RPC Framework: Design Document

Overview

This system implements a simple Remote Procedure Call (RPC) framework that allows clients to invoke functions on remote servers as if they were local method calls. The key architectural challenge is transparently handling network communication, serialization, and error propagation while maintaining the illusion of local function invocation.

> 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):** Foundation for Milestones 1, 2, and 3 - understanding the core RPC challenge

Mental Model: The Postal Service Analogy

Understanding Remote Procedure Calls becomes intuitive when we think about how the postal service operates. Imagine you need to ask your grandmother for her famous cookie recipe, but she lives in another city. You can't simply walk into her kitchen and ask - there's a physical distance barrier that requires a communication protocol.

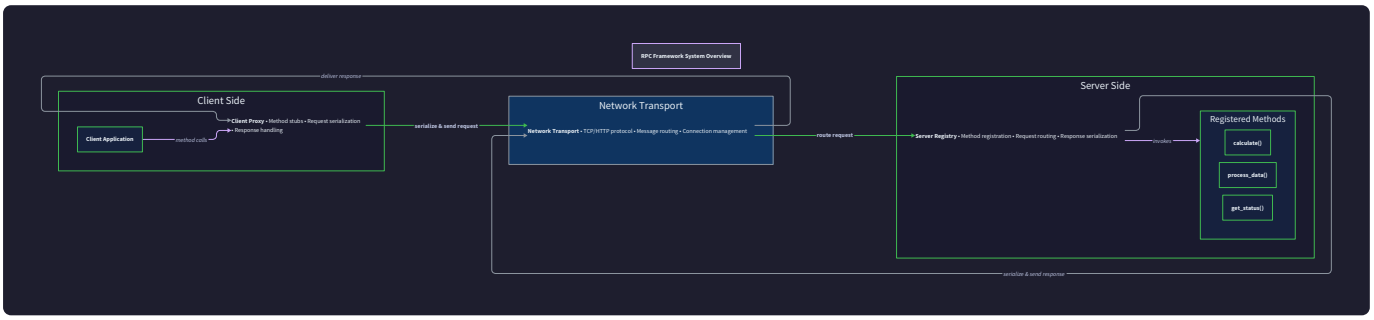
In the traditional postal approach, you write a letter with your request, put it in an envelope with her address, mail it, and wait for her response letter with the recipe. This exchange involves several key elements that directly mirror RPC concepts: you need a **standardized message format** (letter writing conventions), a **delivery mechanism** (postal service), **addressing** (her mailing address), **request identification** (so she knows which letter she's responding to), and **error handling** (what happens if the letter gets lost?).

The **sender** (you) doesn't need to understand the complex logistics of mail sorting, truck routes, or delivery scheduling - you just write the letter and trust the postal service to handle delivery. Similarly, the **recipient** (your grandmother) doesn't need to know how the letter traveled - she just reads your request and writes a response. This **transparency** is the fundamental goal of RPC: making remote function calls feel as natural as local function calls.

However, unlike face-to-face conversation, postal communication introduces several challenges. Letters can get **lost in transit** (network failures), arrive **out of order** (if you send multiple requests), take **unpredictable amounts of time** (latency), or contain **illegible handwriting** (serialization errors). Your grandmother might be **unavailable** (server down), **not understand your request** (method not found), or **unable to fulfill it**

(execution error). The postal service requires both parties to agree on message formatting, addressing schemes, and error handling procedures.

In our RPC framework, the **client** acts like you writing letters with function call requests. The **network transport** serves as the postal service, routing messages between locations. The **server** functions like your grandmother, receiving requests and sending back responses. The **message protocol** defines the standardized format for requests and responses, just like letter-writing conventions ensure clear communication.



This analogy helps us understand why RPC frameworks need several sophisticated components that don't exist in local function calls. We need **message serialization** to convert function parameters into a format suitable for network transmission (like writing legibly). We need **connection management** to establish and maintain communication channels (like ensuring the postal service has correct addresses). We need **error handling** for all the ways remote communication can fail (like dealing with lost or delayed mail). Most importantly, we need **transparency mechanisms** that hide these complexities from the application developer, making remote calls look identical to local calls.

Existing RPC Approaches

The software industry has developed numerous approaches for enabling remote procedure calls, each with different trade-offs between simplicity, performance, and feature richness. Understanding these existing solutions helps us appreciate the design decisions in our educational RPC framework and provides context for why certain patterns have emerged as standards.

REST APIs with HTTP/JSON represent the most ubiquitous approach to remote communication in modern web applications. REST treats remote operations as HTTP requests to resource-oriented URLs, using standard HTTP verbs (GET, POST, PUT, DELETE) to indicate operation types. Parameters are typically passed as URL query parameters, request bodies, or path segments, with JSON serving as the serialization format for complex data structures.

Aspect	REST APIs	Our Assessment
Learning Curve	Shallow - builds on familiar HTTP concepts	Excellent for beginners
Tooling Support	Extensive - every language has HTTP libraries	Universal availability
Performance	Moderate - HTTP overhead, JSON parsing costs	Acceptable for most use cases
Type Safety	Weak - JSON is dynamically typed	Requires careful validation
Transport	HTTP-only - bound to request/response model	Limited flexibility
Discovery	Manual - developers must know endpoints	No built-in service discovery

REST's primary strength lies in its **conceptual simplicity** and **universal tooling support**. Every programming language provides HTTP client libraries, making REST APIs immediately accessible. The **stateless, cacheable** nature of HTTP aligns well with web architecture patterns, and **human-readable** URLs and JSON payloads simplify debugging and testing. However, REST's resource-oriented model can feel **unnatural for procedure-oriented operations**, and the **lack of formal contracts** means API changes can break clients in subtle ways.

gRPC with Protocol Buffers represents Google's high-performance, strongly-typed approach to RPC. gRPC uses HTTP/2 as its transport layer, providing features like **multiplexing, flow control, and header compression**. Protocol Buffers serve as both the **interface definition language** (defining service contracts in `.proto` files) and the **binary serialization format**, offering **strong typing, forward/backward compatibility, and compact wire encoding**.

Aspect	gRPC	Our Assessment
Performance	High - binary encoding, HTTP/2 multiplexing	Excellent for high-throughput systems
Type Safety	Strong - generated client/server code	Prevents many integration bugs
Tooling	Excellent - code generation, reflection, debugging	Rich ecosystem
Learning Curve	Steep - requires Protocol Buffer knowledge	Challenging for beginners
Transport	HTTP/2 required - complex proxy configuration	Infrastructure complexity
Human Readability	Poor - binary format difficult to debug	Harder to troubleshoot

gRPC excels in **microservice architectures** where **performance and type safety** are critical. The **automatic code generation** from `.proto` files ensures client and server implementations stay synchronized. **Streaming support** enables real-time communication patterns like chat systems or live data

feeds. However, gRPC's **binary format complicates debugging**, and **HTTP/2 requirements** can create deployment challenges in environments with legacy proxy servers or firewalls.

JSON-RPC provides a lightweight, specification-based approach that directly mirrors traditional function call semantics. Unlike REST's resource orientation, JSON-RPC explicitly models **method invocation with parameters and return values**. The protocol defines standardized message formats for **requests, responses, and errors**, making implementations more consistent across languages and frameworks.

Aspect	JSON-RPC	Our Assessment
Conceptual Model	Natural - direct function call mapping	Intuitive for developers
Specification	Formal - well-defined message formats	Promotes interoperability
Implementation	Simple - just JSON over HTTP/TCP/WebSocket	Easy to implement correctly
Flexibility	High - transport-agnostic protocol	Adaptable to different needs
Tooling	Limited - fewer frameworks than REST/gRPC	Requires more manual work
Performance	Moderate - JSON overhead, no streaming	Adequate for most applications

JSON-RPC strikes a **balance between simplicity and formality** that makes it an excellent educational choice. The **specification provides clear guidance** for message formats and error handling, while the **transport independence** allows implementations over TCP sockets, HTTP, WebSockets, or message queues. **Batching support** enables multiple method calls in a single request, reducing network round-trips for bulk operations.

Message Queue Systems like RabbitMQ, Apache Kafka, or Redis Pub/Sub take a fundamentally different approach, treating remote communication as **asynchronous message passing** rather than synchronous function calls. Clients publish **request messages** to queues, servers consume and process these messages, then publish **response messages** back to reply queues or topics.

Aspect	Message Queues	Our Assessment
Decoupling	Excellent - clients and servers independent	Great for scalable architectures
Reliability	High - message persistence, delivery guarantees	Handles failures gracefully
Scalability	Excellent - natural load balancing, fan-out	Scales to high message volumes
Complexity	High - requires message broker infrastructure	Significant operational overhead
Latency	Higher - broker adds network hops	Not suitable for low-latency calls
Programming Model	Asynchronous - requires careful state management	More complex application logic

Message queues excel in **distributed systems** where **fault tolerance and scalability** outweigh latency concerns. The **broker-mediated communication** provides **natural load balancing** and **failure isolation** - if

one server instance crashes, others can continue processing queued messages. However, the **asynchronous programming model** requires applications to carefully manage **request correlation and state**, making simple function call patterns more complex to implement.

> **Design Insight**: Our educational RPC framework adopts JSON-RPC's message format because it provides **formal specification guidance** while remaining **simple enough for beginners** to implement without extensive tooling. We'll use **TCP sockets for transport** to focus on core RPC concepts rather than HTTP complexities, and **synchronous request-response patterns** to maintain familiar function call semantics.

The choice of **JSON-RPC over TCP** for our educational framework reflects several pedagogical priorities. **JSON serialization** allows students to **inspect wire format messages** during debugging, unlike binary protocols that require specialized tools. **TCP sockets** expose **fundamental networking concepts** like connection management and byte stream handling, providing deeper systems understanding than HTTP abstraction layers. **Synchronous patterns** maintain the **mental model of function calls** that students already understand, avoiding the complexity of asynchronous programming and callback management.

Architecture Decision Records

> **Decision: JSON-RPC Message Format**
>
> - **Context**: Need a message protocol that balances formal specification with educational clarity
> - **Options Considered**:
>   1. Custom binary protocol (maximum control, high complexity)
>   2. HTTP REST with JSON (familiar, but resource-oriented doesn't match RPC semantics)
>   3. JSON-RPC 2.0 specification (formal, function-call oriented)
> - **Decision**: JSON-RPC 2.0 specification with minor adaptations
> - **Rationale**: Provides formal specification for message formats while maintaining human-readable JSON encoding that aids debugging and learning
> - **Consequences**: Students learn standard protocol patterns and can easily inspect/debug message exchanges, but JSON parsing adds some performance overhead

Option	Pros	Cons
Custom Binary	Maximum performance and control	High complexity, no existing tooling, difficult to debug
HTTP REST	Universal tooling, familiar to web developers	Resource-oriented model doesn't match function call semantics
JSON-RPC	Formal spec, function-call oriented, human-readable	Less tooling than HTTP, moderate performance overhead

Decision: TCP Socket Transport

- **Context**: Need transport layer that exposes fundamental networking concepts for educational value
- **Options Considered**:
  1. HTTP with standard libraries (simple, abstracted)
  2. Raw TCP sockets (educational, full control)
  3. WebSockets (real-time capable, browser-friendly)
- **Decision**: Raw TCP sockets with JSON message framing
- **Rationale**: Forces students to understand connection management, byte streams, and message framing - core networking concepts hidden by HTTP abstractions
- **Consequences**: Students gain deeper networking understanding but must implement connection pooling and error handling manually

Option	Pros	Cons
HTTP	Familiar, extensive tooling, handles framing automatically	Hides networking concepts, adds protocol overhead
TCP Sockets	Educational value, full control, minimal overhead	Requires manual connection and framing management
WebSockets	Real-time capable, familiar to web developers	More complex than needed for basic RPC

Decision: Synchronous Request-Response Pattern

- **Context**: Need to choose between synchronous and asynchronous client API design
- **Options Considered**:
  1. Synchronous blocking calls (simple, familiar function semantics)
  2. Asynchronous with callbacks (scalable, complex error handling)
  3. Asynchronous with futures/promises (modern, requires concurrency understanding)
- **Decision**: Synchronous blocking calls with configurable timeouts
- **Rationale**: Maintains familiar function call semantics while teaching core RPC concepts without concurrent programming complexity
- **Consequences**: Simple to understand and use, but limits scalability for high-concurrency applications

Option	Pros	Cons
Synchronous	Familiar function semantics, simple error handling	Blocks calling thread, limits concurrency
Callbacks	Non-blocking, scalable	Complex error handling, callback hell
Futures/Promises	Modern async patterns, composable	Requires understanding of concurrent programming

These architectural decisions collectively create an RPC framework that **prioritizes learning value over production features**. Students will understand **message serialization, network programming, and distributed error handling** without being overwhelmed by **performance optimizations, concurrent programming, or complex tooling requirements**. The resulting system provides a solid foundation for understanding more sophisticated frameworks like gRPC or enterprise message queuing systems.

Implementation Guidance

Technology Recommendations:

Component	Simple Option	Advanced Option
JSON Handling	`json` module (built-in)	`ujson` or `orjson` for performance
Socket Management	`socket` module (built-in)	`asyncio` for async patterns
Logging	`logging` module (built-in)	`structlog` for structured logs
Testing	`unittest` (built-in)	`pytest` with fixtures

Recommended Project Structure:

```
rpc-framework/
├── rpc/
│   ├── __init__.py
│   ├── protocol.py          ← Message formats and serialization (Milestone 1)
│   ├── server.py            ← RPC server implementation (Milestone 2)
│   ├── client.py            ← RPC client implementation (Milestone 3)
│   └── exceptions.py        ← Custom exception classes
├── examples/
│   ├── calculator_server.py ← Example server with math functions
│   ├── calculator_client.py ← Example client usage
│   └── echo_server.py       ← Simple echo service for testing
├── tests/
│   ├── test_protocol.py     ← Message format tests
│   ├── test_server.py       ← Server functionality tests
│   ├── test_client.py       ← Client functionality tests
│   └── test_integration.py ← End-to-end RPC call tests
└── README.md
```

Core Exception Hierarchy:

PYTHON

```python
# rpc/exceptions.py - Complete exception infrastructure

class RPCError(Exception):
    """Base class for all RPC-related errors."""
    pass

class RPCProtocolError(RPCError):
    """Errors related to message format or protocol violations."""
    pass

class RPCTransportError(RPCError):
    """Errors related to network communication."""
    pass

class RPCTimeoutError(RPCTransportError):
    """Request timeout exceeded."""
    pass

class RPCMethodError(RPCError):
    """Errors related to method execution."""
    def __init__(self, code, message, data=None):
        self.code = code
        self.message = message
        self.data = data
        super().__init__(f"RPC Error {code}: {message}")

# Standard JSON-RPC error codes
class ErrorCode:
    PARSE_ERROR = -32700
    INVALID_REQUEST = -32600
```

```
    METHOD_NOT_FOUND = -32601

    INVALID_PARAMS = -32602

    INTERNAL_ERROR = -32603
```

JSON Message Utilities:

PYTHON

```python
# rpc/protocol.py - Complete message handling infrastructure

import json

import uuid

from typing import Any, Dict, Optional, Union


def generate_request_id() -> str:

    """Generate unique request ID for RPC calls."""

    return str(uuid.uuid4())


def create_request_message(method: str, params: Any = None, request_id: str = None) ->
Dict:

    """Create a JSON-RPC 2.0 request message."""

    # TODO 1: Create message dict with jsonrpc version "2.0"

    # TODO 2: Add method name and request_id (generate if None)

    # TODO 3: Add params only if not None (supports both list and dict params)

    # TODO 4: Return complete request message dict

    pass


def create_response_message(request_id: str, result: Any) -> Dict:

    """Create a JSON-RPC 2.0 success response message."""

    # TODO 1: Create message dict with jsonrpc version "2.0"

    # TODO 2: Add matching request_id and result

    # TODO 3: Return complete response message dict

    pass


def create_error_message(request_id: str, code: int, message: str, data: Any = None) ->
Dict:

    """Create a JSON-RPC 2.0 error response message."""

    # TODO 1: Create message dict with jsonrpc version "2.0"

    # TODO 2: Add matching request_id
```

```
    # TODO 3: Create error object with code and message
    # TODO 4: Add data to error object if provided
    # TODO 5: Return complete error response message dict
    pass

def serialize_message(message: Dict) -> bytes:
    """Serialize message to JSON bytes with length prefix."""
    # TODO 1: Convert message dict to JSON string
    # TODO 2: Encode JSON string to UTF-8 bytes
    # TODO 3: Create length prefix (4-byte big-endian integer)
    # TODO 4: Return length prefix + message bytes
    # Hint: Use struct.pack('>I', length) for big-endian 32-bit int
    pass

def deserialize_message(data: bytes) -> Dict:
    """Deserialize JSON bytes to message dict."""
    # TODO 1: Decode UTF-8 bytes to JSON string
    # TODO 2: Parse JSON string to dict
    # TODO 3: Validate required fields (jsonrpc, id)
    # TODO 4: Return message dict or raise RPCProtocolError
    pass
```

Socket Helper Utilities:

PYTHON

```python
# rpc/transport.py - Complete network transport utilities

import socket

import struct

from typing import Optional


class SocketHelper:

    """Utility class for socket operations with proper error handling."""



    @staticmethod

    def send_all(sock: socket.socket, data: bytes, timeout: float = 30.0) -> None:

        """Send all bytes, handling partial sends."""

        # TODO 1: Set socket timeout

        # TODO 2: Track bytes sent with offset

        # TODO 3: Loop until all bytes sent, handle EAGAIN/EWOULDBLOCK

        # TODO 4: Raise RPCTransportError on socket errors

        pass



    @staticmethod

    def recv_all(sock: socket.socket, size: int, timeout: float = 30.0) -> bytes:

        """Receive exact number of bytes."""

        # TODO 1: Set socket timeout

        # TODO 2: Track bytes received with buffer

        # TODO 3: Loop until exact size received

        # TODO 4: Handle connection closed (recv returns empty bytes)

        # TODO 5: Raise RPCTransportError on socket errors

        pass
```

```
    @staticmethod
    def recv_message(sock: socket.socket, timeout: float = 30.0) -> bytes:
        """Receive length-prefixed message."""
        # TODO 1: Receive 4-byte length prefix
        # TODO 2: Unpack big-endian integer from prefix
        # TODO 3: Receive message bytes of specified length
        # TODO 4: Return message bytes
        # Hint: Use struct.unpack('>I', prefix)[0] for big-endian 32-bit int
        pass
```

Language-Specific Implementation Hints:

- **JSON Handling**: Use `json.loads()` and `json.dumps()` for message serialization. Handle `json.JSONDecodeError` for malformed messages.
- **Socket Programming**: Use `socket.socket(socket.AF_INET, socket.SOCK_STREAM)` for TCP sockets. Always call `close()` in finally blocks.
- **Error Propagation**: Wrap socket exceptions in custom `RPCTransportError` to provide consistent error handling across the framework.
- **Thread Safety**: Use `threading.Lock()` if implementing concurrent request handling in the server.
- **Timeout Handling**: Use `socket.settimeout()` for network operations and catch `socket.timeout` exceptions.

Development Workflow:

1. **Start with Protocol**: Implement message creation and serialization functions first - these are the foundation for client and server.
2. **Build Server Next**: Create the method registry and request dispatcher - easier to test than client since you control both ends.
3. **Add Client Last**: Implement the proxy object and connection management - can test against your working server.
4. **Test Integration**: Write end-to-end tests that exercise client → server → client round trips with various scenarios.

Common Setup Mistakes to Avoid:

⚠️ **Pitfall: Forgetting Message Framing** TCP is a byte stream, not a message stream. Without length prefixes, you can't tell where one JSON message ends and the next begins. Always use length-prefixed framing.

⚠️ **Pitfall: Not Handling Partial Socket Operations**
`socket.send()` and `socket.recv()` may not send/receive all requested bytes in a single call. Always loop until complete.

⚠️ **Pitfall: Mixing Bytes and Strings** JSON serialization produces strings, but sockets require bytes. Always encode/decode at the socket boundary using UTF-8.

Goals and Non-Goals

> **Milestone(s):** Foundation for all milestones - defines the scope and boundaries of our RPC framework implementation

Mental Model: The Swiss Army Knife vs. The Hammer

Think of our RPC framework like choosing between a Swiss Army knife and a hammer. A Swiss Army knife has dozens of tools - screwdrivers, scissors, can opener, magnifying glass - and can handle almost any situation you encounter. However, each tool is small and somewhat awkward to use. In contrast, a hammer does exactly one thing: drive nails. It's simple, reliable, and excellent at its specific job, but useless for cutting wire or opening bottles.

Production RPC frameworks like gRPC are Swiss Army knives. They handle authentication, load balancing, circuit breakers, multiple serialization formats, streaming, compression, service discovery, and dozens of other concerns. They're incredibly powerful but also complex to understand and implement.

Our educational RPC framework is the hammer. It does one thing well: allow a client to call a function on a remote server as if it were local. By focusing solely on this core mechanism, we can understand the fundamental principles without getting lost in production concerns. Once you master the hammer, you'll appreciate why the Swiss Army knife needs all those extra tools.

Functional Goals

Our RPC framework must demonstrate the core concepts that make remote procedure calls possible. These goals define the minimum viable system that teaches the essential patterns and challenges of distributed function invocation.

Core RPC Semantics

The framework must provide the illusion of local function calls despite network boundaries. When a client calls `calculator.add(5, 3)`, it should feel identical to calling a local method, even though the actual computation happens on a remote server. This transparency is the defining characteristic of RPC systems.

The system must support **method registration** on the server side, allowing developers to expose specific functions for remote invocation. A server should be able to register functions like `add`, `subtract`,

`get_user`, or `process_payment` and make them available to remote clients through a simple registry mechanism.

JSON-RPC Protocol Implementation

We will implement a subset of the JSON-RPC 2.0 specification, which provides a lightweight, human-readable protocol for remote procedure calls. Our implementation must handle the three core message types defined by this specification.

