerminology checks that code comments and documentation use consistent terminology
func (g *GlossaryManager) ValidateTerminology(text string) []TerminologyIssue {
    // TODO: Scan text for terminology usage
    // TODO: Check against preferred terms vs aliases
    // TODO: Flag inconsistent usage
    // TODO: Suggest corrections for common mistakes
    return nil
}
```

```
// GetDefinition retrieves the definition for a specific term
func (g *GlossaryManager) GetDefinition(term string) (TermDefinition, bool) {
    // TODO: Lookup term in glossary
    // TODO: Handle case-insensitive matching
    // TODO: Return definition with metadata
    return TermDefinition{}, false
}
```

Documentation Generation Utilities

```go
// DocumentationGenerator creates API docs with consistent terminology
type DocumentationGenerator struct {
    glossary *GlossaryManager
    templates map[string]string
}

// GenerateAPIDocumentation creates endpoint documentation with terminology links
func (d *DocumentationGenerator) GenerateAPIDocumentation(endpoints []APIEndpoint) string {
    // TODO: Generate endpoint documentation
    // TODO: Link technical terms to glossary definitions
    // TODO: Include request/response examples with proper terminology
    // TODO: Validate terminology consistency across docs
    return ""
}

// GenerateGlossaryHTML creates interactive HTML glossary
func (d *DocumentationGenerator) GenerateGlossaryHTML(categories []string) string {
    // TODO: Generate HTML with search functionality
    // TODO: Group terms by category
    // TODO: Create cross-references between related terms
    // TODO: Include usage examples and code snippets
    return ""
}
```

Language-Specific Hints

Go Implementation Notes:

- Use `go doc` to generate documentation that references glossary terms consistently

- Create custom types for domain concepts (e.g., `type InstanceID string`) to enforce terminology in code
- Use struct tags to ensure JSON serialization uses glossary-defined field names
- Implement `String()` methods that use proper terminology in debug output

Terminology Validation:

- Run terminology consistency checks as part of CI/CD pipelines
- Use linters to enforce preferred terms over aliases in code comments
- Generate glossary updates automatically from code documentation
- Validate API documentation against glossary definitions

Milestone Checkpoints

After completing the glossary:

- All project documentation uses consistent terminology
- Code comments reference glossary terms appropriately
- API documentation includes links to relevant definitions
- Team members understand domain-specific vocabulary

Validation Commands:

```bash
# Check terminology consistency in documentation
go run ./cmd/glossary-check ./docs/

# Generate API documentation with glossary links
go run ./cmd/doc-generator -output ./docs/api-reference.md

# Validate code comments against glossary
go run ./cmd/terminology-lint ./internal/...
```

Expected Behavior:

- Documentation uses preferred terms consistently (not aliases)
- Technical discussions reference common vocabulary
- New team members can quickly understand domain concepts
- API responses use standardized field names and error messages

Debugging Tips

Issue	Likely Cause	Diagnosis	Fix
Inconsistent terminology in logs	Multiple developers using different terms	Grep logs for term variations	Standardize logging messages, use glossary terms
Confusing API responses	Field names don't match documentation	Compare JSON output to glossary	Update response serialization to use standard terms
Documentation conflicts	Glossary out of sync with implementation	Cross-reference docs with code	Update glossary, regenerate documentation
Team communication issues	Different understanding of concepts	Review meeting transcripts for term usage	Conduct glossary review session, align on definitions