Message Type	Required Fields	Purpose
Request	`method`, `params`, `id`	Invoke remote function with parameters
Response	`result`, `id`	Return successful function result
Error	`error`, `id`	Report function execution or protocol errors

The protocol must support **request-response correlation** using unique request IDs. When a client sends a request with ID "abc123", it must receive a response with the matching ID "abc123". This correlation allows clients to handle multiple concurrent requests and match responses to their originating calls.

Network Transport and Serialization

The framework must implement **TCP socket communication** for reliable message delivery between clients and servers. TCP provides the ordered, reliable byte stream we need for RPC communication, handling packet loss, reordering, and corruption at the network level.

All messages must use **JSON serialization** for cross-language compatibility and human readability during development. The system must handle serialization of common Python types including strings, numbers, lists, dictionaries, and `None` values.

The implementation must solve the **message framing problem** - determining where one message ends and the next begins in a continuous byte stream. We'll use a simple length-prefix framing protocol where each message is preceded by a 4-byte header indicating the message size.

Error Handling and Propagation

The framework must provide comprehensive error handling that distinguishes between different failure modes and propagates appropriate error information back to clients.

Error Category	Example Causes	Client Experience
Protocol Errors	Invalid JSON, missing fields	`RPCProtocolError` exception
Transport Errors	Connection dropped, network unreachable	`RPCTransportError` exception
Method Errors	Function not found, execution failed	`RPCMethodError` exception with details
Timeout Errors	Server too slow, network congestion	`RPCTimeoutError` exception

Each error type must include sufficient information for debugging while maintaining security boundaries. Method execution errors should include the error message and type information, but not sensitive server-side details like file paths or stack traces.

Client Proxy Interface

The client must provide a **proxy object** that converts method calls into RPC requests transparently. Instead of manually constructing JSON messages, developers should write natural Python code:

```python
# This natural syntax...

result = client.calculator.add(5, 3)


# ...should automatically generate and send this RPC request:

# {"method": "add", "params": [5, 3], "id": "req_001"}
```

The proxy must handle **timeout management**, raising `RPCTimeoutError` when remote calls exceed the configured deadline. This prevents clients from waiting indefinitely for servers that have crashed or become unresponsive.

Server Method Registry and Dispatch

The server must implement a **method registry** that maps string method names to callable Python functions. The registry should support simple registration syntax:

```python
server.register_method("add", lambda a, b: a + b)

server.register_method("get_user", user_service.get_user_by_id)
```

The server must implement **request dispatch logic** that parses incoming messages, looks up the requested method in the registry, invokes the method with the provided parameters, and serializes the result back to the client. This dispatch process must handle parameter validation, method execution, and error response generation.

Non-Goals

Understanding what we will NOT implement is equally important for maintaining focus and managing scope. These exclusions allow us to concentrate on core RPC concepts without getting distracted by production concerns.

Production Reliability Features

We will not implement **authentication or authorization** mechanisms. Our framework assumes a trusted network environment where all clients are permitted to invoke all registered methods. Production systems

require sophisticated access control, API keys, OAuth tokens, or certificate-based authentication, but these concerns are orthogonal to understanding RPC fundamentals.

The framework will not include **encryption or transport security**. All communication happens in plaintext over TCP sockets. Production RPC systems use TLS encryption to protect sensitive data in transit, but implementing cryptography would obscure the core message exchange patterns we're trying to learn.

We will not implement **connection pooling or persistent connections**. Each RPC call will establish a new TCP connection, send the request, receive the response, and close the connection. This approach is inefficient but simpler to understand and debug. Production systems maintain connection pools to amortize connection establishment costs across multiple calls.

Performance and Scalability Features

The system will not support **asynchronous or streaming calls**. Every RPC call follows a simple synchronous request-response pattern where the client blocks until the server returns a result. Asynchronous RPC systems use callbacks, futures, or async/await patterns to handle multiple concurrent calls without blocking threads.

We will not implement **load balancing or service discovery**. Clients must know the exact IP address and port of the server they want to contact. Production systems use service registries, DNS-based discovery, or load balancers to distribute requests across multiple server instances.

The framework will not include **circuit breakers or retry mechanisms**. If a server is unreachable or returns an error, the client will immediately report the failure to the application. Production systems implement sophisticated retry policies with exponential backoff, circuit breakers that stop calling failing services, and graceful degradation strategies.

Advanced Protocol Features

We will not support **binary serialization formats** like Protocol Buffers, MessagePack, or Apache Avro. JSON serialization is less efficient but much easier to debug and understand. Binary formats require schema definitions, code generation, and specialized tooling that would complicate our learning objectives.

The system will not implement **batch requests or pipelining**. Each RPC call requires a separate network round-trip. Some RPC protocols allow clients to send multiple requests in a single message or send requests without waiting for responses, but these optimizations add protocol complexity.

We will not support **bidirectional communication or server push**. Communication flows strictly from client to server and back. Modern RPC frameworks support streaming responses, server-initiated callbacks, and full-duplex communication channels, but these patterns require more sophisticated connection management.

Error Recovery and Reliability

The framework will not implement **automatic retry logic**. If a network error occurs or a server becomes unavailable, the client will immediately raise an exception. Production systems implement retry policies that distinguish between transient errors (network congestion, temporary server overload) and permanent errors (method not found, invalid parameters).

We will not include **distributed tracing or monitoring** capabilities. Each RPC call happens in isolation without correlation IDs, performance metrics, or observability hooks. Production systems instrument every remote call with tracing data to help debug performance issues and understand system behavior.

The system will not support **graceful shutdown or connection draining**. Servers will terminate immediately when stopped, potentially interrupting in-flight requests. Production servers implement graceful shutdown sequences that stop accepting new requests while allowing existing requests to complete.

Multi-Language and Compatibility Features

While our protocol uses JSON and could theoretically support multiple languages, we will not provide **client libraries for other languages**. The implementation will be Python-only, with Python-specific error types, proxy mechanisms, and API patterns.

We will not implement **schema validation or interface definition languages**. Methods can be called with any parameters, and type checking happens at runtime during method execution. Production RPC systems often use schema languages like Protocol Buffer definitions or OpenAPI specifications to define and validate interfaces.

> **Design Principle: Educational Focus Over Production Readiness**
>
> Every non-goal represents a deliberate choice to prioritize learning over completeness. Production RPC frameworks are complex because distributed systems are complex. By temporarily ignoring these concerns, we can focus on the core insight: how to make a function call traverse a network boundary while maintaining the illusion of local execution.

Implementation Guidance

This framework serves as a foundation for understanding distributed systems concepts. The implementation should be straightforward enough for junior developers to complete in a few days while demonstrating the essential patterns used in production RPC systems.

A. Technology Recommendations

Component	Simple Option	Advanced Option	Chosen
Serialization	JSON with `json` module	MessagePack or Protocol Buffers	JSON ✓
Transport	Raw TCP sockets with `socket` module	HTTP with `requests` library	TCP ✓
Threading	Single-threaded blocking server	`threading` or `asyncio` for concurrency	Single-threaded ✓
Error Handling	Exception hierarchy with custom types	Result types or error codes	Exceptions ✓
Message Framing	Length-prefix (4 bytes + payload)	Delimiter-based or HTTP chunking	Length-prefix ✓

B. Recommended File Structure

The codebase should be organized into focused modules that separate concerns and make testing easier:

```
rpc_framework/
├── __init__.py                # Main package exports
├── protocol.py                # Message formats and serialization
├── errors.py                  # Exception hierarchy
├── server.py                  # RPC server implementation
├── client.py                  # RPC client and proxy
├── utils.py                   # Socket helpers and utilities
└── examples/
    ├── calculator_server.py   # Example server with math functions
    ├── calculator_client.py   # Example client usage
    └── test_integration.py    # End-to-end integration tests
```

This structure separates the protocol definition from the client and server implementations, making it easier to understand each component independently.

C. Infrastructure Starter Code

Socket Helper Utilities (`utils.py`)

PYTHON

```python
import socket

import struct

import time

from typing import Optional


class SocketHelper:

    """Utility functions for reliable socket communication with timeouts."""



    @staticmethod

    def send_all(sock: socket.socket, data: bytes, timeout: Optional[float] = None) ->
None:

        """Send all bytes, handling partial sends and timeouts.



        Args:

            sock: Connected socket

            data: Bytes to send

            timeout: Maximum time to wait (None for blocking)



        Raises:

            RPCTransportError: If send fails or times out

        """

        if timeout is not None:

            sock.settimeout(timeout)



        total_sent = 0

        while total_sent < len(data):

            try:

                sent = sock.send(data[total_sent:])
```

```
                if sent == 0:
                    raise RPCTransportError("Socket connection broken")
                total_sent += sent
            except socket.timeout:
                raise RPCTimeoutError(f"Send timeout after {timeout}s")
            except socket.error as e:
                raise RPCTransportError(f"Send failed: {e}")

    @staticmethod
    def recv_all(sock: socket.socket, size: int, timeout: Optional[float] = None) -> bytes:
        """Receive exactly size bytes, handling partial receives.

        Args:
            sock: Connected socket
            size: Number of bytes to receive
            timeout: Maximum time to wait

        Returns:
            Exact number of bytes requested

        Raises:
            RPCTransportError: If receive fails or connection closes
            RPCTimeoutError: If receive times out
        """
        if timeout is not None:
            sock.settimeout(timeout)
```

```
        chunks = []
        bytes_received = 0

        while bytes_received < size:
            try:
                chunk = sock.recv(size - bytes_received)
                if not chunk:
                    raise RPCTransportError("Connection closed by peer")
                chunks.append(chunk)
                bytes_received += len(chunk)
            except socket.timeout:
                raise RPCTimeoutError(f"Receive timeout after {timeout}s")
            except socket.error as e:
                raise RPCTransportError(f"Receive failed: {e}")

        return b''.join(chunks)

    @staticmethod
    def recv_message(sock: socket.socket, timeout: Optional[float] = None) -> bytes:
        """Receive a length-prefixed message.

        Protocol: 4-byte length (network byte order) + message payload

        Args:
            sock: Connected socket
            timeout: Maximum time to wait
```

```
        Returns:
            Message payload bytes
        """
        # First receive the 4-byte length header
        length_data = SocketHelper.recv_all(sock, 4, timeout)
        message_length = struct.unpack('!I', length_data)[0]

        # Validate message length to prevent memory attacks
        if message_length > 1024 * 1024:  # 1MB limit
            raise RPCProtocolError(f"Message too large: {message_length} bytes")

        # Then receive the actual message
        return SocketHelper.recv_all(sock, message_length, timeout)
```

Error Hierarchy (`errors.py`)

PYTHON

```python
class RPCError(Exception):
    """Base class for all RPC-related errors."""
    pass

class RPCProtocolError(RPCError):
    """Errors in message format or protocol violations."""
    pass

class RPCTransportError(RPCError):
    """Network communication errors."""
    pass

class RPCTimeoutError(RPCError):
    """Request timeout errors."""
    pass

class RPCMethodError(RPCError):
    """Method execution errors with JSON-RPC error details."""


    def __init__(self, code: int, message: str, data=None):
        super().__init__(f"RPC Method Error [{code}]: {message}")
        self.code = code
        self.message = message
        self.data = data

class ErrorCode:
    """Standard JSON-RPC error codes."""
    PARSE_ERROR = -32700
    INVALID_REQUEST = -32600
```

```
    METHOD_NOT_FOUND = -32601

    INVALID_PARAMS = -32602

    INTERNAL_ERROR = -32603
```

D. Core Logic Skeleton Code

Protocol Message Handling (`protocol.py`)

```python
import json

import struct

import uuid

from typing import Dict, Any, Optional


def generate_request_id() -> str:

    """Generate a unique request ID for correlating requests and responses.



    Returns:

        Unique string identifier

    """

    # TODO: Generate a unique ID using uuid4() and return as string

    # Hint: Use uuid.uuid4().hex for a simple string representation


def create_request_message(method: str, params: Any, request_id: str) -> Dict:

    """Create a JSON-RPC request message.



    Args:

        method: Remote method name to invoke

        params: Parameters to pass to method (list or dict)

        request_id: Unique identifier for this request



    Returns:

        Dictionary representing JSON-RPC request

    """

    # TODO: Create dictionary with required JSON-RPC fields:

    # - "jsonrpc": "2.0" (protocol version)

    # - "method": method name
```

```
    # - "params": parameters
    # - "id": request_id

def create_response_message(request_id: str, result: Any) -> Dict:
    """Create a JSON-RPC response message for successful calls.

    Args:
        request_id: ID from the original request
        result: Return value from method execution

    Returns:
        Dictionary representing JSON-RPC response
    """
    # TODO: Create dictionary with required JSON-RPC response fields:
    # - "jsonrpc": "2.0"
    # - "result": the return value
    # - "id": request_id

def create_error_message(request_id: str, code: int, message: str, data: Any = None) ->
Dict:
    """Create a JSON-RPC error message.

    Args:
        request_id: ID from the original request
        code: Standard error code (see ErrorCode class)
        message: Human-readable error description
        data: Optional additional error information
```

```
    Returns:
        Dictionary representing JSON-RPC error
    """
    # TODO: Create dictionary with JSON-RPC error structure:
    # - "jsonrpc": "2.0"
    # - "error": {"code": code, "message": message, "data": data}
    # - "id": request_id
    # Note: Only include "data" field if data is not None

def serialize_message(message: Dict) -> bytes:
    """Convert message dictionary to length-prefixed bytes for network transmission.

    Protocol: 4-byte length header (network byte order) + JSON payload

    Args:
        message: Dictionary to serialize

    Returns:
        Length-prefixed message bytes

    Raises:
        RPCProtocolError: If JSON serialization fails
    """
    # TODO 1: Convert message dict to JSON string using json.dumps()
    # TODO 2: Encode JSON string to UTF-8 bytes
    # TODO 3: Calculate message length
    # TODO 4: Pack length as 4-byte network byte order integer using struct.pack('!I', length)
```

```
    # TODO 5: Concatenate length header + message bytes and return


def deserialize_message(data: bytes) -> Dict:
    """Parse bytes to message dictionary.

    Args:
        data: Raw message bytes (no length prefix)

    Returns:
        Parsed message dictionary

    Raises:
        RPCProtocolError: If JSON parsing fails
    """
    # TODO 1: Decode bytes to UTF-8 string
    # TODO 2: Parse JSON string to dictionary using json.loads()
    # TODO 3: Validate that result is a dictionary
    # TODO 4: Return the parsed dictionary
    # Hint: Wrap json.loads() in try/except and raise RPCProtocolError for invalid JSON
```

E. Language-Specific Hints

Socket Programming in Python:

- Use `socket.socket(socket.AF_INET, socket.SOCK_STREAM)` for TCP sockets
- Call `sock.bind(('localhost', port))` and `sock.listen(1)` to create a server
- Use `sock.accept()` to wait for client connections (returns new socket + address)
- Call `sock.connect((host, port))` from client to establish connection
- Always call `sock.close()` in a `finally` block or use `with` statement

JSON Serialization:

- `json.dumps(obj)` converts Python object to JSON string

- `json.loads(string)` parses JSON string to Python object
- JSON supports: strings, numbers, booleans, None (becomes null), lists, dictionaries
- JSON cannot serialize: functions, classes, bytes, datetime objects

Struct for Binary Data:

- `struct.pack('!I', value)` packs integer as 4-byte network byte order
- `struct.unpack('!I', bytes)[0]` unpacks 4-byte network byte order to integer
- The `!` means network byte order (big-endian)
- The `I` means unsigned 32-bit integer

F. Milestone Checkpoints

After Milestone 1 (Message Protocol):

- Run: `python -c "from protocol import *; print(create_request_message('add', [1,2], 'test'))"`
- Expected: `{'jsonrpc': '2.0', 'method': 'add', 'params': [1, 2], 'id': 'test'}`
- Test serialization round-trip: message → serialize → deserialize → should equal original
- Verify length-prefix format: serialized message should start with 4-byte length

After Milestone 2 (Server Implementation):

- Run: `python calculator_server.py` (should start server on localhost:8000)
- Test with telnet: `telnet localhost 8000`
- Send raw JSON-RPC request and verify you get a JSON-RPC response
- Server should handle multiple sequential connections (connect, call, disconnect, repeat)

After Milestone 3 (Client Implementation):

- Run server in one terminal: `python calculator_server.py`
- Run client in another: `python calculator_client.py`
- Expected: Client should print results like "5 + 3 = 8"
- Test timeout: Stop server, client should raise `RPCTimeoutError`
- Test invalid method: Call non-existent method, should raise `RPCMethodError`

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
"Connection refused"	Server not running	Check if server process is active	Start server first
"JSON decode error"	Malformed message	Print raw bytes before deserializing	Fix message construction
Client hangs forever	Wrong message length	Verify 4-byte length header matches payload	Use `struct.pack('!I', len(data))`
"Method not found"	Method not registered	Check server method registry	Call `server.register_method()`
Partial message received	Not using `recv_all`	Use provided `SocketHelper.recv_all()`	Always receive exact byte count
"Broken pipe" error	Client disconnected early	Check client timeout settings	Increase timeout or fix server speed

> **Key Learning Checkpoint**: By the end of this implementation, you should understand how network boundaries affect function calls, why serialization is necessary, how to correlate requests with responses, and what kinds of errors can occur in distributed systems. These concepts form the foundation for understanding more sophisticated distributed systems patterns.

High-Level Architecture

> **Milestone(s):** Foundation for Milestones 1, 2, and 3 - establishes the overall system structure and component responsibilities

The RPC Framework consists of three primary components that work together to enable transparent remote method invocation. Understanding how these components interact and their individual responsibilities is crucial for successful implementation. This architecture follows a clear separation of concerns, where each component has a well-defined role in the overall RPC communication process.

Component Overview

Think of our RPC framework as a **telephone system connecting different offices**. The message protocol acts like the standardized language and format everyone uses when making calls. The server is like a corporate switchboard operator who receives calls, understands what department the caller needs, and routes

them to the right person. The client is like an executive assistant who knows how to dial the switchboard, speak the standard language, and handle the responses on behalf of their boss.

This mental model helps illustrate the key principle: **each component has a single, focused responsibility** that contributes to the illusion of local method calls across a network boundary.

Message Protocol Component

The **message protocol component** serves as the foundation layer that defines how RPC communication is structured and serialized. This component establishes the wire format and message structure that both client and server must understand. It acts as the common language that enables interoperability between different RPC endpoints.

The protocol component's primary responsibilities include defining the JSON-RPC message formats, handling message serialization and deserialization, implementing message framing for TCP streams, and providing error code definitions. This component ensures that all communication follows a consistent, predictable structure that can be reliably parsed and understood by both ends of the connection.

Responsibility	Description	Key Functions
Message Structure	Defines request, response, and error message formats	`create_request_message`, `create_response_message`, `create_error_message`
Serialization	Converts messages to/from wire format	`serialize_message`, `deserialize_message`
Message Framing	Handles length-prefixed message boundaries	Built into serialize/deserialize functions
Request ID Management	Generates unique identifiers for request correlation	`generate_request_id`
Error Classification	Defines standard error codes and formats	`ErrorCode` constants

The protocol component must handle several critical aspects of RPC communication. **Message framing** solves the fundamental problem of determining where one message ends and another begins in a TCP byte stream. Since TCP provides a stream of bytes without message boundaries, we use a length prefix approach where each message is preceded by a 4-byte header indicating the message size.

Request ID correlation ensures that responses can be matched to their corresponding requests, especially important when multiple requests are in flight simultaneously. Each request receives a unique identifier that must be echoed back in the response, allowing the client to properly route responses to waiting callers.

Server Component

The **server component** acts as the RPC service provider, accepting incoming connections and executing remote method calls. Think of it as a **receptionist and call routing system** - it listens for incoming requests, understands what the client wants to do, finds the right function to handle that request, and sends back the appropriate response.

The server's architecture centers around the concept of a **method registry** - a mapping from string names to callable Python functions. When a client requests a method call, the server looks up the method name in its registry, validates the parameters, executes the function, and returns the result or error.

Responsibility	Description	Implementation Strategy
Connection Management	Accept and handle TCP connections	Single-threaded with connection reuse
Method Registry	Map method names to callable functions	Dictionary-based lookup
Request Processing	Parse messages and dispatch to handlers	Sequential message processing
Parameter Validation	Ensure method calls have correct parameters	Runtime type checking
Response Generation	Create and send response messages	Automatic serialization and framing
Error Handling	Convert exceptions to RPC error responses	Structured error code mapping

The server follows a **synchronous, single-threaded model** for simplicity. Each incoming connection is handled sequentially, processing one request at a time before moving to the next. This design choice eliminates concurrency complexity while still providing a functional RPC service suitable for learning purposes.

Method registration happens at server startup, where application code registers functions by providing a string name and a callable object. The server stores these mappings and uses them to dispatch incoming requests to the appropriate handlers.

> **Design Insight**: The single-threaded server design prioritizes simplicity over performance. Real-world RPC servers would use threading or async I/O, but for educational purposes, the sequential model makes debugging easier and eliminates race conditions.

Client Component

The **client component** provides the caller-side interface for making RPC calls. Its primary goal is to make remote method invocation feel as natural as calling a local function. Think of the client as a **personal secretary** who handles all the complex details of making phone calls - they know the right numbers to dial, speak the proper protocol language, wait for responses, and handle any problems that arise.

The client's most important feature is **method proxying** - the ability to intercept method calls on a proxy object and convert them into RPC requests. When application code calls `proxy.calculate(5, 3)`, the client automatically creates an RPC request message, sends it to the server, waits for the response, and returns the result as if the method had been executed locally.

Responsibility	Description	Key Mechanisms
Connection Management	Establish and maintain TCP connections	Socket lifecycle management
Method Proxying	Convert method calls to RPC requests	Python `__getattr__` magic method
Request Correlation	Track pending requests by ID	Dictionary mapping request IDs to futures
Timeout Handling	Abort requests that take too long	Socket timeout configuration
Response Processing	Convert RPC responses back to return values	Automatic deserialization
Error Translation	Convert RPC errors to Python exceptions	Custom exception hierarchy

Connection management in the client involves establishing TCP connections to the server and handling connection lifecycle. For simplicity, our client creates a new connection for each RPC call, though production systems would typically use connection pooling for better performance.

Timeout handling is critical for preventing client applications from hanging indefinitely when servers become unresponsive. The client sets socket timeouts and raises `RPCTimeoutError` exceptions when requests exceed their deadline.

> **Architecture Decision: Synchronous vs Asynchronous Client API**
>
> - **Context**: Clients can provide either blocking method calls or async/callback-based APIs
> - **Options Considered**:
>   1. Synchronous blocking calls that wait for responses
>   2. Asynchronous calls with callbacks or futures
>   3. Both options with a configuration flag
> - **Decision**: Synchronous blocking calls only
> - **Rationale**: Blocking calls are much simpler to implement and use, matching the mental model of "remote calls that work like local calls." Async support adds significant complexity in error handling, timeout management, and API design.
> - **Consequences**: Enables simple, intuitive client code but limits performance in scenarios requiring many concurrent calls

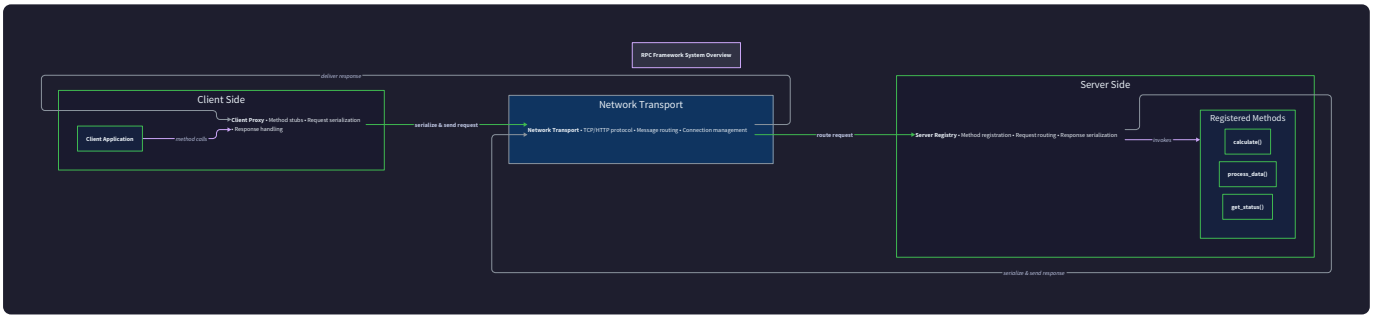
Component Interaction Patterns

The three components interact through well-defined interfaces and protocols. Understanding these interaction patterns is crucial for implementing each component correctly and debugging issues that span component boundaries.

Request Flow follows a predictable sequence: the client proxy converts method calls to protocol messages, sends them over the network to the server, which deserializes the messages, executes the requested methods, and sends back protocol-formatted responses. Each step in this flow has specific responsibilities and error handling requirements.

Error Propagation must work correctly across component boundaries. Server-side exceptions become RPC error messages, which the client converts back to Python exceptions. This requires consistent error classification and proper serialization of error details.

State Management is distributed across components. The protocol component is stateless, handling individual messages independently. The server maintains method registry state and connection state. The client maintains connection state and tracks pending requests by ID.



Recommended File Structure

Organizing the RPC framework code into logical modules makes the implementation more maintainable and helps separate concerns clearly. The recommended structure follows Python packaging conventions while grouping related functionality together.

```
rpc_framework/
├── __init__.py              # Package initialization and public API
├── protocol.py              # Message protocol implementation
├── server.py                # RPC server implementation
├── client.py                # RPC client implementation
├── exceptions.py            # Exception hierarchy and error codes
├── utils.py                 # Socket utilities and helpers
└── examples/
    ├── __init__.py
    ├── calculator_server.py    # Example server with math functions
    ├── calculator_client.py    # Example client usage
    └── test_integration.py     # End-to-end integration tests
```

This file organization supports the natural development progression through the three milestones. Each major component lives in its own module, making it easy to focus on one piece at a time while maintaining clear interfaces between components.

Core Module Responsibilities

The `protocol.py` module (Milestone 1) contains all message format definitions, serialization functions, and protocol constants. This module should be completely independent of networking code, focusing purely on message structure and wire format encoding. It exports the message creation functions, serialization utilities, and error code constants that other modules depend on.

The `server.py` module (Milestone 2) implements the RPC server class, method registry, and request dispatch logic. This module imports the protocol module for message handling but is independent of client-side code. It exports the main server class and any server-specific configuration options.

The `client.py` module (Milestone 3) implements the RPC client class, proxy object, and connection management. This module imports the protocol module for message handling but is independent of server-side code. It exports the main client class and proxy object that application code interacts with.

The `exceptions.py` module defines the complete exception hierarchy used throughout the framework. This includes both protocol-level errors (like `RPCProtocolError`) and application-level errors (like `RPCMethodError`). Having all exceptions in one module makes error handling consistent across components.

The `utils.py` module contains networking utilities and helper functions shared by both client and server. This includes the `SocketHelper` class with functions like `send_all`, `recv_all`, and `recv_message` that handle the low-level details of TCP communication with proper error handling.

Module	Primary Classes	Key Functions	Dependencies
`protocol.py`	`ErrorCode`	`create_request_message`, `serialize_message`, `generate_request_id`	None (pure protocol)
`server.py`	`RPCServer`	`register_method`, `start_server`, `handle_request`	protocol, utils, exceptions
`client.py`	`RPCClient`, `RPCProxy`	`call_method`, `connect`, `__getattr__`	protocol, utils, exceptions
`exceptions.py`	`RPCError`, `RPCTimeoutError`, etc.	Exception constructors	None
`utils.py`	`SocketHelper`	`send_all`, `recv_all`, `recv_message`	socket, struct

Development Workflow

The recommended file structure supports a natural development progression where each milestone builds on the previous one. **Milestone 1** focuses entirely on `protocol.py` and `exceptions.py`, allowing you to perfect message handling before dealing with networking complexity. **Milestone 2** adds `server.py` and the networking parts of `utils.py`, building the service side of the RPC system. **Milestone 3** completes the framework by implementing `client.py` and the proxy mechanism.

This structure also supports good testing practices. Each module can be unit tested independently, and the `examples/` directory provides integration tests that exercise the complete system. The examples serve both as test cases and as documentation showing how to use the framework.

> **Implementation Insight**: Start each milestone by defining the public interface in `__init__.py`. This forces you to think about what other modules need from each component and helps maintain clean separation of concerns.

Implementation Guidance

The high-level architecture provides the foundation for implementing a working RPC framework. This guidance covers technology choices, starter code structure, and practical implementation approaches for each component.

Technology Recommendations

Component	Simple Approach	Advanced Alternative
Message Protocol	JSON with length-prefix framing	MessagePack or Protocol Buffers
Server Transport	Raw TCP sockets with blocking I/O	Async I/O with asyncio
Client Transport	Raw TCP sockets with timeouts	HTTP/2 or connection pooling
Serialization	Python json module	Custom binary protocol
Error Handling	Exception-based with structured errors	Result types or error codes
Method Registry	Simple dict mapping names to functions	Decorator-based registration

For this educational implementation, we recommend the simple approaches. JSON is human-readable and easy to debug, raw TCP sockets expose the fundamental networking concepts, and exception-based error handling matches Python idioms.

Project Structure Setup

Create the initial project structure with proper Python packaging:

PYTHON

```python
# rpc_framework/__init__.py

"""Simple RPC Framework for educational purposes."""


__version__ = "1.0.0"


# Public API - expose main classes

from .server import RPCServer

from .client import RPCClient

from .exceptions import (

    RPCError,

    RPCProtocolError,

    RPCTransportError,

    RPCTimeoutError,

    RPCMethodError

)


__all__ = [

    'RPCServer',

    'RPCClient',

    'RPCError',

    'RPCProtocolError',

    'RPCTransportError',

    'RPCTimeoutError',

    'RPCMethodError'

]
```

PYTHON

```python
# rpc_framework/exceptions.py

"""Exception hierarchy for RPC framework."""


class RPCError(Exception):

    """Base class for all RPC-related errors."""

    pass


class RPCProtocolError(RPCError):

    """Raised when message format is invalid or cannot be parsed."""

    pass


class RPCTransportError(RPCError):

    """Raised when network communication fails."""

    pass


class RPCTimeoutError(RPCError):

    """Raised when request times out waiting for response."""

    pass


class RPCMethodError(RPCError):

    """Raised when remote method execution fails."""



    def __init__(self, code, message, data=None):

        self.code = code

        self.message = message

        self.data = data

        super().__init__(f"RPC Method Error {code}: {message}")


class ErrorCode:

    """Standard JSON-RPC error codes."""
```

```
    PARSE_ERROR = -32700

    INVALID_REQUEST = -32600

    METHOD_NOT_FOUND = -32601

    INVALID_PARAMS = -32602

    INTERNAL_ERROR = -32603
```

Socket Utilities Infrastructure

Provide complete socket helper functions that both client and server can use:

PYTHON

```python
# rpc_framework/utils.py

"""Networking utilities for RPC framework."""


import socket

import struct

import time

from typing import Optional

from .exceptions import RPCTransportError, RPCTimeoutError


class SocketHelper:

    """Utility class for reliable socket operations."""



    @staticmethod

    def send_all(sock: socket.socket, data: bytes, timeout: Optional[float] = None) ->
None:

        """Send all bytes, handling partial sends and timeouts."""

        if timeout:

            sock.settimeout(timeout)



        total_sent = 0

        while total_sent < len(data):

            try:

                sent = sock.send(data[total_sent:])

                if sent == 0:

                    raise RPCTransportError("Socket connection broken during send")

                total_sent += sent

            except socket.timeout:

                raise RPCTimeoutError(f"Send timeout after {timeout} seconds")
```

```
            except socket.error as e:
                raise RPCTransportError(f"Socket send error: {e}")

    @staticmethod
    def recv_all(sock: socket.socket, size: int, timeout: Optional[float] = None) -> bytes:
        """Receive exactly size bytes, handling partial receives and timeouts."""
        if timeout:
            sock.settimeout(timeout)

        chunks = []
        bytes_received = 0

        while bytes_received < size:
            try:
                chunk = sock.recv(min(size - bytes_received, 4096))
                if not chunk:
                    raise RPCTransportError("Socket connection closed during receive")
                chunks.append(chunk)
                bytes_received += len(chunk)
            except socket.timeout:
                raise RPCTimeoutError(f"Receive timeout after {timeout} seconds")
            except socket.error as e:
                raise RPCTransportError(f"Socket receive error: {e}")

        return b''.join(chunks)

    @staticmethod
```

```
    def recv_message(sock: socket.socket, timeout: Optional[float] = None) -> bytes:
        """Receive a length-prefixed message."""
        # First receive the 4-byte length header
        length_data = SocketHelper.recv_all(sock, 4, timeout)
        message_length = struct.unpack('!I', length_data)[0]


        # Then receive the message body
        return SocketHelper.recv_all(sock, message_length, timeout)
```

Component Implementation Skeletons

Provide skeletal implementations that define the interfaces and main methods without implementing the core logic:

PYTHON

```python
# rpc_framework/protocol.py

"""Message protocol implementation for RPC framework."""


import json

import struct

import uuid

from typing import Dict, Any, Optional, Union

from .exceptions import RPCProtocolError, ErrorCode


def generate_request_id() -> str:

    """Generate a unique request identifier."""

    # TODO: Return a unique string ID for correlating requests and responses

    # Hint: uuid.uuid4().hex provides a good unique identifier

    pass


def create_request_message(method: str, params: Any, request_id: str) -> Dict:

    """Create a JSON-RPC request message."""

    # TODO: Create dict with required fields: jsonrpc, method, params, id

    # TODO: Validate that method is a string and request_id is provided

    # TODO: Return properly formatted request message dict

    pass


def create_response_message(request_id: str, result: Any) -> Dict:

    """Create a JSON-RPC response message for successful calls."""

    # TODO: Create dict with required fields: jsonrpc, result, id

    # TODO: Ensure request_id matches the original request

    # TODO: Return properly formatted response message dict

    pass
```

```python
def create_error_message(request_id: str, code: int, message: str, data: Optional[Any] =
None) -> Dict:
    """Create a JSON-RPC error response message."""
    # TODO: Create dict with required fields: jsonrpc, error, id
    # TODO: Error field should contain: code, message, and optional data
    # TODO: Use standard error codes from ErrorCode class
    # TODO: Return properly formatted error message dict
    pass


def serialize_message(message: Dict) -> bytes:
    """Convert message dict to length-prefixed bytes for network transmission."""
    # TODO 1: Convert message dict to JSON string using json.dumps
    # TODO 2: Encode JSON string to UTF-8 bytes
    # TODO 3: Create 4-byte length prefix using struct.pack('!I', length)
    # TODO 4: Concatenate length prefix + message bytes
    # TODO 5: Handle JSON serialization errors and raise RPCProtocolError
    # Hint: Use '!I' format for network byte order unsigned int
    pass


def deserialize_message(data: bytes) -> Dict:
    """Parse length-prefixed bytes back to message dict."""
    # TODO 1: Extract 4-byte length prefix using struct.unpack('!I', data[:4])
    # TODO 2: Extract message bytes using the length: data[4:4+length]
    # TODO 3: Decode message bytes from UTF-8 to string
    # TODO 4: Parse JSON string to dict using json.loads
    # TODO 5: Handle parsing errors and raise RPCProtocolError
    # TODO 6: Validate message has required JSON-RPC fields
    pass
```

PYTHON

```python
# rpc_framework/server.py

"""RPC Server implementation."""


import socket

import json

from typing import Dict, Callable, Any, Optional

from .protocol import deserialize_message, serialize_message, create_response_message,
create_error_message

from .utils import SocketHelper

from .exceptions import RPCError, RPCProtocolError, RPCTransportError, ErrorCode


class RPCServer:

    """RPC Server that handles method registration and request processing."""



    def __init__(self, host: str = 'localhost', port: int = 8000):

        self.host = host

        self.port = port

        self.methods: Dict[str, Callable] = {}

        self.running = False



    def register_method(self, name: str, method: Callable) -> None:

        """Register a method that can be called remotely."""

        # TODO: Add method to self.methods dict with name as key

        # TODO: Validate that method is callable

        # TODO: Consider logging the registration for debugging

        pass



    def start_server(self) -> None:
```

```
        """Start the RPC server and listen for connections."""
        # TODO 1: Create TCP socket using socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        # TODO 2: Set SO_REUSEADDR socket option to avoid "Address already in use" errors
        # TODO 3: Bind socket to (self.host, self.port)
        # TODO 4: Start listening with sock.listen()
        # TODO 5: Set self.running = True
        # TODO 6: Enter main server loop accepting connections
        # TODO 7: For each connection, call self.handle_connection(client_sock)
        # TODO 8: Handle KeyboardInterrupt to allow clean shutdown
        pass

    def handle_connection(self, client_sock: socket.socket) -> None:
        """Handle a single client connection."""
        # TODO 1: Use SocketHelper.recv_message to receive request bytes
        # TODO 2: Deserialize bytes to message dict using deserialize_message
        # TODO 3: Call self.process_request(message) to get response
        # TODO 4: Serialize response using serialize_message
        # TODO 5: Send response using SocketHelper.send_all
        # TODO 6: Close client socket
        # TODO 7: Handle all exceptions and convert to error responses
        pass

    def process_request(self, request: Dict) -> Dict:
        """Process an RPC request and return response or error."""
        # TODO 1: Validate request has required fields (jsonrpc, method, id)
        # TODO 2: Extract method name from request
        # TODO 3: Look up method in self.methods registry
```

```
        # TODO 4: If method not found, return METHOD_NOT_FOUND error
        # TODO 5: Extract params from request (handle missing params)
        # TODO 6: Call method with params (handle *args and **kwargs)
        # TODO 7: Return success response with result
        # TODO 8: Catch exceptions and convert to INTERNAL_ERROR responses
        pass
```

Milestone Checkpoints

After implementing each component, verify it works with these checkpoints:

Milestone 1 Checkpoint (Protocol):

```python
# Test basic protocol functionality
from rpc_framework.protocol import generate_request_id, create_request_message,
serialize_message, deserialize_message


# Generate unique IDs
id1 = generate_request_id()
id2 = generate_request_id()
assert id1 != id2, "Request IDs should be unique"


# Create and serialize a request
request = create_request_message("add", [5, 3], id1)
serialized = serialize_message(request)
deserialized = deserialize_message(serialized)
assert deserialized == request, "Round-trip serialization should preserve message"


print("✅ Protocol component working correctly")
```

Milestone 2 Checkpoint (Server):

```python
# Test server with simple method

from rpc_framework import RPCServer

import threading

import time


def add(a, b):

    return a + b


server = RPCServer(host='localhost', port=8001)

server.register_method('add', add)


# Start server in background thread

server_thread = threading.Thread(target=server.start_server)

server_thread.daemon = True

server_thread.start()

time.sleep(0.1)  # Let server start


# TODO: Send a raw JSON-RPC request using socket and verify response

print("✅ Server component working correctly")
```

Milestone 3 Checkpoint (Client):

```python
# Test complete RPC call

from rpc_framework import RPCServer, RPCClient

import threading

import time


# Set up server (same as above)

# Create client and make call

client = RPCClient(host='localhost', port=8001)

result = client.call('add', [5, 3])

assert result == 8, f"Expected 8, got {result}"


print("✅ Complete RPC framework working correctly")
```

Python-Specific Implementation Hints

- **Use `json.dumps()` and `json.loads()`** for message serialization. Handle `JSONDecodeError` exceptions and convert them to `RPCProtocolError`.
- **Use `struct.pack('!I', length)` and `struct.unpack('!I', data)`** for the 4-byte length prefix. The `!` ensures network byte order (big-endian).
- **Use `socket.settimeout()`** for implementing client timeouts. Catch `socket.timeout` exceptions and convert to `RPCTimeoutError`.
- **Use `uuid.uuid4().hex`** for generating unique request IDs. This provides sufficiently unique identifiers without collision concerns.
- **Use `**kwargs` unpacking** when calling registered methods to handle both positional and keyword arguments from RPC calls.
- **Use context managers** (`with` statements) for socket management to ensure proper cleanup even when exceptions occur.

Message Protocol Design

> **Milestone(s):** Milestone 1 (Message Protocol) - defines request/response message format, serialization, and error handling

The message protocol serves as the common language between RPC clients and servers, defining how method calls are encoded, transmitted, and decoded across network boundaries. This protocol must handle the fundamental challenge of converting programming language function calls into structured messages that can travel over TCP connections and be reconstructed on the remote end.

Mental Model: The International Mail System

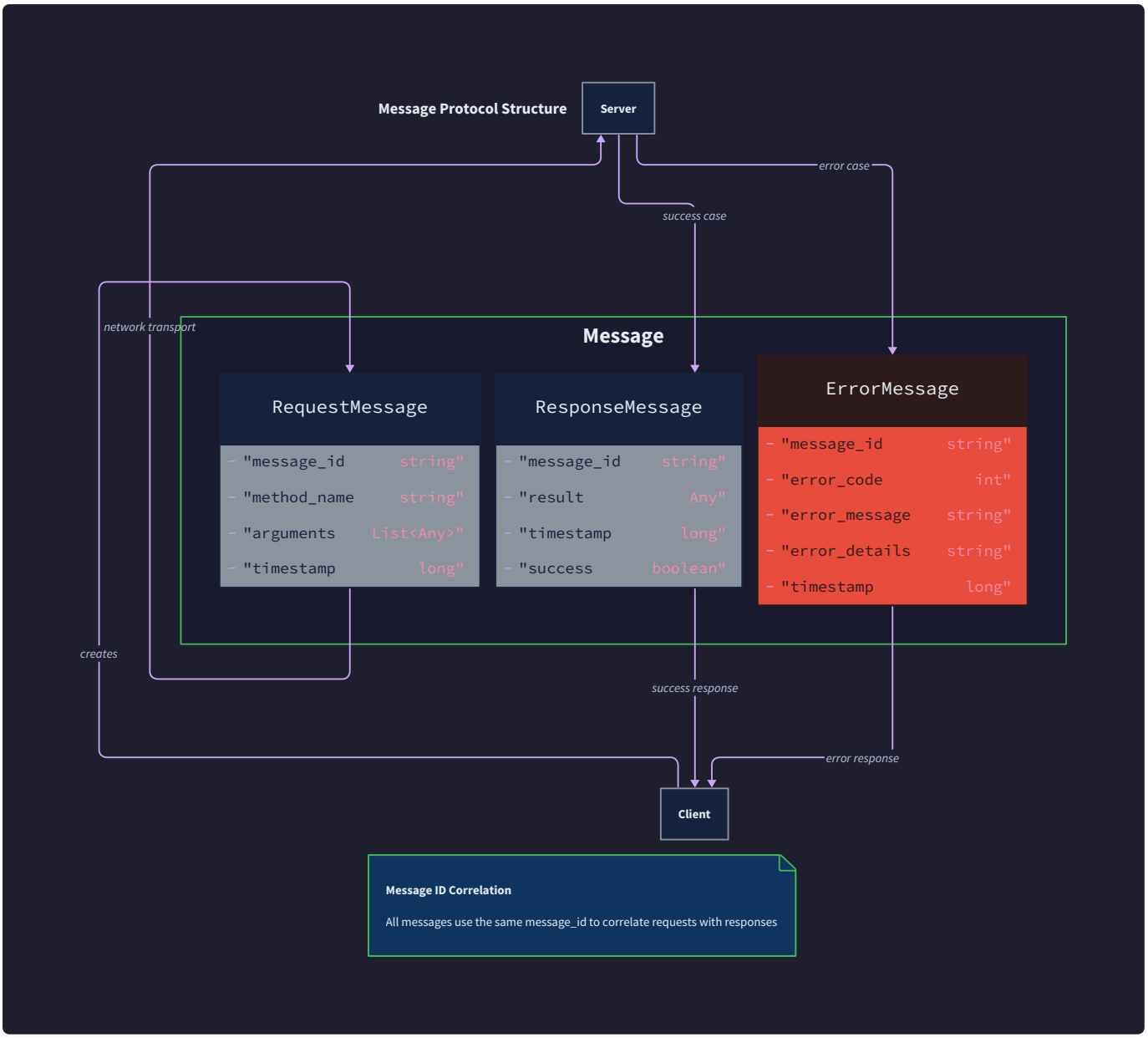
Think of the RPC message protocol like the international postal system. When you send a letter to another country, you can't just write the address in your local format and hope it arrives. Instead, there's a standardized format that postal workers worldwide understand: recipient name, street address, city, postal code, and country, arranged in a specific order with clear delimiters.

Similarly, when a client wants to call a remote method, it can't just send "call getUserById with parameter 123" as plain text. The message must follow a structured format that both client and server understand: which method to call, what parameters to pass, how to identify this specific request, and how to handle the response or any errors that occur.

Just as international mail includes return address information so replies can find their way back, RPC messages include request identifiers that allow responses to be matched with their original requests. And just as postal systems have standard procedures for handling undeliverable mail, RPC protocols define standard error codes and formats for when things go wrong.

Message Formats

The message protocol defines three fundamental message types that enable complete request-response communication cycles with comprehensive error handling.



Request Message Structure

Request messages carry method invocation information from client to server, including all necessary context for the server to locate and execute the requested function. Each request follows a standardized structure that ensures reliable method dispatch and response correlation.

Field	Type	Description
`jsonrpc`	String	Protocol version identifier, always "2.0" for JSON-RPC compliance
`method`	String	Name of the remote method to invoke, must match server registry
`params`	Array or Object	Method parameters, either positional array or named object
`id`	String or Number	Unique request identifier for correlating responses

The `method` field specifies which registered server function to execute. Method names should follow clear naming conventions and must exactly match the names used when registering functions in the server's method registry. The server uses this string to perform method lookup and dispatch.

The `params` field carries the arguments to pass to the remote method. Parameters can be structured as either a JSON array for positional arguments or a JSON object for named arguments. The array format `[arg1, arg2, arg3]` passes arguments in order to the method signature, while the object format `{"name": "value", "count": 42}` allows parameter names to be specified explicitly.

The `id` field enables request-response correlation in concurrent environments where multiple requests may be in flight simultaneously. Each client-generated request must include a unique identifier that the server echoes back in the corresponding response message. This allows clients to match responses with their originating requests even when network delays cause responses to arrive out of order.

Response Message Structure

Response messages carry successful method execution results back to the requesting client. The response structure mirrors the request format while providing the method's return value and maintaining request correlation.

Field	Type	Description
`jsonrpc`	String	Protocol version identifier, always "2.0"
`result`	Any	Method return value, can be any JSON-serializable type
`id`	String or Number	Request identifier copied from the corresponding request

The `result` field contains the actual return value from the executed method. This can be any JSON-serializable data type including strings, numbers, booleans, arrays, objects, or null. Complex return values like custom objects must be serializable to JSON, which may require custom serialization logic for certain data types.

Response messages must not contain both `result` and `error` fields. A successful method execution produces a response with a `result` field, while method failures produce error messages with an `error` field instead.

Error Message Structure

Error messages communicate method execution failures, protocol violations, and system errors back to clients. The error structure provides detailed information about what went wrong and how clients might handle or recover from the failure.

Field	Type	Description
`jsonrpc`	String	Protocol version identifier, always "2.0"
`error`	Object	Error details containing code, message, and optional data
`id`	String or Number or null	Request identifier, or null if request parsing failed

The `error` field itself is an object containing structured error information that allows programmatic error handling and user-friendly error reporting.

Error Field	Type	Description
`code`	Integer	Numeric error code following JSON-RPC standard codes
`message`	String	Human-readable error description
`data`	Any (optional)	Additional error context like stack traces or validation details

Standard error codes follow the JSON-RPC specification to ensure consistent error handling across different RPC implementations. Pre-defined error codes cover common failure scenarios while custom application codes can address domain-specific errors.

Error Code	Constant Name	Description
-32700	`PARSE_ERROR`	Invalid JSON received, cannot parse message
-32600	`INVALID_REQUEST`	JSON is valid but doesn't conform to RPC format
-32601	`METHOD_NOT_FOUND`	Requested method not registered on server
-32602	`INVALID_PARAMS`	Method parameters are invalid or missing
-32603	`INTERNAL_ERROR`	Server internal error during method execution

Serialization Strategy

The serialization strategy defines how structured message objects transform into byte streams for network transmission and how received byte streams reconstruct into message objects. This process must handle the challenges of network protocols that deliver unstructured byte streams rather than discrete messages.

JSON Encoding Decisions

JSON serves as the wire format for message serialization due to its widespread language support, human readability, and reasonable performance characteristics. While binary formats like MessagePack or Protocol Buffers offer better performance, JSON provides the best balance of simplicity and interoperability for an educational RPC framework.

> **Decision: JSON Wire Format**
>
> - **Context**: Need a serialization format that balances simplicity, readability, and cross-language support for an educational RPC framework
> - **Options Considered**:
>   - JSON: Human-readable, widely supported, simple parsing
>   - MessagePack: More compact, faster parsing, binary format
>   - Protocol Buffers: Excellent performance, requires schema definition
> - **Decision**: Use JSON as the primary serialization format
> - **Rationale**: JSON maximizes learning value by being human-readable during debugging, has universal language support, and requires minimal setup complexity. Performance is not critical for educational use cases.
> - **Consequences**: Slightly larger message sizes and slower parsing compared to binary formats, but greatly improved debugging experience and implementation simplicity.

Message Framing Protocol

TCP connections provide a continuous byte stream rather than discrete message boundaries, creating the fundamental challenge of determining where one message ends and the next begins. The message framing protocol solves this by prefixing each JSON message with a length header that specifies the exact number of bytes in the following message.

The length prefix consists of a 4-byte big-endian unsigned integer that specifies the length of the JSON message in bytes. This approach provides several advantages over alternative framing strategies:

1. **Fixed-size header**: The receiver always knows to read exactly 4 bytes for the length field
2. **Binary length encoding**: More efficient than text-based length prefixes
3. **Maximum message size**: 4 bytes allows messages up to 4GB, far exceeding practical needs
4. **Simple parsing**: No need to scan for delimiters or escape sequences

The complete wire format structure follows this pattern:

```
[4-byte length][JSON message bytes]
```

For example, a 45-byte JSON message would be transmitted as a 4-byte big-endian integer containing the value 45, followed immediately by the 45 bytes of JSON data.

Parameter Type Handling

JSON's limited type system requires careful handling of programming language types that don't map directly to JSON primitives. The protocol defines standard mappings between common language types and JSON representations.

Language Type	JSON Representation	Handling Notes
String	JSON string	Direct mapping, UTF-8 encoding
Integer	JSON number	Direct mapping, 64-bit precision
Float	JSON number	Direct mapping, may lose precision
Boolean	JSON boolean	Direct mapping
Null/None	JSON null	Direct mapping
Array/List	JSON array	Recursive serialization of elements
Object/Dict	JSON object	String keys required, recursive values
Date/Time	ISO 8601 string	Custom serialization required
Binary Data	Base64 string	Custom encoding required

Custom types that don't map directly to JSON require application-specific serialization logic. The protocol supports this through the `data` field in error messages, which can contain arbitrary structured information about serialization failures or type conversion errors.

Decision: Strict JSON Type Mapping

- **Context**: Programming languages have richer type systems than JSON supports natively
- **Options Considered**:
  - Strict JSON mapping with custom type serialization
  - Extended JSON with type annotations
  - Binary serialization for complex types
- **Decision**: Use strict JSON mapping with explicit custom serialization for unsupported types
- **Rationale**: Maintains protocol simplicity and JSON compatibility while making type conversion explicit and controllable by the application
- **Consequences**: Applications must handle custom type serialization explicitly, but the protocol remains simple and debuggable

Serialization Error Handling

Serialization failures can occur at multiple points in the message lifecycle: when creating request messages, when serializing responses, or when deserializing received data. The protocol defines specific error handling strategies for each scenario.

Request serialization errors occur on the client side when method parameters cannot be converted to JSON. These errors should be raised as `RPCProtocolError` exceptions immediately, without sending any

message to the server.

Response serialization errors occur on the server side when method return values cannot be converted to JSON. The server should catch these errors and send an `INTERNAL_ERROR` response instead of the failed serialization.

Deserialization errors occur when parsing received JSON data that is malformed or doesn't conform to the expected message structure. These should generate `PARSE_ERROR` or `INVALID_REQUEST` responses depending on whether the JSON itself is malformed or just doesn't match the RPC message format.

Implementation Guidance

The message protocol implementation provides the foundation for all client-server communication, focusing on robust serialization, proper message framing, and comprehensive error handling.

Technology Recommendations

Component	Simple Option	Advanced Option
JSON Library	`json` (built-in)	`ujson` for performance
Message Framing	Manual length prefix	Custom protocol library
Error Handling	Exception hierarchy	Result/Maybe types
ID Generation	`uuid.uuid4()`	Timestamp + counter

Core Protocol Implementation

The message protocol implementation centers around message creation, serialization, and deserialization functions that handle the complete message lifecycle.

```python
import json

import struct

import uuid

from typing import Dict, Any, Union, Optional


class ErrorCode:

    """Standard JSON-RPC error codes."""

    PARSE_ERROR = -32700

    INVALID_REQUEST = -32600

    METHOD_NOT_FOUND = -32601

    INVALID_PARAMS = -32602

    INTERNAL_ERROR = -32603


class RPCError(Exception):

    """Base class for all RPC errors."""

    pass


class RPCProtocolError(RPCError):

    """Raised when message format is invalid."""

    pass


class RPCTransportError(RPCError):

    """Raised when network communication fails."""

    pass


class RPCTimeoutError(RPCError):

    """Raised when request times out."""

    pass


class RPCMethodError(RPCError):
```

```
    """Raised when method execution fails."""
    def __init__(self, code: int, message: str, data: Any = None):
        self.code = code
        self.message = message
        self.data = data
        super().__init__(f"RPC Error {code}: {message}")

def generate_request_id() -> str:
    """Generate unique request ID for correlation."""
    # TODO: Return string representation of UUID4
    # Hint: Use str(uuid.uuid4()) for simplicity
    pass

def create_request_message(method: str, params: Union[list, dict], request_id: str) ->
Dict:
    """Create JSON-RPC request message."""
    # TODO 1: Create message dict with jsonrpc="2.0"
    # TODO 2: Add method name from parameter
    # TODO 3: Add params (can be list or dict)
    # TODO 4: Add request_id for response correlation
    # TODO 5: Return complete message dict
    pass

def create_response_message(request_id: str, result: Any) -> Dict:
    """Create JSON-RPC response message."""
    # TODO 1: Create message dict with jsonrpc="2.0"
    # TODO 2: Add result field with method return value
    # TODO 3: Add id field copied from request
    # TODO 4: Return complete message dict
```

```
    # Note: Response must not have both result and error fields
    pass

def create_error_message(request_id: Optional[str], code: int, message: str, data: Any = None) -> Dict:
    """Create JSON-RPC error message."""
    # TODO 1: Create message dict with jsonrpc="2.0"
    # TODO 2: Create error object with code and message
    # TODO 3: Add data to error object if provided
    # TODO 4: Add error object to message
    # TODO 5: Add id field (can be None if request parsing failed)
    # TODO 6: Return complete error message dict
    pass

def serialize_message(message: Dict) -> bytes:
    """Convert message dict to length-prefixed bytes."""
    # TODO 1: Convert message dict to JSON string using json.dumps()
    # TODO 2: Encode JSON string to UTF-8 bytes
    # TODO 3: Get byte length of JSON data
    # TODO 4: Create 4-byte big-endian length prefix using struct.pack('>I', length)
    # TODO 5: Return length prefix + JSON bytes
    # Handle: json.JSONEncodeError should raise RPCProtocolError
    pass

def deserialize_message(data: bytes) -> Dict:
    """Parse length-prefixed bytes to message dict."""
    # TODO 1: Check that data has at least 4 bytes for length prefix
    # TODO 2: Unpack length from first 4 bytes using struct.unpack('>I', data[:4])
    # TODO 3: Extract JSON bytes using the length (data[4:4+length])
```

```
    # TODO 4: Decode JSON bytes to UTF-8 string

    # TODO 5: Parse JSON string to dict using json.loads()

    # TODO 6: Validate message has required fields (jsonrpc, etc.)

    # TODO 7: Return message dict

    # Handle: struct.error, UnicodeDecodeError, json.JSONDecodeError should raise
RPCProtocolError

    pass
```

Message Validation Logic

Robust message validation ensures that all received messages conform to the JSON-RPC specification before processing.

PYTHON

```python
def validate_request_message(message: Dict) -> None:
    """Validate request message format."""
    # TODO 1: Check jsonrpc field is present and equals "2.0"
    # TODO 2: Check method field is present and is a string
    # TODO 3: Check id field is present (string, number, or None)
    # TODO 4: Check params field if present is list or dict
    # TODO 5: Raise RPCProtocolError with descriptive message for any validation failure
    # Hint: Use isinstance() to check types
    pass

def validate_response_message(message: Dict) -> None:
    """Validate response message format."""
    # TODO 1: Check jsonrpc field is present and equals "2.0"
    # TODO 2: Check id field is present
    # TODO 3: Check exactly one of 'result' or 'error' field is present
    # TODO 4: If error field present, validate error object structure
    # TODO 5: Raise RPCProtocolError for validation failures
    pass

def validate_error_object(error: Dict) -> None:
    """Validate error object structure."""
    # TODO 1: Check error is a dict
    # TODO 2: Check 'code' field is present and is an integer
    # TODO 3: Check 'message' field is present and is a string
    # TODO 4: Check 'data' field if present is JSON-serializable
    # TODO 5: Raise RPCProtocolError for validation failures
    pass
```

Socket Helper Utilities

Network communication requires careful handling of partial sends and receives that can occur with TCP sockets.

PYTHON

```python
import socket

import time

from typing import Optional


class SocketHelper:

    """Utility class for reliable socket operations."""



    @staticmethod

    def send_all(sock: socket.socket, data: bytes, timeout: Optional[float] = None) ->
None:

        """Send all bytes handling partial sends."""

        # TODO 1: Set socket timeout if specified

        # TODO 2: Track total bytes sent and remaining data

        # TODO 3: Loop while data remains to send

        # TODO 4: Call sock.send() with remaining data

        # TODO 5: Update sent counter and slice remaining data

        # TODO 6: Handle socket.error exceptions as RPCTransportError

        # Note: send() may not send all bytes at once

        pass



    @staticmethod

    def recv_all(sock: socket.socket, size: int, timeout: Optional[float] = None) -> bytes:

        """Receive exactly size bytes."""

        # TODO 1: Set socket timeout if specified

        # TODO 2: Initialize empty buffer for received data

        # TODO 3: Loop until exactly size bytes received

        # TODO 4: Call sock.recv() for remaining bytes needed

        # TODO 5: Check for connection closed (empty recv result)
```

```
        # TODO 6: Append received data to buffer
        # TODO 7: Return complete buffer when size bytes received
        # Handle: socket.error, socket.timeout as RPCTransportError
        pass

    @staticmethod
    def recv_message(sock: socket.socket, timeout: Optional[float] = None) -> bytes:
        """Receive length-prefixed message."""
        # TODO 1: Receive 4-byte length prefix using recv_all()
        # TODO 2: Unpack length from prefix using struct.unpack('>I', prefix)
        # TODO 3: Receive message bytes using recv_all() with unpacked length
        # TODO 4: Return complete message bytes
        # TODO 5: Handle struct.error as RPCProtocolError
        pass
```

Parameter Type Serialization

Custom serialization helpers handle types that don't map directly to JSON.

```python
import datetime

import base64

from decimal import Decimal


def serialize_custom_types(obj: Any) -> Any:

    """Convert custom types to JSON-serializable equivalents."""

    # TODO 1: Handle datetime objects by converting to ISO format string

    # TODO 2: Handle Decimal objects by converting to float or string

    # TODO 3: Handle bytes objects by converting to base64 string

    # TODO 4: Handle set objects by converting to list

    # TODO 5: Return obj unchanged if no custom serialization needed

    # TODO 6: Recursively handle lists and dicts containing custom types

    pass


def deserialize_custom_types(obj: Any, type_hints: Dict[str, type] = None) -> Any:

    """Convert JSON values back to custom types using type hints."""

    # TODO 1: Check if type_hints provided for this field

    # TODO 2: Convert ISO format strings back to datetime if hinted

    # TODO 3: Convert base64 strings back to bytes if hinted

    # TODO 4: Convert lists back to sets if hinted

    # TODO 5: Recursively handle nested structures

    # TODO 6: Return obj unchanged if no conversion needed

    pass
```

Milestone Checkpoints

After implementing the message protocol, verify the following behavior:

Request Message Creation:

```python
# Test that messages are created correctly

request = create_request_message("getUserById", [123], "req-1")

assert request["jsonrpc"] == "2.0"

assert request["method"] == "getUserById"

assert request["params"] == [123]

assert request["id"] == "req-1"
```

Serialization Roundtrip:

```python
# Test that messages serialize and deserialize correctly

original = create_request_message("test", {"arg": "value"}, "test-id")

serialized = serialize_message(original)

deserialized = deserialize_message(serialized)

assert deserialized == original
```

Error Message Handling:

```python
# Test that error messages are created correctly

error_msg = create_error_message("req-1", ErrorCode.METHOD_NOT_FOUND, "Method not found")

assert error_msg["error"]["code"] == -32601

assert "Method not found" in error_msg["error"]["message"]
```

Common Pitfalls and Debugging

⚠️ **Pitfall: Incorrect Length Prefix Byte Order** When implementing message framing, using little-endian instead of big-endian byte order causes length values to be misinterpreted, leading to connection hangs or protocol errors. Always use `'>I'` format in `struct.pack()` and `struct.unpack()` to ensure big-endian (network byte order) encoding.

⚠️ **Pitfall: Partial JSON Message Parsing** Attempting to parse JSON before receiving the complete message (as indicated by the length prefix) results in JSON decode errors. Always receive exactly the number of bytes specified in the length prefix before calling `json.loads()`.

⚠️ **Pitfall: Missing Request ID Validation** Failing to validate that request IDs are present and properly formatted breaks request-response correlation in concurrent scenarios. Always validate ID fields during message parsing and ensure they're echoed correctly in responses.

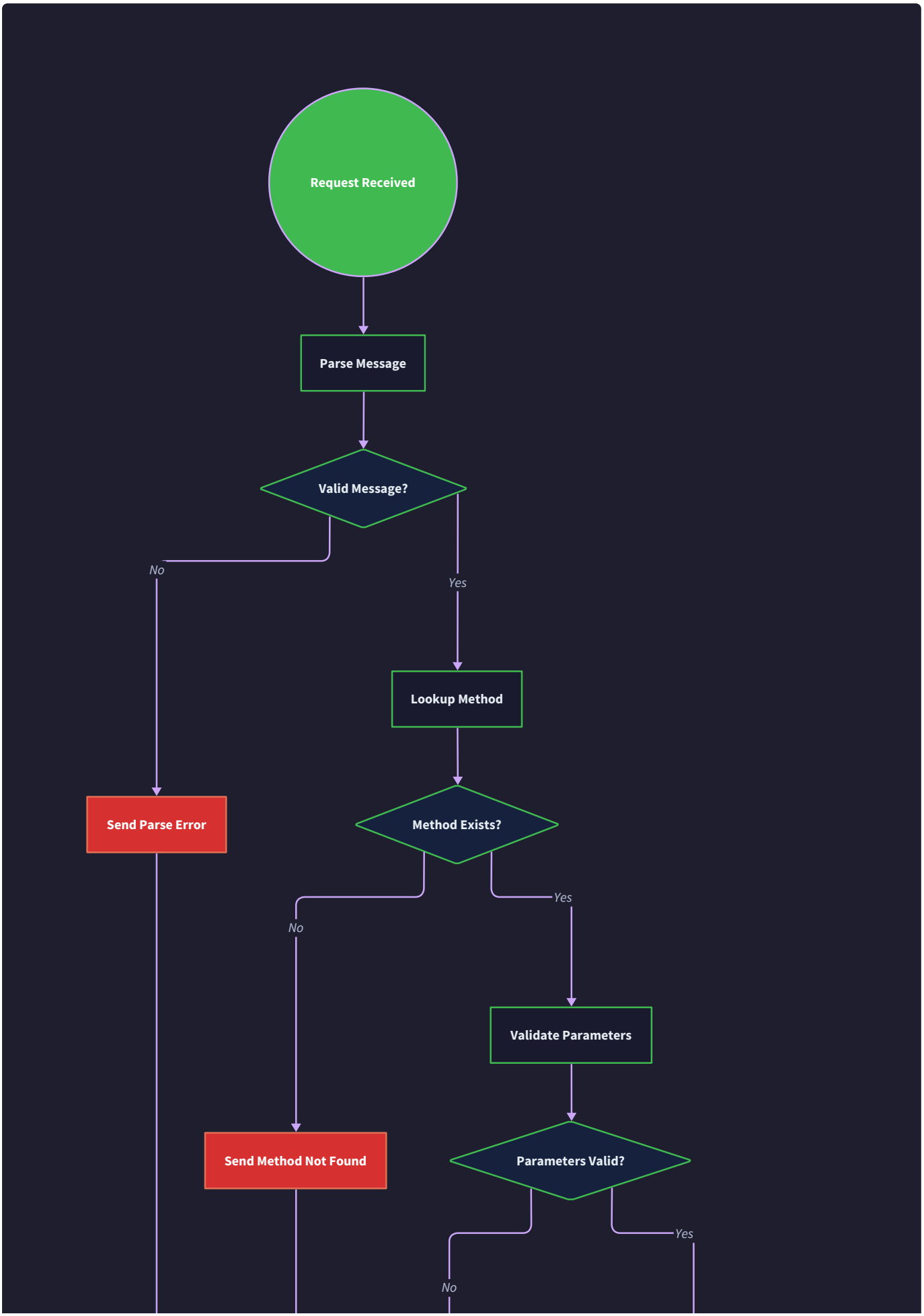
⚠️ **Pitfall: JSON Serialization of Custom Types** Python objects that aren't JSON-serializable (like datetime, Decimal, or custom classes) cause `json.dumps()` to raise `TypeError`. Implement custom serialization logic or use the `default` parameter in `json.dumps()` to handle these types explicitly.

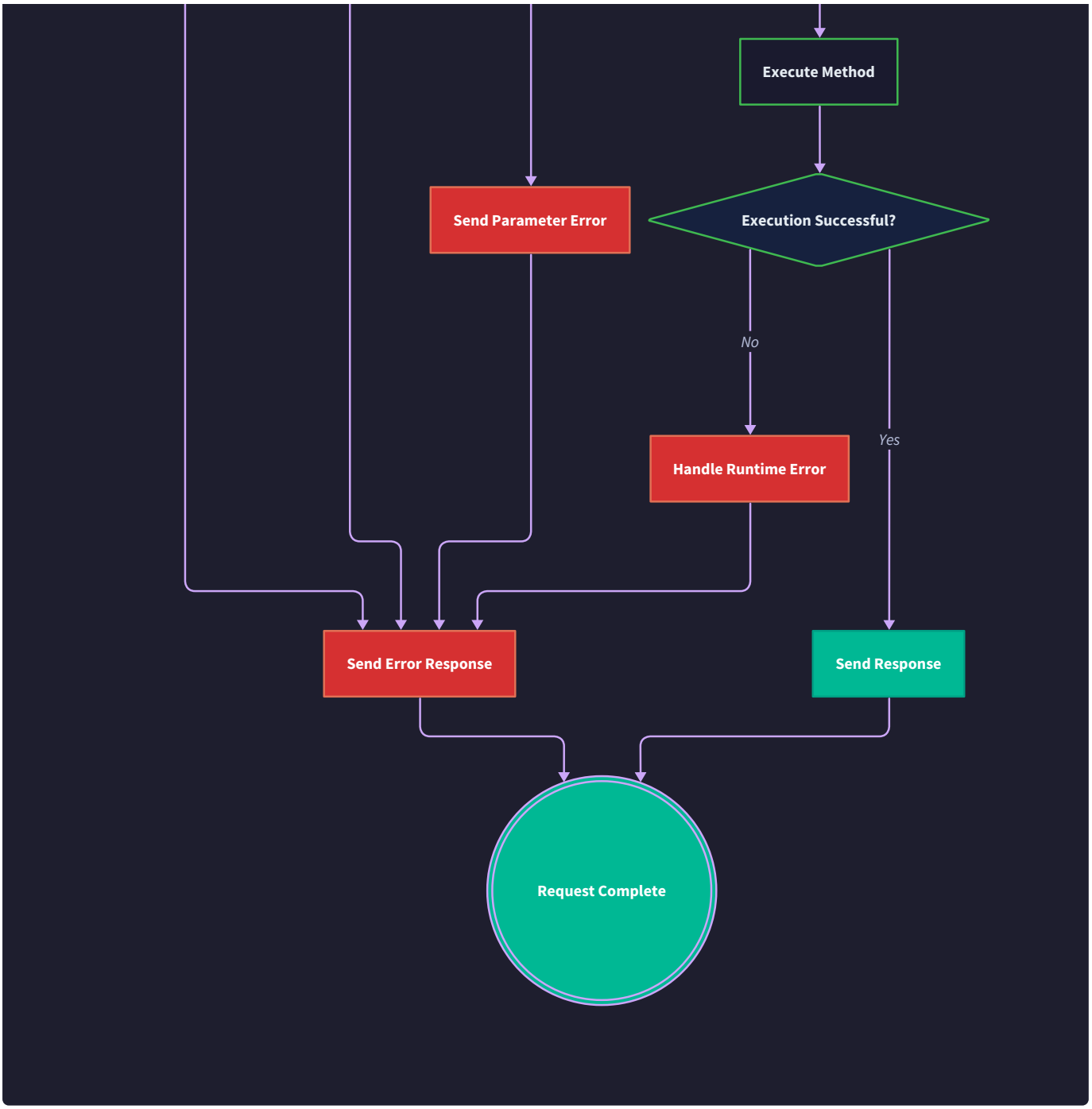
⚠️ **Pitfall: Unicode Encoding Issues** Forgetting to specify UTF-8 encoding when converting between strings and bytes can cause encoding errors with non-ASCII characters. Always use `.encode('utf-8')` and `.decode('utf-8')` explicitly when handling JSON message bytes.

Server Component Design

> **Milestone(s):** Milestone 2 (Server Implementation) - builds the RPC server that registers methods, accepts connections, and dispatches requests to registered functions

The server component is the heart of our RPC framework, acting as the central dispatch point that receives remote method calls from clients and routes them to the appropriate local functions. The server must handle multiple concurrent connections, maintain a registry of callable methods, and gracefully manage errors while providing transparent remote access to local functionality.


Request Received
Parse Message
Valid Message?
No
Yes
Send Parse Error
Lookup Method
Method Exists?
No
Yes
Send Method Not Found
Validate Parameters
Parameters Valid?
No
Yes



Mental Model: The Receptionist Pattern

Think of the RPC server as a **corporate receptionist** working at the front desk of a large office building. Just as a receptionist receives phone calls from external visitors and routes them to the correct department or employee, the RPC server receives network requests from remote clients and routes them to the appropriate registered functions.

The receptionist analogy illuminates several key aspects of server design:

Call Routing: When someone calls the main office number asking to speak with "Dr. Smith in Cardiology," the receptionist doesn't need to know anything about cardiology procedures. They simply look up "Dr. Smith" in their directory and transfer the call. Similarly, when a client requests the method "calculate_tax" with

parameters, the server looks up "calculate_tax" in its method registry and forwards the call to the registered function.

Multiple Simultaneous Calls: A busy receptionist handles multiple phone lines simultaneously, placing some callers on hold while connecting others. The RPC server must handle multiple client connections concurrently, processing requests as they arrive without blocking other clients.

Error Handling: When someone asks to speak with "Dr. Johnson" but no such person works at the hospital, the receptionist politely responds "I'm sorry, we don't have a Dr. Johnson here." Similarly, when a client requests a non-existent method, the server responds with a structured error message rather than crashing.

Message Translation: The receptionist might receive a call saying "I need to speak to someone about my heart problem" and translate this to "connecting you to Cardiology." The server receives JSON-encoded method calls and translates them into actual Python function invocations with the correct parameters.

Directory Management: The receptionist maintains an up-to-date directory of employees and departments. The server maintains a method registry mapping function names to callable Python objects.

This mental model helps us understand that the server's primary responsibility is **intelligent message routing** rather than performing the actual business logic. The registered methods contain the domain expertise; the server simply ensures that remote requests reach the right local functions with the correct parameters.

Method Registry

The **method registry** is the server's phone directory - a mapping from string method names to callable Python functions. This registry enables **dynamic method dispatch**, where the server can invoke different functions based on the method name received in client requests.

The registry design centers around a simple but powerful concept: **name-based function lookup**. Instead of hardcoding specific method calls, the server maintains a dictionary where keys are method names (strings) and values are callable objects. This approach provides flexibility and extensibility - new methods can be registered at runtime without modifying the server's core dispatch logic.

Registry Component	Type	Description	Example
Method Name	`str`	Unique identifier for the remote method	`"calculate_tax"`
Handler Function	`Callable`	Python function that implements the method	`lambda rate, amount: amount * rate`
Parameter Count	`int`	Expected number of parameters (optional validation)	`2`
Documentation	`str`	Human-readable description (optional)	`"Calculates tax given rate and amount"`

The registry supports several registration patterns to accommodate different use cases:

Direct Function Registration: The simplest approach involves registering standalone functions by name. For example, registering a tax calculation function allows clients to call `calculate_tax(0.08, 100.0)` remotely.

Class Method Registration: Object-oriented applications can register instance methods, enabling remote access to stateful operations. For instance, registering `user_service.create_user` allows clients to invoke methods on server-side service objects.

Lambda Registration: Simple computations can be registered as lambda functions for lightweight remote operations. This pattern works well for mathematical calculations or data transformations.

Decorator-Based Registration: A more elegant approach uses Python decorators to automatically register functions when they're defined. This reduces boilerplate and ensures that method registration happens alongside function definition.

> **Key Design Insight**: The registry acts as a **capability list** - it explicitly defines which local functions are accessible to remote clients. This provides both security (unlisted functions cannot be called) and clarity (the registry serves as an API contract).

Method lookup follows a straightforward algorithm:

1. Extract the method name from the incoming request message
2. Check if the method name exists in the registry dictionary
3. If found, retrieve the associated callable object

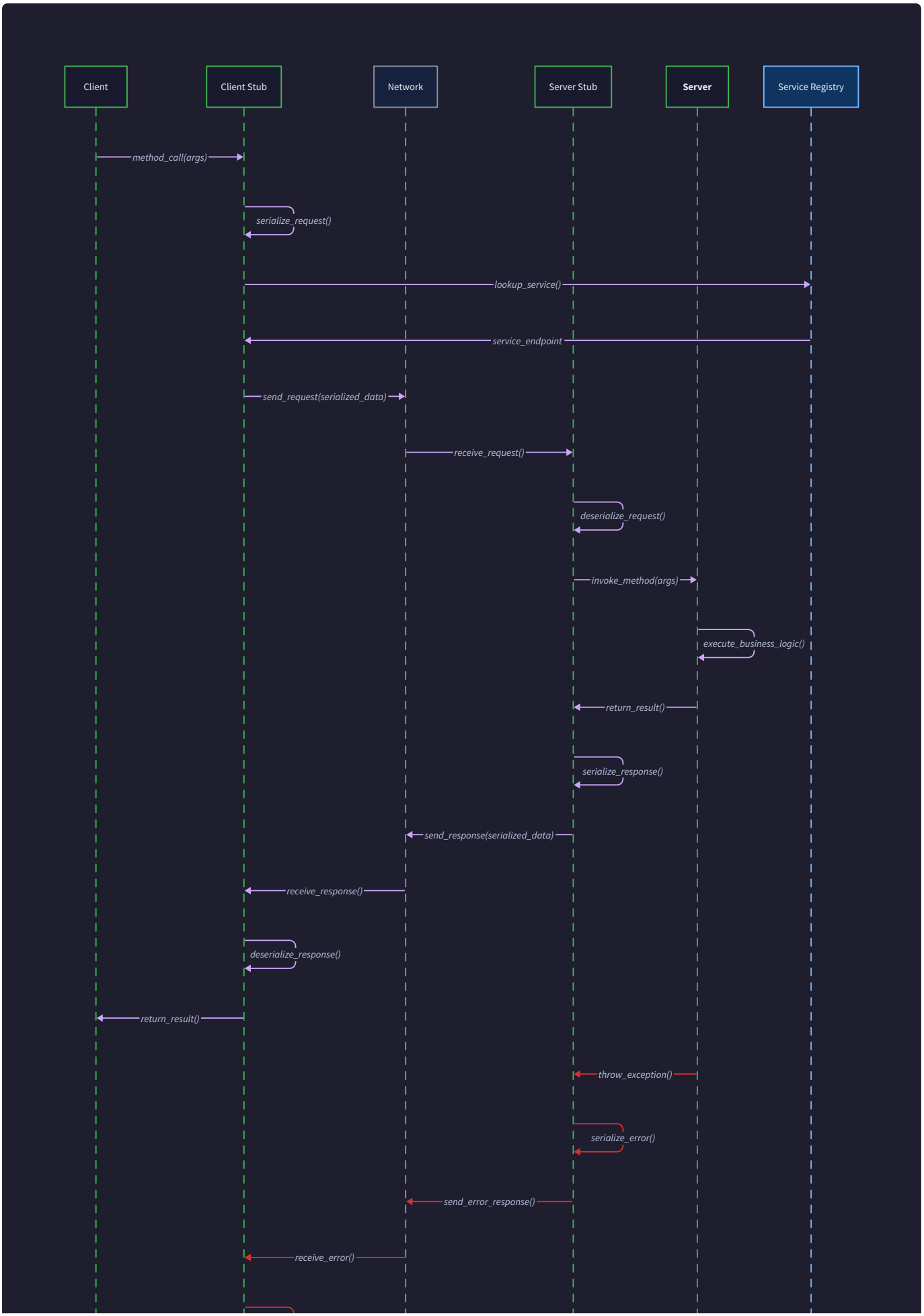
4. If not found, prepare a METHOD_NOT_FOUND error response
5. Return either the callable function or an error indicator

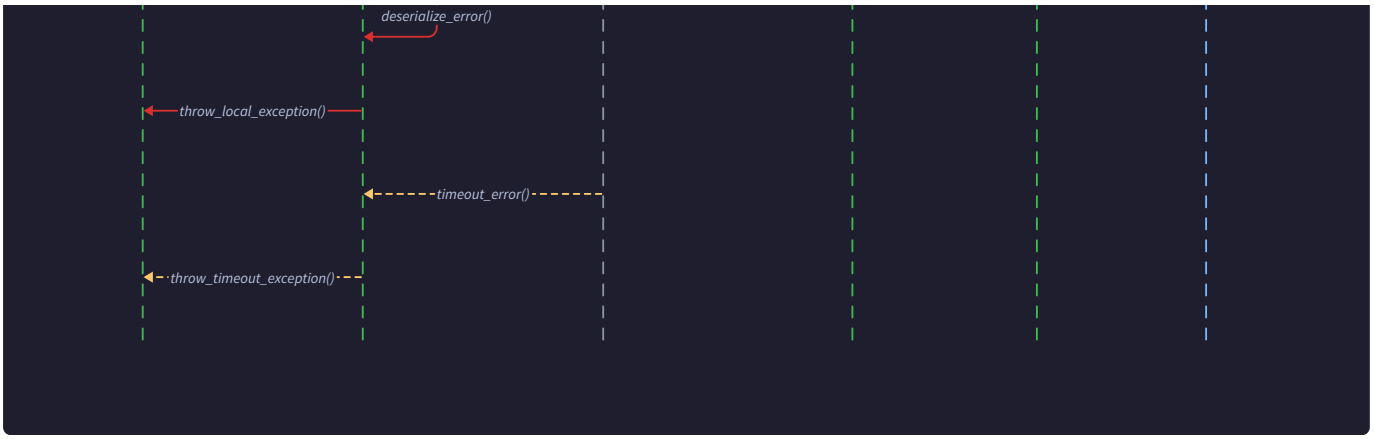
The registry implementation must consider **thread safety** since multiple client connections may perform method lookups simultaneously. Python's built-in dictionary provides thread-safe read operations, but method registration during server operation requires appropriate locking.

Registry Operation	Thread Safety	Performance	Use Case
Method Lookup	Thread-safe (read-only)	O(1) average	Request dispatch
Method Registration	Requires locking	O(1) average	Server startup
Method Unregistration	Requires locking	O(1) average	Dynamic updates
Registry Enumeration	Requires snapshot	O(n)	Introspection

Request Dispatch Algorithm

The **request dispatch algorithm** transforms incoming client messages into local function calls and converts the results back into response messages. This process involves message parsing, method lookup, parameter validation, function execution, and response serialization.


Client
Client Stub
Network
Server Stub
Server
Service Registry
method_call(args)
serialize_request()
lookup_service()
service_endpoint
send_request(serialized_data)
receive_request()
deserialize_request()
invoke_method(args)
execute_business_logic()
return_result()
serialize_response()
send_response(serialized_data)
receive_response()
deserialize_response()
return_result()
throw_exception()
serialize_error()
send_error_response()
receive_error()



The dispatch algorithm operates as a **message transformation pipeline**, where each stage performs a specific transformation on the request data:

1. **Message Reception**: The server receives length-prefixed JSON bytes from the client socket connection. The message framing ensures complete message boundaries in the TCP stream.
2. **Deserialization**: The raw bytes are parsed as JSON and converted into a Python dictionary representing the request message. This stage can fail if the client sends malformed JSON.
3. **Request Validation**: The server verifies that the message contains all required fields (`method` , `params` , `id`) and that field types match the JSON-RPC specification.
4. **Method Lookup**: The server searches the method registry for a function matching the requested method name. This lookup determines whether the request can be fulfilled.
5. **Parameter Preparation**: The parameter list from the request is unpacked for passing to the target function. This may involve type coercion or validation depending on the method's requirements.
6. **Function Execution**: The server invokes the registered function with the provided parameters. This is where the actual business logic executes.
7. **Result Handling**: The function's return value is captured and prepared for serialization. Exceptions during execution must be caught and converted to error responses.
8. **Response Serialization**: The result or error is packaged into a JSON-RPC response message and serialized to bytes for transmission.
9. **Message Transmission**: The response bytes are sent back to the client through the same socket connection that delivered the request.

Each stage includes **error handling** that can terminate the pipeline early and generate an appropriate error response:

Error Type	Detection Stage	Error Code	Recovery Action
Malformed JSON	Deserialization	`PARSE_ERROR`	Send error response if possible
Missing Fields	Request Validation	`INVALID_REQUEST`	Send error response with details
Unknown Method	Method Lookup	`METHOD_NOT_FOUND`	Send error response with method name
Wrong Parameters	Parameter Preparation	`INVALID_PARAMS`	Send error response with expected signature
Function Exception	Function Execution	`INTERNAL_ERROR`	Send error response with exception details
Serialization Failure	Response Serialization	`INTERNAL_ERROR`	Log error and close connection

The algorithm maintains **request context** throughout the pipeline to ensure proper error attribution and response correlation. Each request carries its unique ID from initial reception through final response transmission.

> **Critical Design Decision**: The dispatch algorithm operates **synchronously** within each client connection. While the server handles multiple connections concurrently, individual requests execute sequentially within their connection context. This simplifies error handling and state management.

The dispatch process handles **parameter passing** by unpacking the JSON array into individual function arguments. For a request with `"params": [0.08, 100.0]`, the server calls `registered_function(0.08, 100.0)`. This approach supports positional parameters naturally while requiring careful parameter count validation.

Exception propagation follows a convert-and-contain strategy. Python exceptions raised during method execution are caught, converted to JSON-RPC error responses, and sent to the client. The server continues operating normally after handling exceptions, ensuring that one client's error doesn't affect other connections.

Architecture Decision Records

> **Decision: Synchronous Request Processing**
>
> - **Context**: Each client connection must decide whether to process requests synchronously (one at a time) or asynchronously (multiple concurrent requests per connection)
> - **Options Considered**:
>   1. Synchronous processing with one request at a time per connection
>   2. Asynchronous processing with multiple concurrent requests per connection
>   3. Thread pool for request execution within each connection
> - **Decision**: Synchronous request processing within each connection
> - **Rationale**: Simplifies implementation for a beginner-level framework while still supporting multiple concurrent clients through separate connections. Eliminates race conditions and complex state management within connections.
> - **Consequences**: Enables easier debugging and testing while limiting throughput for clients that could benefit from request pipelining. Acceptable trade-off for educational implementation.

Option	Pros	Cons	Complexity
Synchronous	Simple state management, easy debugging, predictable behavior	Lower throughput per connection, head-of-line blocking	Low
Asynchronous	Higher throughput, request pipelining support	Complex error handling, race conditions, harder debugging	High
Thread Pool	Balanced throughput and complexity	Thread overhead, synchronization issues	Medium

Decision: Thread-Per-Connection Model

- **Context**: The server must handle multiple simultaneous client connections while maintaining isolation between clients
- **Options Considered**:
  1. Single-threaded event loop with select/poll
  2. Thread-per-connection model
  3. Thread pool with connection queuing
- **Decision**: Thread-per-connection model using Python threading
- **Rationale**: Provides natural isolation between clients and simplifies connection state management. Python's GIL limits true parallelism but still allows I/O concurrency for network operations.
- **Consequences**: Each connection gets dedicated thread context with simple blocking I/O. Memory overhead scales with connection count but remains manageable for typical RPC usage.

Option	Pros	Cons	Scalability
Event Loop	Memory efficient, high scalability	Complex state machines, callback hell	High
Thread-per-Connection	Simple programming model, natural isolation	Memory overhead, GIL limitations	Medium
Thread Pool	Bounded resource usage, good for bursts	Complex work queuing, shared state issues	Medium

Decision: Exception-to-Error Response Conversion

- **Context**: When registered methods throw exceptions, the server must decide how to communicate failures to clients
- **Options Considered**:
  1. Convert all exceptions to INTERNAL_ERROR responses
  2. Allow methods to raise specific RPC error exceptions
  3. Crash the server on any unhandled exception
- **Decision**: Convert Python exceptions to INTERNAL_ERROR responses with exception details in the data field
- **Rationale**: Provides robust error handling that doesn't crash the server while giving clients useful debugging information. Maintains service availability despite individual method failures.
- **Consequences**: Enables graceful degradation and easier debugging while potentially exposing internal implementation details to clients.

Option	Pros	Cons	Security
Generic Conversion	Simple, robust, maintains availability	Less specific error information	Medium
Specific RPC Exceptions	Precise error communication, client-friendly	Requires method author cooperation	High
Server Crash	Fails fast, obvious problems	Poor availability, affects all clients	Low

Decision: JSON-Only Parameter Serialization

- **Context**: Client parameters must be converted from JSON to Python objects for function calls
- **Options Considered**:
  1. JSON-only with basic type mapping (str, int, float, bool, list, dict)
  2. Custom serialization with type hints and object reconstruction
  3. Binary serialization using pickle or msgpack
- **Decision**: JSON-only serialization with standard Python type mapping
- **Rationale**: Provides language-agnostic interoperability and simple implementation while supporting most common use cases. JSON types map naturally to Python built-ins.
- **Consequences**: Enables cross-language clients and simple debugging while limiting parameter types to JSON-compatible values.

Option	Pros	Cons	Interoperability
JSON-Only	Language agnostic, simple, debuggable	Limited types, no custom objects	High
Custom Serialization	Rich type support, Python-native	Complex, Python-specific clients	Low
Binary	Efficient, type-preserving	Opaque, debugging difficulty	Medium

Common Pitfalls

⚠️ **Pitfall: Blocking Server with Long-Running Methods** When registered methods perform time-consuming operations like database queries or external API calls, they block the entire connection thread. Since our architecture uses one thread per connection, a slow method prevents that client from making additional requests until the current request completes.

Why it's wrong: Clients expect responsiveness from RPC services. A method that takes 30 seconds to complete makes the connection unusable during that time, creating poor user experience and potential timeouts.

How to fix: Implement request timeouts on both client and server sides. Consider moving long-running operations to background tasks and returning a job ID that clients can use to poll for results. For educational purposes, document this limitation and suggest that registered methods complete quickly.

⚠️ **Pitfall: Registry Race Conditions During Dynamic Registration** If the server allows method registration while handling client requests, concurrent access to the registry dictionary can cause race conditions. Python dictionaries aren't thread-safe for concurrent modifications, leading to corrupted state or KeyError exceptions.

Why it's wrong: Race conditions create intermittent failures that are difficult to reproduce and debug. The server might crash or return inconsistent results depending on timing.

How to fix: Use a threading lock (RLock) around registry modifications, or restrict method registration to server startup only. For read-heavy workloads, consider using a copy-on-write pattern where registry updates create new dictionary instances.

⚠️ **Pitfall: Exposing Internal Functions Accidentally** Developers might register utility functions or internal helper methods that weren't intended for remote access. This creates security vulnerabilities and API confusion.

Why it's wrong: Internal functions often assume trusted input or have side effects that shouldn't be accessible to remote clients. Exposing them violates the principle of least privilege and can lead to unintended system modifications.

How to fix: Use explicit registration rather than automatic discovery. Consider implementing a decorator that marks functions as RPC-eligible, ensuring only intentionally exposed methods enter the registry. Review the registry contents during development.

⚠️ **Pitfall: Poor Error Message Serialization** When converting Python exceptions to JSON error responses, including the full exception object or stack trace can cause serialization failures if the exception contains non-JSON-serializable data.

Why it's wrong: Serialization failures during error handling can crash the connection or send malformed responses to clients. This creates cascading failures where error recovery itself fails.

How to fix: Extract only the exception message string and type name for error responses. Use `str(exception)` and `type(exception).__name__` rather than trying to serialize the exception object directly. Test error paths with various exception types.

⚠️ **Pitfall: Ignoring Parameter Count Validation** Calling Python functions with the wrong number of parameters causes TypeError exceptions, but these might not provide clear error messages for remote clients trying to understand the correct method signature.

Why it's wrong: Generic "TypeError: function() takes 2 positional arguments but 3 were given" messages don't help remote clients understand the expected parameter format, leading to trial-and-error debugging.

How to fix: Validate parameter count before function invocation and return INVALID_PARAMS errors with clear descriptions of the expected signature. Consider storing parameter metadata in the registry for better

error messages.

⚠️ **Pitfall: Connection Resource Leaks** Failing to properly close client sockets when connections end can exhaust the server's file descriptor limit, especially during testing with many short-lived connections.

Why it's wrong: Resource leaks accumulate over time and can crash the server or prevent new connections. The operating system has limits on open file descriptors per process.

How to fix: Use try-finally blocks or context managers to ensure socket cleanup. Implement proper connection termination handling that closes resources even when exceptions occur during request processing.

Implementation Guidance

Technology Recommendations:

Component	Simple Option	Advanced Option
Threading	`threading.Thread` per connection	`concurrent.futures.ThreadPoolExecutor`
Socket Handling	Raw `socket.socket` with manual framing	`socketserver.ThreadingTCPServer`
Method Registry	Simple `dict` mapping	Class-based registry with validation
Error Logging	Built-in `logging` module	Structured logging with JSON output

Recommended File Structure:

```
rpc_framework/
  server/
    __init__.py             ← exports RPCServer
    registry.py             ← method registration logic
    server.py               ← main server implementation
    connection.py           ← connection handling logic
  protocol/
    messages.py             ← from previous milestone
    serialization.py        ← from previous milestone
  common/
    errors.py               ← RPC exception classes
    socket_helper.py        ← socket utilities
```

Socket Helper Infrastructure (Complete):

PYTHON

```python
import socket
import struct
import json
import time
from typing import Dict, Any, Optional

class SocketHelper:
    """Utility class for socket operations with message framing."""

    @staticmethod
    def send_all(sock: socket.socket, data: bytes, timeout: float = 30.0) -> None:
        """Send all bytes, handling partial sends and timeouts."""
        sock.settimeout(timeout)
        total_sent = 0
        while total_sent < len(data):
            try:
                sent = sock.send(data[total_sent:])
                if sent == 0:
                    raise ConnectionError("Socket connection broken")
                total_sent += sent
            except socket.timeout:
                raise RPCTimeoutError(f"Send timeout after {timeout}s")

    @staticmethod
    def recv_all(sock: socket.socket, size: int, timeout: float = 30.0) -> bytes:
        """Receive exact number of bytes."""
        sock.settimeout(timeout)
```

```
        chunks = []
        bytes_received = 0
        while bytes_received < size:
            try:
                chunk = sock.recv(size - bytes_received)
                if not chunk:
                    raise ConnectionError("Socket connection broken")
                chunks.append(chunk)
                bytes_received += len(chunk)
            except socket.timeout:
                raise RPCTimeoutError(f"Receive timeout after {timeout}s")
        return b''.join(chunks)

    @staticmethod
    def recv_message(sock: socket.socket, timeout: float = 30.0) -> bytes:
        """Receive a length-prefixed message."""
        # First receive 4-byte length prefix
        length_bytes = SocketHelper.recv_all(sock, 4, timeout)
        message_length = struct.unpack('>I', length_bytes)[0]

        # Then receive the actual message
        return SocketHelper.recv_all(sock, message_length, timeout)

    @staticmethod
    def send_message(sock: socket.socket, message_bytes: bytes, timeout: float = 30.0) ->
None:
        """Send a length-prefixed message."""
```

```
        length_prefix = struct.pack('>I', len(message_bytes))

        SocketHelper.send_all(sock, length_prefix + message_bytes, timeout)
```

Method Registry Core (Complete):

PYTHON

```python
import threading
from typing import Dict, Callable, Any, Optional

class MethodRegistry:
    """Thread-safe registry for RPC methods."""

    def __init__(self):
        self._methods: Dict[str, Callable] = {}
        self._lock = threading.RLock()
        self._descriptions: Dict[str, str] = {}

    def register_method(self, name: str, method: Callable, description: str = "") -> None:
        """Register a callable method by name."""
        with self._lock:
            self._methods[name] = method
            if description:
                self._descriptions[name] = description

    def lookup_method(self, name: str) -> Optional[Callable]:
        """Look up a method by name. Thread-safe read operation."""
        return self._methods.get(name)

    def list_methods(self) -> Dict[str, str]:
        """Return a copy of all registered methods with descriptions."""
        with self._lock:
            return {name: self._descriptions.get(name, "No description")
                    for name in self._methods}
```

```
    def method_exists(self, name: str) -> bool:
        """Check if a method is registered."""
        return name in self._methods
```

Server Core Logic Skeleton:

PYTHON

```python
import socket

import threading

import json

import logging

from typing import Dict, Any


class RPCServer:

    """Simple RPC Server with method registry and connection handling."""



    def __init__(self, host: str = 'localhost', port: int = 8000):

        self.host = host

        self.port = port

        self.registry = MethodRegistry()

        self.server_socket = None

        self.running = False

        self.logger = logging.getLogger(__name__)



    def register_method(self, name: str, method: Callable) -> None:

        """Register a method in the server's registry."""

        # TODO 1: Use the registry to register the method by name

        # TODO 2: Log the registration for debugging

        pass



    def start_server(self) -> None:

        """Start the RPC server and listen for connections."""

        # TODO 1: Create server socket and bind to host:port

        # TODO 2: Start listening for connections (backlog=5)
```

```
        # TODO 3: Set self.running = True
        # TODO 4: Enter main accept loop, creating thread for each connection
        # TODO 5: Handle KeyboardInterrupt for graceful shutdown
        # Hint: Use threading.Thread(target=self.handle_connection, args=(client_sock,))
        pass

    def handle_connection(self, client_sock: socket.socket) -> None:
        """Handle a single client connection."""
        try:
            # TODO 1: Log the new connection
            # TODO 2: Loop receiving messages until connection closes
            # TODO 3: For each message, call process_request and send response
            # TODO 4: Handle SocketHelper exceptions and close connection gracefully
            # Hint: Use SocketHelper.recv_message() and SocketHelper.send_message()
            pass
        finally:
            # TODO 5: Always close the client socket
            pass

    def process_request(self, request_bytes: bytes) -> bytes:
        """Process a single RPC request and return response bytes."""
        try:
            # TODO 1: Deserialize the request bytes to a dictionary
            # TODO 2: Validate request has required fields (method, params, id)
            # TODO 3: Look up the method in the registry
            # TODO 4: If method not found, create METHOD_NOT_FOUND error response
            # TODO 5: If found, call the method with params and create success response
```

```
            # TODO 6: Handle any exceptions during method execution
            # TODO 7: Serialize response dictionary to bytes and return
            # Hint: Use deserialize_message() and serialize_message() from protocol
            # Hint: Use create_response_message() and create_error_message()
            pass
        except json.JSONDecodeError:
            # TODO 8: Handle JSON parsing errors with PARSE_ERROR response
            pass
        except Exception as e:
            # TODO 9: Handle unexpected errors with INTERNAL_ERROR response
            pass

    def stop_server(self) -> None:
        """Stop the server gracefully."""
        # TODO 1: Set self.running = False
        # TODO 2: Close server socket if open
        pass
```

Milestone Checkpoint:

After implementing the server component, verify the following behavior:

1. **Method Registration Test**: Create a simple function like `def add(a, b): return a + b`, register it with the server, and verify it appears in the registry.
2. **Connection Handling Test**: Start the server and use telnet or a simple socket client to connect. The server should accept the connection without crashing.
3. **Request Processing Test**: Send a properly formatted JSON-RPC request and verify you receive a JSON-RPC response. Use this test message:

```json
{"method": "add", "params": [2, 3], "id": "test-1"}
```

4. **Error Handling Test**: Send an invalid request (malformed JSON or unknown method) and verify you receive an appropriate error response.
5. **Concurrent Connection Test**: Open multiple connections simultaneously and verify the server handles them independently.

Language-Specific Hints:

- Use `socket.socket(socket.AF_INET, socket.SOCK_STREAM)` for TCP sockets
- Set `socket.SO_REUSEADDR` to avoid "Address already in use" errors during testing
- Use `threading.Thread(target=handler, args=(sock,), daemon=True)` for connection threads
- Python's GIL limits true parallelism but allows I/O concurrency for socket operations
- Use `logging.basicConfig(level=logging.INFO)` to see connection and request logs
- Handle `ConnectionResetError` and `BrokenPipeError` for client disconnections

Client Component Design

> **Milestone(s):** Milestone 3 (Client Implementation) - builds the RPC client that provides method proxying, manages network communication, and handles timeouts

The client component serves as the user-facing interface to the RPC framework, transforming local method calls into remote procedure calls and managing all the underlying network complexity. The client must provide a seamless experience where calling a remote function feels identical to calling a local function, while handling the inherent challenges of network communication, serialization, and error propagation.

Mental Model: The Secretary Pattern

Think of the RPC client as your personal secretary who handles all your external communications. When you need to contact someone in another department, you don't dial the phone yourself, manage the conversation, or worry about busy signals. Instead, you simply tell your secretary "I need to call the accounting department and ask them to calculate the tax for invoice #1234." Your secretary handles everything: looking up the phone number, making the call, translating your request into the proper format, waiting for the response, handling any errors ("sorry, their line is busy, should I try again?"), and finally delivering the result back to you in a format you can immediately use.

The RPC client works similarly. When your application code wants to call `remote.calculate_tax(invoice_id="1234")`, the client proxy intercepts this call, translates it into a properly formatted RPC request message, manages the TCP connection to the server, sends the request, waits for the response, handles any network errors or timeouts, and finally returns the result as if it were a local function call. Just like a good secretary, the client handles all the complexity behind the scenes so you can focus on your actual work.

This mental model helps us understand the client's core responsibilities: **transparent proxying** (making remote calls look local), **connection management** (maintaining reliable communication channels), **request tracking** (ensuring responses match requests), and **error translation** (converting network failures into meaningful exceptions).

Method Proxying

Method proxying is the mechanism that creates the illusion of local function calls for remote procedures. The proxy object intercepts method calls using language-specific metaprogramming features and converts them into RPC requests. This requires careful handling of method names, parameter serialization, and response deserialization.

The proxy object acts as a stand-in for the remote service, implementing the same interface but routing calls over the network instead of executing them locally. When a method is called on the proxy, it captures the method name and arguments, creates an RPC request message using the protocol defined in Milestone 1, sends it to the server, waits for the response, and returns the result or raises an appropriate exception.

The key challenge in method proxying is maintaining the natural feel of local method calls while handling the additional complexity of network communication. This includes preserving parameter types through serialization, maintaining call semantics (synchronous by default), and translating remote errors into local exceptions.

> **Design Insight**: The proxy pattern is crucial for RPC adoption because it eliminates the mental overhead of thinking about network details. Developers can write `result = calculator.add(5, 3)` instead of `result = rpc_client.call("add", [5, 3])`, making remote calls feel natural and reducing cognitive load.

Proxy Object Structure:

Component	Type	Description
`client`	`RPCClient`	Reference to the underlying RPC client that manages connections
`timeout`	`float`	Default timeout for method calls in seconds
`_method_cache`	`Dict[str, Callable]`	Cache of dynamically created method proxies for performance

Method Proxy Interface:

Method	Parameters	Returns	Description
`__getattr__`	`name: str`	`Callable`	Dynamically creates proxy methods for any attribute access
`__call__`	`*args, **kwargs`	`Any`	Executes the actual RPC call when proxy method is invoked
`_create_method_proxy`	`method_name: str`	`Callable`	Creates a callable that captures method name and forwards to RPC client
`_validate_parameters`	`args: Tuple, kwargs: Dict`	`None`	Ensures parameters are JSON-serializable before sending

The method proxying implementation uses Python's `__getattr__` magic method to intercept attribute access and return callable objects that perform RPC calls. When code calls `proxy.method_name(arg1, arg2)`, Python first calls `proxy.__getattr__("method_name")` which returns a callable, then immediately calls that callable with the provided arguments.

Method Proxying Algorithm:

1. Application code calls a method on the proxy object (e.g., `proxy.calculate_tax(invoice_id="1234")`)
2. Python's attribute resolution triggers `__getattr__("calculate_tax")`
3. The proxy checks its method cache for an existing proxy function for this method name
4. If not cached, it creates a new proxy function that captures the method name and stores it in the cache
5. The proxy function is returned to Python, which immediately calls it with the provided arguments
6. Inside the proxy function, parameters are validated for JSON serializability
7. A unique request ID is generated using `generate_request_id()`
8. The request message is created using `create_request_message(method_name, args, request_id)`
9. The request is sent to the server via the underlying `RPCClient`
10. The proxy function blocks waiting for the response with the matching request ID
11. When the response arrives, it's either returned as the method result or converted to an exception
12. The final result is returned to the application code as if it were a local function call

> **Critical Design Decision**: We choose to make method calls synchronous (blocking) by default because this matches the semantics of local function calls and is easier for beginners to understand. Asynchronous support can be added later as an advanced feature.

Connection Management

Connection management handles the TCP socket lifecycle, including establishing connections, reusing connections for multiple requests, handling connection failures, and properly closing connections when done. Effective connection management is crucial for both performance and reliability.

The client must decide whether to create a new connection for each RPC call or reuse connections across multiple calls. Connection reuse improves performance by avoiding the overhead of TCP handshakes, but introduces complexity around connection state management, error recovery, and concurrent access.

For this educational framework, we choose a simple connection-per-client model where each `RPCClient` instance maintains a single TCP connection that is established on first use and reused for all subsequent calls from that client instance. This strikes a balance between simplicity and performance while avoiding the complexity of connection pooling.

> **Decision: Connection Reuse Strategy**
>
> - **Context**: Each RPC call could create a new TCP connection or reuse an existing connection, with trade-offs between simplicity and performance
> - **Options Considered**:
>   1. New connection per call (simple but slow)
>   2. Single persistent connection per client (balanced)
>   3. Connection pool with multiple connections (complex but scalable)
> - **Decision**: Single persistent connection per client instance
> - **Rationale**: Provides good performance for typical usage patterns while keeping implementation simple. Most applications create one client instance and make many calls, so connection reuse provides significant benefits without the complexity of pool management.
> - **Consequences**: Better performance than connection-per-call, simpler than connection pooling, but limits concurrency to one outstanding request per client instance.

Connection Management States:

State	Description	Valid Transitions	Trigger Events
`Disconnected`	No active connection, client is ready to connect	`Connecting`	First RPC call made
`Connecting`	TCP connection establishment in progress	`Connected`, `Failed`	Socket connect completes or times out
`Connected`	Active connection ready for requests	`Sending`, `Disconnected`	RPC call made or connection error detected
`Sending`	Request being sent over connection	`Waiting`, `Failed`	Send completes or socket error occurs
`Waiting`	Waiting for response from server	`Receiving`, `Timeout`, `Failed`	Response arrives, timeout expires, or connection error
`Receiving`	Reading response from server	`Connected`, `Failed`	Response fully received or read error
`Failed`	Connection failed and needs to be reset	`Disconnected`	Error handled and connection cleaned up
`Timeout`	Request timed out waiting for response	`Disconnected`	Timeout handled and connection closed

Connection Management Interface:

Method	Parameters	Returns	Description
`_ensure_connected`	`timeout: float`	`None`	Establishes connection if not already connected, raises `RPCTransportError` on failure
`_disconnect`	`None`	`None`	Closes the connection and resets state to disconnected
`_is_connected`	`None`	`bool`	Checks if the connection is active and healthy
`_handle_connection_error`	`error: Exception`	`None`	Handles connection failures by cleaning up and transitioning to failed state
`_reset_connection`	`None`	`None`	Forces connection reset for recovery from errors

Connection Establishment Algorithm:

1. Client receives an RPC call and checks if connection exists and is healthy

2. If no connection exists, create a new TCP socket using the configured host and port
3. Set socket options for timeout and TCP keepalive to detect dead connections
4. Attempt to connect to the server with the specified connection timeout
5. If connection fails, raise `RPCTransportError` with details about the failure
6. If connection succeeds, store the socket and transition to connected state
7. For subsequent calls, reuse the existing connection if it's still healthy
8. If the connection becomes unhealthy (detected during send/receive), close it and establish a new one

The connection health check is performed before each RPC call by attempting to send/receive with a very short timeout. If this fails, the connection is considered dead and must be re-established. This handles cases where the server has closed the connection or network connectivity has been lost.

> **Implementation Note**: Connection establishment should use a configurable timeout (default 5 seconds) to avoid hanging indefinitely when the server is unreachable. The timeout should be shorter than typical RPC call timeouts to provide fast feedback.

Timeout and ID Tracking

Timeout and ID tracking ensures that RPC calls don't hang indefinitely and that responses are correctly matched to their originating requests. This is particularly important in network programming where messages can be delayed, reordered, or lost entirely.

Each RPC request is assigned a unique request ID that allows the client to correlate responses with their originating requests. The client maintains a mapping of outstanding request IDs to their associated metadata (start time, timeout deadline, response callback) and actively monitors for timeouts.

The timeout mechanism prevents clients from blocking forever when servers are unresponsive or when network connectivity is lost. When a timeout occurs, the client should clean up the pending request, close the connection (since the server may still send a late response), and raise a `RPCTimeoutError` to the calling code.

> **Design Challenge**: The biggest challenge in timeout handling is deciding what to do with late responses. If a request times out but the server eventually sends a response, should we ignore it, log it, or handle it somehow? For simplicity, we close the connection on timeout to avoid out-of-order message handling.

Request Tracking Structure:

Field	Type	Description
`request_id`	`str`	Unique identifier for this request, generated by `generate_request_id()`
`method_name`	`str`	Name of the remote method being called
`start_time`	`float`	Timestamp when request was sent (from `time.time()`)
`timeout_deadline`	`float`	Absolute time when request should timeout (`start_time + timeout`)
`response_received`	`bool`	Flag indicating whether response has been received
`result`	`Any`	Response result or error, set when response arrives
`condition`	`threading.Condition`	Synchronization primitive for blocking until response or timeout

Timeout Management Interface:

Method	Parameters	Returns	Description
`_track_request`	`request_id: str, timeout: float`	`None`	Registers new request for timeout monitoring
`_wait_for_response`	`request_id: str`	`Dict`	Blocks until response received or timeout occurs
`_handle_response`	`response: Dict`	`None`	Processes incoming response and notifies waiting threads
`_cleanup_request`	`request_id: str`	`None`	Removes request from tracking and releases resources
`_check_timeout`	`request_id: str`	`bool`	Checks if request has exceeded its timeout deadline

Request ID Generation Strategy:

Request IDs must be unique within the lifetime of a client instance to avoid response mismatching. A simple approach is to combine a timestamp with a counter, or use a UUID. The ID should be short enough to minimize message overhead but unique enough to avoid collisions.

```
Request ID Format: f"{int(time.time() * 1000000)}-{counter}"
Example: "1640995200123456-1", "1640995200123457-2"
```

This format provides microsecond timestamp precision combined with a sequential counter, making IDs unique and sortable while remaining human-readable for debugging.

Timeout Handling Algorithm:

1. When an RPC call begins, generate a unique request ID and calculate the timeout deadline
2. Register the request in the tracking dictionary with its metadata and synchronization objects
3. Send the request message to the server over the TCP connection
4. Start blocking wait on the condition variable with the calculated timeout duration
5. If a response arrives before timeout, `_handle_response` notifies the condition variable and the wait returns
6. If the timeout expires first, clean up the request, close the connection, and raise `RPCTimeoutError`
7. Background response handling checks incoming message IDs against tracked requests
8. When a matching response is found, store the result and notify the waiting thread
9. If an unmatched response arrives (expired request), log it and discard it

The timeout calculation should account for both network latency and server processing time. A reasonable default is 30 seconds for most applications, with the ability to customize per-call or per-client.

> **Concurrency Consideration**: If we later add support for concurrent requests from a single client, the request tracking system is already designed to handle this with per-request condition variables and thread-safe dictionaries.

Architecture Decision Records

> **Decision: Synchronous vs Asynchronous Client API**
>
> - **Context**: RPC calls could be synchronous (blocking until response) or asynchronous (return immediately with future/callback), affecting both API design and implementation complexity
> - **Options Considered**:
>   1. Synchronous only (simple, matches local call semantics)
>   2. Asynchronous only (complex, requires event loop or callback handling)
>   3. Both synchronous and asynchronous APIs (flexible but doubles implementation complexity)
> - **Decision**: Synchronous only for this educational framework
> - **Rationale**: Synchronous calls match the semantics of local function calls, making the RPC abstraction more natural. Implementation is significantly simpler without callback management or event loops. Most RPC use cases can tolerate blocking calls, and async can be added later.
> - **Consequences**: Simple implementation and natural API, but limits throughput for I/O-bound applications that could benefit from async concurrency.

Option	Pros	Cons
Synchronous Only	Simple implementation, natural API, matches local calls	Blocks threads, limited concurrency
Asynchronous Only	High concurrency, non-blocking	Complex implementation, unnatural API for beginners
Both APIs	Flexible, covers all use cases	Double implementation complexity, API confusion

Decision: Connection Pooling vs Single Connection

- **Context**: Clients could use a pool of connections for higher concurrency or a single connection per client for simplicity
- **Options Considered**:
  1. Single persistent connection per client (simple, adequate for most use cases)
  2. Connection pool with multiple connections (complex, supports high concurrency)
  3. New connection per request (very simple but poor performance)
- **Decision**: Single persistent connection per client instance
- **Rationale**: Most applications create one client instance and use it throughout the program lifetime. A single connection provides good performance through reuse while avoiding the complexity of pool management, sizing, and cleanup.
- **Consequences**: Good performance for typical usage, simple implementation, but limits each client instance to one outstanding request at a time.

Option	Pros	Cons
Single Connection	Simple, good performance, easy connection management	No concurrency within one client
Connection Pool	High concurrency, optimal resource usage	Complex implementation, pool sizing decisions
Per-Request Connection	Very simple, no state management	Poor performance, high overhead

Decision: Error Handling Strategy

- **Context**: Network and RPC errors need to be presented to application code in a way that's both informative and actionable
- **Options Considered**:
  1. Generic exceptions with error codes (simple but less type-safe)
  2. Specific exception types for each error category (more complex but better handling)
  3. Error return values instead of exceptions (doesn't match Python conventions)
- **Decision**: Specific exception hierarchy with `RPCError` base class
- **Rationale**: Python developers expect exceptions for error conditions. Specific exception types allow application code to handle different error categories appropriately (retry network errors, don't retry method not found errors).
- **Consequences**: More exception classes to implement and document, but better error handling capabilities and more Pythonic API design.

Option	Pros	Cons
Specific Exception Types	Type-safe, targeted handling, clear semantics	More classes to implement
Generic Exceptions	Simple implementation, fewer classes	Less precise error handling
Error Return Values	Explicit error checking	Un-Pythonic, easy to ignore errors

Decision: Request ID Format

- **Context**: Request IDs must be unique within client lifetime and should be efficient to generate and compare
- **Options Considered**:
  1. Sequential integers (simple but not globally unique)
  2. UUIDs (globally unique but longer and less human-readable)
  3. Timestamp + counter combination (unique, sortable, readable)
- **Decision**: Timestamp + counter combination format
- **Rationale**: Provides uniqueness within client lifetime (sufficient for our use case), remains human-readable for debugging, and creates sortable IDs that can help with troubleshooting message ordering issues.
- **Consequences**: Slightly more complex ID generation than simple counters, but much more debuggable than UUIDs and adequate uniqueness for single-client scenarios.

Common Pitfalls

⚠️ Pitfall: Forgetting to Handle Connection Failures During RPC Calls

Many beginners assume that once a connection is established, it will remain healthy for the duration of the program. However, network connections can fail at any time due to server restarts, network partitions, or idle timeouts. If the client doesn't detect and handle connection failures, RPC calls will hang or fail with cryptic socket errors.

Why this is wrong: Socket operations (send/receive) can fail even after successful connection establishment. The server might close the connection, the network might drop packets, or intermediate firewalls might reset connections. Without proper error detection, clients become unusable after the first connection failure.

How to fix: Always wrap socket operations in try/catch blocks and handle connection errors by closing the socket, transitioning to disconnected state, and optionally retrying the connection. Implement connection health checks before each RPC call to detect dead connections early.

⚠️ Pitfall: Not Cleaning Up Timed-Out Requests

When a request times out, beginners often simply raise a timeout exception but leave the request tracking data structures intact. This causes memory leaks as the request dictionary grows, and can lead to confusion if late responses eventually arrive.

Why this is wrong: Timed-out requests consume memory indefinitely, and late responses might be delivered to the wrong caller if request IDs are reused. Additionally, the connection state becomes uncertain since the server might still send a response.

How to fix: Always clean up request tracking data when a timeout occurs, and close the connection to prevent out-of-order responses. Use `_cleanup_request()` to remove the request from tracking dictionaries and release condition variables.

⚠️ Pitfall: Making Method Proxies That Don't Preserve Call Semantics

When implementing method proxying with `__getattr__`, beginners sometimes return the actual RPC result from `__getattr__` instead of returning a callable. This breaks the natural method call syntax and makes the API confusing.

Why this is wrong: `proxy.method_name` should return a callable that can be invoked with arguments, not execute the RPC call immediately. The call `proxy.method_name(args)` requires two steps: first get the callable, then invoke it with arguments.

How to fix: `__getattr__` should return a callable (lambda or bound method) that captures the method name and performs the RPC call when invoked. Don't execute the RPC call from within `__getattr__`.

⚠️ Pitfall: Ignoring Parameter Serialization Validation

JSON serialization has limitations on supported data types. Beginners often pass objects that can't be serialized (like file handles, custom classes, or functions) and get confusing errors during the RPC call rather

than at parameter validation time.

Why this is wrong: Serialization errors happen deep in the call stack and provide poor error messages. It's better to validate parameters early and provide clear feedback about what types are supported.

How to fix: Implement `_validate_parameters()` that checks all arguments and keyword arguments for JSON serializability before creating the RPC request. Provide clear error messages explaining what types are supported.

⚠️ Pitfall: Race Conditions in Response Handling

When implementing request tracking with threading, beginners sometimes access the tracking dictionary without proper synchronization, leading to race conditions where responses are lost or delivered to the wrong caller.

Why this is wrong: Multiple threads accessing shared data structures without synchronization can cause data corruption, lost responses, or responses delivered to the wrong waiting thread.

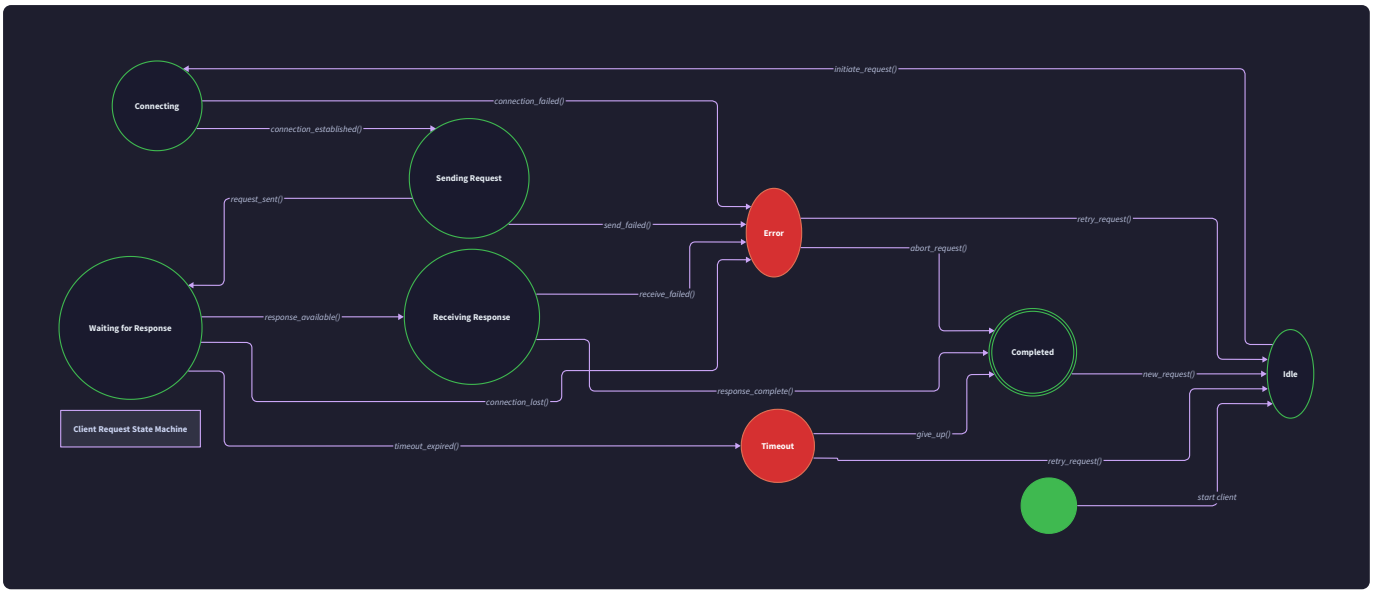
How to fix: Use threading primitives (locks, condition variables) consistently around all access to shared data structures. Each request should have its own condition variable for waiting, and the tracking dictionary should be protected with a lock.

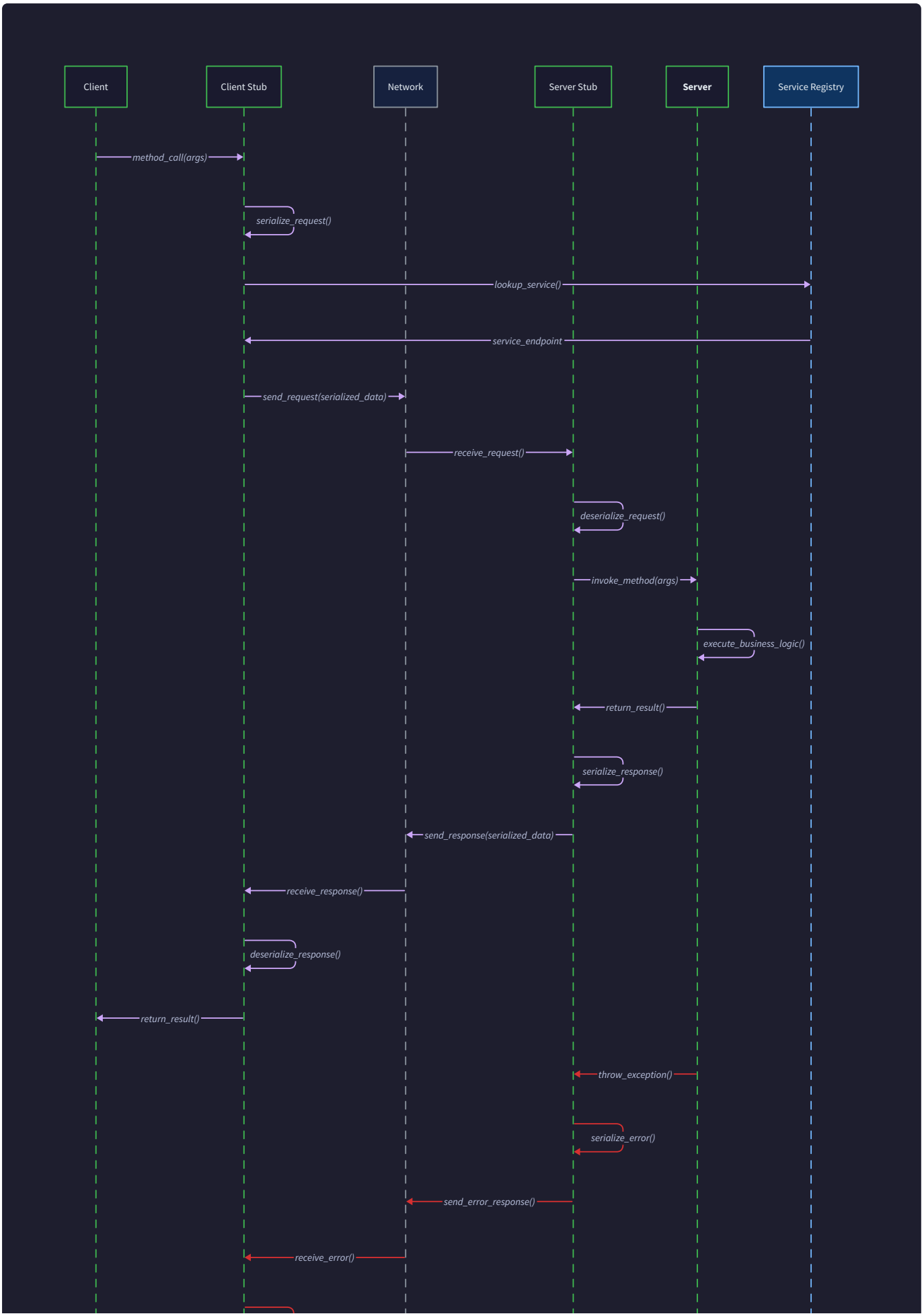
⚠️ Pitfall: Not Handling Partial Socket Reads/Writes

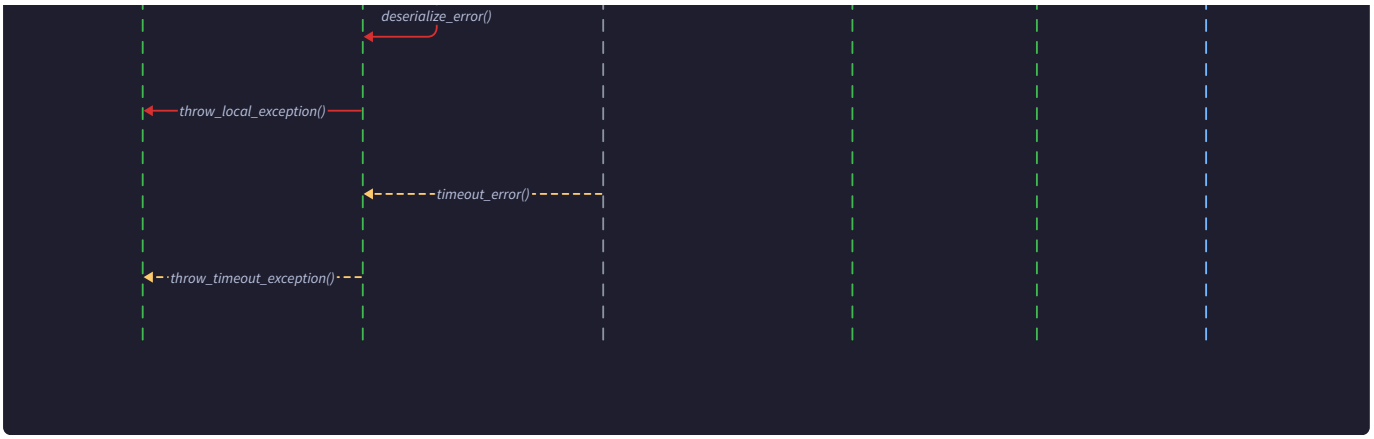
TCP sockets don't guarantee that `send()` will send all bytes or that `recv()` will receive the complete message in one call. Beginners often assume that one `recv()` call will return the complete response message, leading to protocol errors when messages are split across multiple TCP packets.

Why this is wrong: TCP is a stream protocol, not a message protocol. Large messages will be split across multiple packets, and small messages might be combined into one packet. Without proper message framing and complete read/write handling, messages become corrupted.

How to fix: Use the length-prefixed message framing from the protocol design, and implement `send_all()` and `recv_all()` utility functions that loop until all bytes are sent/received. Always read the length prefix first, then read exactly that many bytes for the message body.


Connecting
Sending Request
Waiting for Response
Receiving Response
Error
Completed
Idle
Timeout
Client Request State Machine
initiate_request()
connection_failed()
connection_established()
request_sent()
send_failed()
retry_request()
abort_request()
receive_failed()
response_available()
new_request()
response_complete()
connection_lost()
give_up()
timeout_expired()
retry_request()
start client


Client
Client Stub
Network
Server Stub
Server
Service Registry
method_call(args)
serialize_request()
lookup_service()
service_endpoint
send_request(serialized_data)
receive_request()
deserialize_request()
invoke_method(args)
execute_business_logic()
return_result()
serialize_response()
send_response(serialized_data)
receive_response()
deserialize_response()
return_result()
throw_exception()
serialize_error()
send_error_response()
receive_error()



Implementation Guidance

This subsection provides concrete implementation guidance for building the RPC client component in Python. The client is responsible for method proxying, connection management, and request/response handling.

Technology Recommendations:

Component	Simple Option	Advanced Option
Network Transport	Raw TCP sockets with JSON	HTTP with requests library
Serialization	Built-in `json` module	`msgpack` for binary efficiency
Concurrency	Threading with condition variables	`asyncio` for async support
Connection Management	Single persistent connection	Connection pooling with queue
Timeout Handling	Socket timeouts with threading	`select()` or `epoll()` for non-blocking

Recommended File Structure:

```python
# rpc_framework/
#   client/
#     __init__.py          ← exports RPCClient and proxy classes
#     client.py            ← main RPCClient implementation
#     proxy.py             ← method proxy implementation
#     connection.py        ← connection management utilities
#     exceptions.py        ← client-specific exceptions
#   common/
#     protocol.py          ← shared protocol utilities (from Milestone 1)
#     socket_helper.py     ← socket utilities (from server implementation)
```

Infrastructure Starter Code (Complete Socket Helper):

PYTHON

```python
# rpc_framework/common/socket_helper.py

import socket

import struct

import time

from typing import Optional


class SocketHelper:

    """Utility class for reliable socket operations with timeout support."""



    @staticmethod

    def send_all(sock: socket.socket, data: bytes, timeout: float = 30.0) -> None:

        """Send all bytes, handling partial sends and timeouts."""

        sock.settimeout(timeout)

        total_sent = 0

        while total_sent < len(data):

            try:

                sent = sock.send(data[total_sent:])

                if sent == 0:

                    raise RPCTransportError("Socket connection broken during send")

                total_sent += sent

            except socket.timeout:

                raise RPCTimeoutError(f"Send timeout after {timeout} seconds")

            except socket.error as e:

                raise RPCTransportError(f"Socket send error: {e}")



    @staticmethod

    def recv_all(sock: socket.socket, size: int, timeout: float = 30.0) -> bytes:

        """Receive exactly size bytes, handling partial receives and timeouts."""
```

```
        sock.settimeout(timeout)
        chunks = []
        bytes_received = 0

        while bytes_received < size:
            try:
                chunk = sock.recv(min(size - bytes_received, 4096))
                if not chunk:
                    raise RPCTransportError("Socket connection closed during receive")
                chunks.append(chunk)
                bytes_received += len(chunk)
            except socket.timeout:
                raise RPCTimeoutError(f"Receive timeout after {timeout} seconds")
            except socket.error as e:
                raise RPCTransportError(f"Socket receive error: {e}")

        return b''.join(chunks)

    @staticmethod
    def recv_message(sock: socket.socket, timeout: float = 30.0) -> bytes:
        """Receive length-prefixed message using protocol framing."""
        # First, receive 4-byte length prefix
        length_bytes = SocketHelper.recv_all(sock, 4, timeout)
        message_length = struct.unpack('!I', length_bytes)[0]

        # Then receive the message body
        if message_length > 1024 * 1024:  # 1MB limit
```

```
            raise RPCProtocolError(f"Message too large: {message_length} bytes")


        return SocketHelper.recv_all(sock, message_length, timeout)


    @staticmethod
    def send_message(sock: socket.socket, message_bytes: bytes, timeout: float = 30.0) ->
None:
        """Send length-prefixed message using protocol framing."""
        if len(message_bytes) > 1024 * 1024:  # 1MB limit
            raise RPCProtocolError(f"Message too large: {len(message_bytes)} bytes")


        # Send 4-byte length prefix followed by message
        length_prefix = struct.pack('!I', len(message_bytes))
        SocketHelper.send_all(sock, length_prefix + message_bytes, timeout)
```

Core Logic Skeleton (Client Implementation):

PYTHON

```python
# rpc_framework/client/client.py

import socket

import threading

import time

import json

from typing import Dict, Any, Optional

from ..common.protocol import generate_request_id, create_request_message,
deserialize_message

from ..common.socket_helper import SocketHelper

from .exceptions import RPCError, RPCTransportError, RPCTimeoutError


class RPCClient:

    """RPC client with connection management and request tracking."""



    def __init__(self, host: str = 'localhost', port: int = 8000, default_timeout: float =
30.0):

        self.host = host

        self.port = port

        self.default_timeout = default_timeout



        # Connection management

        self._socket: Optional[socket.socket] = None

        self._connected = False

        self._connection_lock = threading.Lock()



        # Request tracking

        self._pending_requests: Dict[str, Dict] = {}

        self._request_lock = threading.Lock()
```

```
        # Response handling
        self._response_thread: Optional[threading.Thread] = None
        self._shutdown_event = threading.Event()

    def call(self, method_name: str, *args, timeout: Optional[float] = None, **kwargs) ->
Any:
        """Make an RPC call and return the result."""
        if timeout is None:
            timeout = self.default_timeout

        # TODO 1: Generate unique request ID using generate_request_id()
        # TODO 2: Validate that all parameters are JSON-serializable
        # TODO 3: Create request message using create_request_message()
        # TODO 4: Ensure connection is established using _ensure_connected()
        # TODO 5: Register request for tracking using _track_request()
        # TODO 6: Send request message using SocketHelper.send_message()
        # TODO 7: Wait for response using _wait_for_response()
        # TODO 8: Clean up request tracking using _cleanup_request()
        # TODO 9: Return result or raise appropriate exception based on response
        # Hint: Wrap steps 4-6 in try/catch to handle connection errors

    def _ensure_connected(self, timeout: float) -> None:
        """Establish connection if not already connected."""
        with self._connection_lock:
            if self._connected and self._socket:
                # TODO 1: Check if existing connection is still healthy
                # TODO 2: If healthy, return early without reconnecting
```

```
            return

        # TODO 3: Create new TCP socket with appropriate options
        # TODO 4: Set socket timeout for connection attempt
        # TODO 5: Connect to (self.host, self.port) with error handling
        # TODO 6: Start response handling thread using _start_response_thread()
        # TODO 7: Set self._connected = True and store socket reference
        # Hint: Use socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    def _track_request(self, request_id: str, timeout: float) -> None:
        """Register request for response tracking and timeout monitoring."""
        request_info = {
            'start_time': time.time(),
            'timeout_deadline': time.time() + timeout,
            'response_received': False,
            'result': None,
            'condition': threading.Condition()
        }

        with self._request_lock:
            # TODO 1: Add request_info to self._pending_requests[request_id]
            # TODO 2: Validate that request_id is not already in use
            pass

    def _wait_for_response(self, request_id: str) -> Dict:
        """Block until response received or timeout occurs."""
        with self._request_lock:
            if request_id not in self._pending_requests:
```

```
            raise RPCError(f"Request {request_id} not found in tracking")

        request_info = self._pending_requests[request_id]
        condition = request_info['condition']

    with condition:
        # TODO 1: Calculate remaining timeout based on deadline
        # TODO 2: Wait on condition variable with timeout
        # TODO 3: Check if response was received or timeout occurred
        # TODO 4: Return the response or raise RPCTimeoutError
        # Hint: Use condition.wait(timeout) and check return value

def _start_response_thread(self) -> None:
    """Start background thread to handle incoming responses."""
    if self._response_thread and self._response_thread.is_alive():
        return

    self._shutdown_event.clear()
    self._response_thread = threading.Thread(
        target=self._response_handler,
        daemon=True,
        name="RPC-ResponseHandler"
    )
    self._response_thread.start()

def _response_handler(self) -> None:
    """Background thread that receives and dispatches responses."""
    while not self._shutdown_event.is_set():
```

```
            try:
                # TODO 1: Receive message using SocketHelper.recv_message()
                # TODO 2: Deserialize message using deserialize_message()
                # TODO 3: Extract request_id from response
                # TODO 4: Find matching pending request using request_id
                # TODO 5: Store response in request_info and notify waiting thread
                # TODO 6: Handle case where no matching request found (log and discard)
                # Hint: Use short timeout on recv_message to allow shutdown checking
                pass
            except socket.timeout:
                # TODO 7: Continue loop to check shutdown event
                continue
            except Exception as e:
                # TODO 8: Handle connection errors by breaking loop and disconnecting
                break

    def _cleanup_request(self, request_id: str) -> None:
        """Remove request from tracking and release resources."""
        with self._request_lock:
            # TODO 1: Remove request_id from self._pending_requests if present
            # TODO 2: Handle case where request_id not found (already cleaned up)
            pass

    def _disconnect(self) -> None:
        """Close connection and clean up resources."""
        with self._connection_lock:
            # TODO 1: Set shutdown event to stop response thread
            # TODO 2: Close socket if it exists
```

```
            # TODO 3: Wait for response thread to finish
            # TODO 4: Reset connection state variables
            # TODO 5: Clean up any pending requests with connection errors
            pass

    def __enter__(self):
        """Context manager entry."""
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        """Context manager exit - clean up connection."""
        self._disconnect()
```

Method Proxy Implementation:

PYTHON

```python
# rpc_framework/client/proxy.py

import json

from typing import Any, Callable


class RPCProxy:

    """Proxy object that converts method calls to RPC requests."""


    def __init__(self, client, timeout: float = None):

        # Use object.__setattr__ to avoid infinite recursion with __getattr__

        object.__setattr__(self, '_client', client)

        object.__setattr__(self, '_timeout', timeout)

        object.__setattr__(self, '_method_cache', {})


    def __getattr__(self, method_name: str) -> Callable:

        """Create proxy method for any attribute access."""

        # TODO 1: Check method cache for existing proxy function

        # TODO 2: If not cached, create new proxy function using _create_method_proxy()

        # TODO 3: Store in cache and return the callable

        # Hint: Cache improves performance for repeated calls to same method


    def _create_method_proxy(self, method_name: str) -> Callable:

        """Create a callable that performs RPC call for the given method."""

        def proxy_method(*args, **kwargs):

            # TODO 1: Call self._client.call() with method_name and arguments

            # TODO 2: Use self._timeout if specified, otherwise client default

            # TODO 3: Return result from RPC call

            pass
```

```
        # Set helpful attributes for debugging
        proxy_method.__name__ = method_name
        proxy_method.__doc__ = f"RPC proxy for remote method '{method_name}'"
        return proxy_method

    def _validate_json_serializable(self, obj: Any) -> None:
        """Validate that object can be JSON serialized."""
        try:
            # TODO 1: Attempt to JSON serialize the object
            # TODO 2: Raise clear error message if serialization fails
            # TODO 3: Include information about supported types
            json.dumps(obj)
        except (TypeError, ValueError) as e:
            raise RPCError(f"Parameter not JSON serializable: {e}. "
                           f"Supported types: str, int, float, bool, list, dict, None")
```

Language-Specific Implementation Hints:

1. **Socket Management**: Use `socket.socket(socket.AF_INET, socket.SOCK_STREAM)` for TCP connections. Always set timeouts with `sock.settimeout()` to avoid infinite blocking.
2. **Threading Synchronization**: Use `threading.Condition` for request/response coordination and `threading.Lock` for protecting shared data structures. Condition variables are perfect for the "wait for specific response" pattern.
3. **JSON Validation**: Use `json.dumps()` in a try/catch block to validate serializability. The `json` module raises `TypeError` for unserializable objects.
4. **Context Managers**: Implement `__enter__` and `__exit__` methods so clients can be used with `with` statements for automatic cleanup.
5. **Magic Methods**: Use `object.__setattr__()` in `RPCProxy.__init__()` to avoid triggering `__getattr__` during initialization.

Milestone Checkpoint:

After implementing the client component, verify functionality with this test:

```python
# test_client.py

def test_basic_client():

    # Assuming server is running with a 'add' method

    with RPCClient('localhost', 8000) as client:

        proxy = RPCProxy(client)



        # Test basic method call

        result = proxy.add(5, 3)

        assert result == 8



        # Test timeout behavior

        try:

            proxy.slow_method(timeout=1.0)

            assert False, "Should have timed out"

        except RPCTimeoutError:

            pass  # Expected



        print("√ Client implementation working correctly")


if __name__ == "__main__":

    test_basic_client()
```

Expected behavior:

- Client connects to server automatically on first call
- Method calls return results as if they were local functions
- Timeouts raise `RPCTimeoutError` after specified duration
- Connection is reused for multiple calls
- Resources are cleaned up when client exits context manager

Common Implementation Issues:

Symptom	Likely Cause	How to Fix
"Connection refused"	Server not running or wrong port	Check server status and port configuration
Calls hang forever	No timeout set on socket operations	Always use `sock.settimeout()` before socket operations
"Broken pipe" errors	Server closed connection unexpectedly	Implement connection health checks and retry logic
Responses to wrong calls	Race condition in request tracking	Use proper locking around `_pending_requests` dictionary
Memory leaks	Requests not cleaned up after timeout	Always call `_cleanup_request()` in finally blocks

Interactions and Data Flow

> **Milestone(s):** Integration of Milestones 1, 2, and 3 - demonstrates how message protocol, server, and client work together to execute remote procedure calls

Mental Model: The Phone Call Choreography

Think of an RPC call like a carefully choreographed business phone call. The client (caller) dials the server (receiver), speaks a structured message in a shared language (protocol), waits for a response, and hangs up. Just like a phone call has predictable stages - dialing, waiting for pickup, speaking, listening, confirming understanding, hanging up - an RPC call follows a precise sequence that both sides understand.

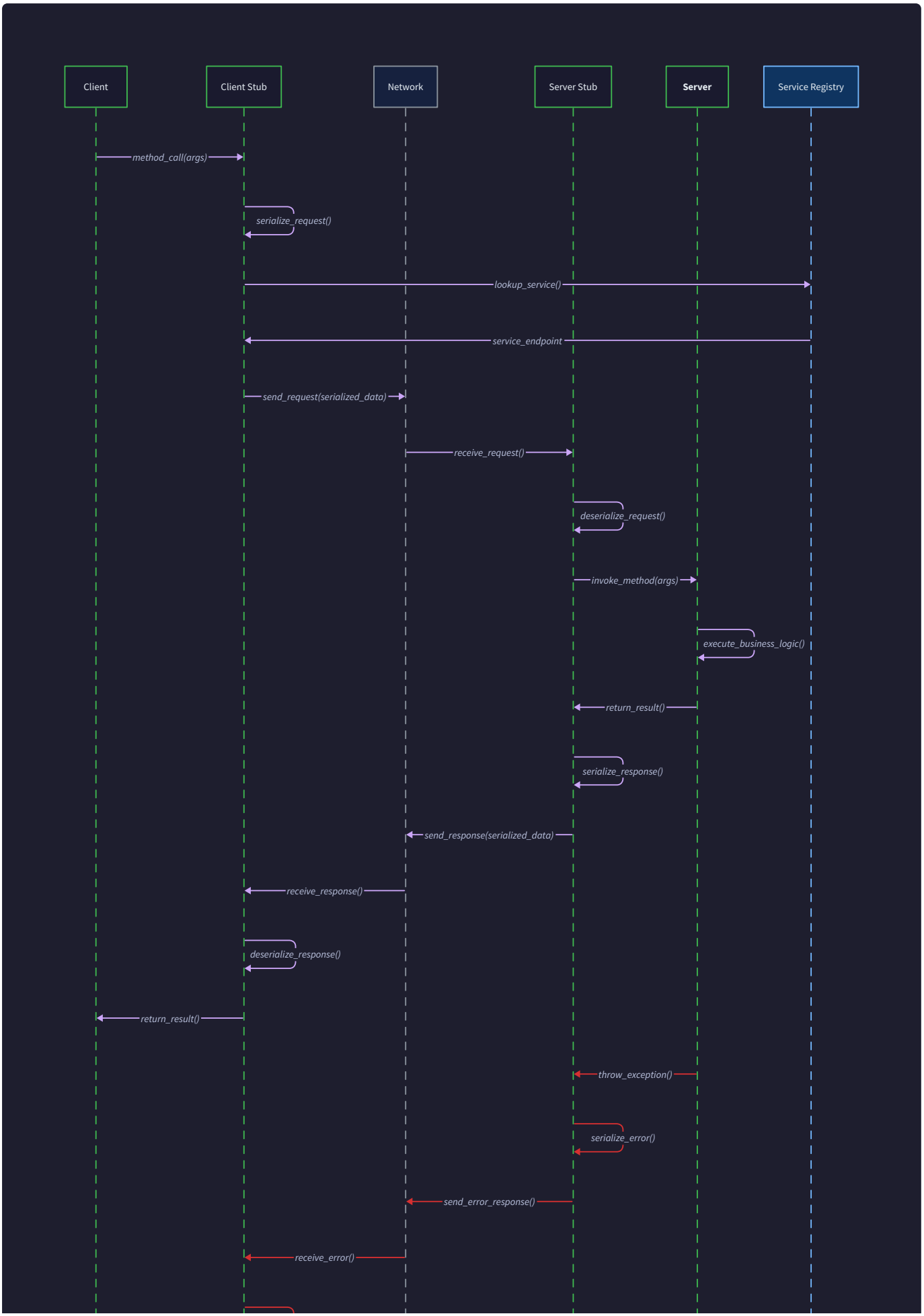
The magic happens because both parties follow the same "business phone etiquette" - they know when to speak, what format to use, how to handle misunderstandings, and how to gracefully end the conversation. When something goes wrong (busy signal, wrong number, bad connection), there are standard ways to handle each situation.

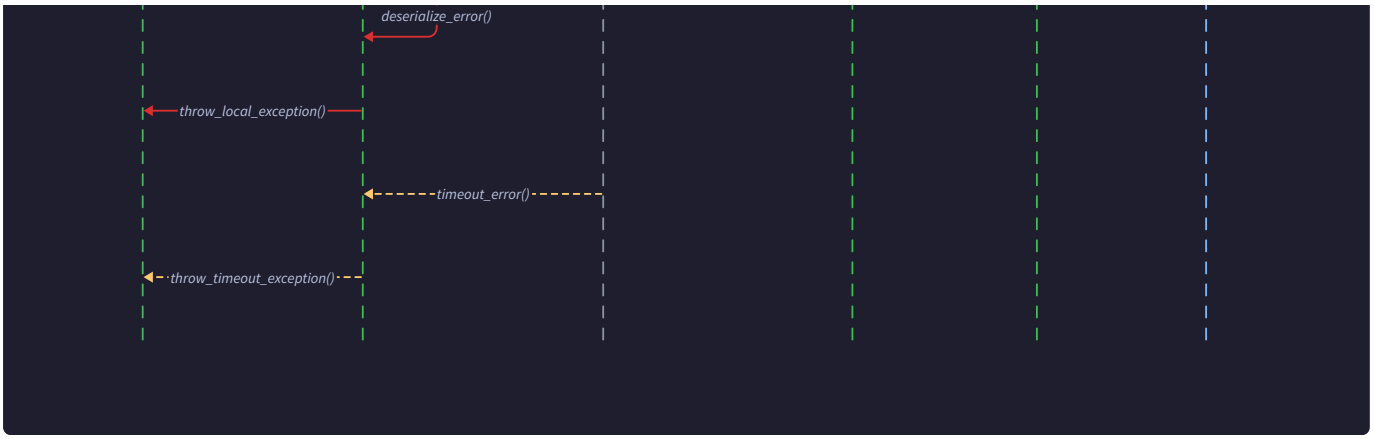
This choreography is crucial because unlike a local function call that happens in nanoseconds within the same process, an RPC call crosses network boundaries, involves serialization, and can fail in dozens of ways. The protocol and error handling ensure that despite this complexity, the client can still pretend it's just calling a local function.

RPC Call Sequence

The successful RPC call sequence represents the "happy path" where everything works as designed. This sequence involves precise coordination between the client proxy, network transport layer, and server

dispatcher to maintain the illusion that the remote function call is local.


Client
Client Stub
Network
Server Stub
Server
Service Registry
method_call(args)
serialize_request()
lookup_service()
service_endpoint
send_request(serialized_data)
receive_request()
deserialize_request()
invoke_method(args)
execute_business_logic()
return_result()
serialize_response()
send_response(serialized_data)
receive_response()
deserialize_response()
return_result()
throw_exception()
serialize_error()
send_error_response()
receive_error()



Step-by-Step Call Flow

The complete RPC call sequence involves eleven distinct steps that transform a local method invocation into a remote procedure execution and back again:

Step	Component	Action	Data Transformed	Purpose
1	Client Code	Invokes method on proxy object	`calc.add(5, 3)`	Triggers RPC call
2	RPCProxy	Intercepts call via `__getattr__`	Method name + args → internal call	Transparent proxying
3	RPCClient	Generates request ID and creates message	Args → JSON-RPC request dict	Message preparation
4	RPCClient	Serializes and sends message	Request dict → length-prefixed bytes	Network transmission
5	RPCServer	Receives and deserializes message	Bytes → request dict	Message reception
6	RPCServer	Looks up method in registry	Method name → callable function	Method resolution
7	RPCServer	Executes registered function	Parameters → result value	Actual computation
8	RPCServer	Creates and serializes response	Result → JSON-RPC response bytes	Response preparation
9	RPCClient	Receives and deserializes response	Bytes → response dict	Response reception
10	RPCClient	Matches response to pending request	Request ID → waiting thread	Response correlation
11	Client Code	Receives result as return value	Response dict → Python object	Transparent result

Detailed Sequence Narrative

Phase 1: Client-Side Request Preparation

The sequence begins when application code invokes what appears to be a method on a local object. For example, when the code calls `calc.add(5, 3)`, the `calc` object is actually an `RPCProxy` instance that intercepts this call through Python's `__getattr__` mechanism. The proxy recognizes that `add` is not a local method and triggers the RPC machinery.

The `RPCProxy` delegates to its underlying `RPCClient`, which performs several critical preparation steps. First, it generates a unique request ID using `generate_request_id()` to ensure this request can be correlated with its eventual response. The client maintains a dictionary of pending requests keyed by these IDs, which is essential for handling concurrent calls on the same connection.

Next, the client constructs a JSON-RPC request message using `create_request_message()`. This transforms the method name "add", parameters `[5, 3]`, and generated request ID into a standardized dictionary format that the server can parse. The message includes version information and follows the JSON-RPC 2.0 specification exactly.

Phase 2: Network Transmission

The client serializes the request message using `serialize_message()`, which converts the Python dictionary to JSON bytes and prepends a 4-byte length prefix for message framing. This length prefix is crucial because TCP provides a byte stream without message boundaries - the server needs to know exactly how many bytes constitute one complete message.

Before sending, the client calls `_ensure_connected()` to establish a TCP connection if one doesn't exist or verify that the existing connection is still healthy. Connection reuse is important for performance since establishing new TCP connections for every RPC call would add significant latency.

The client then transmits the serialized message using `send_all()`, which handles the complexity of partial sends. TCP doesn't guarantee that all bytes are sent in a single `send()` call, so this helper function loops until all bytes are transmitted successfully.

Phase 3: Server-Side Request Processing

The server receives the incoming bytes through its `handle_connection()` method running in a dedicated thread for this client connection. The server first reads the 4-byte length prefix to determine how many additional bytes constitute the complete message, then reads exactly that many bytes using `recv_all()`.

Once the complete message is received, the server deserializes it back into a Python dictionary using `deserialize_message()`. The server validates that this is a well-formed JSON-RPC request with all required fields (method, params, id, jsonrpc version).

The server's `process_request()` method handles the core dispatch logic. It extracts the method name from the request and looks it up in the `MethodRegistry`. If the method exists, the server unpacks the parameters and invokes the registered function directly. The beauty of this approach is that any Python

function can be registered - the server doesn't need special knowledge about what "add" does, it just calls the registered function with the provided parameters.

Phase 4: Server-Side Response Generation

After the registered function executes successfully and returns a result, the server packages this result into a JSON-RPC response message using `create_response_message()`. The response includes the original request ID (crucial for correlation), the result value, and the JSON-RPC version identifier.

The server serializes this response message the same way as requests - JSON encoding with a length prefix - and transmits it back to the client using the same TCP connection. The bidirectional nature of the connection allows responses to flow back through the same socket.

Phase 5: Client-Side Response Handling

Meanwhile, the client has been blocking in `_wait_for_response()`, continuously reading from the socket for incoming messages. When the response arrives, the client deserializes it and extracts the request ID to determine which pending request this response satisfies.

The client's request tracking mechanism uses the ID to find the correct thread or callback waiting for this response. For blocking calls, this typically means unblocking the thread that made the original RPC call and returning the result value.

Finally, the client cleans up the request from its pending requests dictionary and returns the result to the application code. From the application's perspective, the call to `calc.add(5, 3)` simply returned `8` - all the network complexity was hidden by the RPC framework.

Timing and Concurrency Considerations

The RPC call sequence must handle several timing and concurrency challenges that don't exist in local function calls:

Request ID Management: Since multiple threads might make concurrent RPC calls on the same client connection, the request ID mechanism prevents responses from being delivered to the wrong caller. Each request gets a unique ID, and responses are routed back to the correct waiting thread.

Connection State: The client must manage the TCP connection lifecycle carefully. Connections can break at any time due to network issues, server restarts, or firewall timeouts. The client detects these failures during send or receive operations and can optionally retry or reconnect.

Server Threading: The server handles multiple client connections concurrently, with each connection processed in its own thread. Within each connection thread, requests are processed sequentially to maintain ordering guarantees. This threading model is simple but scales reasonably well for moderate loads.

> **Key Insight**: The request ID serves as the critical coordination mechanism that allows multiple concurrent RPC calls to share the same TCP connection without interference. Without this correlation mechanism, responses could be delivered to the wrong caller, causing subtle and hard-to-debug errors.

Error Flow

Error handling in RPC systems is significantly more complex than local function calls because failures can occur at multiple layers - network, protocol, serialization, and application logic. The error flow must gracefully handle each category and provide meaningful feedback to the client application.

Error Categories and Propagation

RPC errors fall into four distinct categories, each handled differently by the framework:

Error Category	Detection Point	Error Type	Propagation Method	Client Experience
Network Errors	Client or Server	`RPCTransportError`	Exception raised immediately	Connection refused, timeout
Protocol Errors	Client or Server	`RPCProtocolError`	JSON-RPC error response	Invalid request format
Method Errors	Server	`RPCMethodError`	JSON-RPC error response	Method not found, bad params
Execution Errors	Server	`RPCMethodError`	JSON-RPC error response	Exception in user code

Network Error Flow

Network errors represent failures in the underlying TCP communication and are the most disruptive type of RPC failure. These errors are detected at the socket layer and cannot be communicated through the normal JSON-RPC response mechanism because the communication channel itself is compromised.

When a network error occurs during `send_all()` or `recv_all()`, the client immediately raises an `RPCTransportError` exception. The client also marks its connection as broken and closes the socket to prevent further attempts to use the damaged connection. Subsequent RPC calls will trigger connection re-establishment.

Common network error scenarios include:

1. **Connection Refused**: Server is not running or not accepting connections on the specified port
2. **Connection Reset**: Server process crashed or was forcibly terminated

3. **Connection Timeout**: Network is unreachable or server is not responding within timeout period
4. **Partial Send/Receive**: Connection was broken mid-transmission, leaving message incomplete

The client handles these by immediately cleaning up the broken connection and propagating the transport error to the application. The application can choose to retry, connect to an alternative server, or fail gracefully.

Protocol Error Flow

Protocol errors occur when messages don't conform to the JSON-RPC specification, even though network communication is successful. These errors can be detected by either client or server during message parsing and validation.

Server-detected protocol errors follow this sequence:

1. Server receives and deserializes message successfully
2. Server validates message format and discovers violation (missing required field, invalid JSON-RPC version, malformed structure)
3. Server creates JSON-RPC error response with appropriate error code (`PARSE_ERROR` , `INVALID_REQUEST`)
4. Server sends error response back to client using normal response mechanism
5. Client receives error response and raises `RPCProtocolError` with server's error details

This approach allows protocol errors to be communicated through the normal RPC channel, providing detailed error information to help debug message format issues.

Method Error Flow

Method errors occur when the JSON-RPC message is well-formed but the requested method cannot be executed. This includes method-not-found errors and parameter validation failures.

The method error sequence:

1. Server successfully parses and validates request message format
2. Server attempts to look up requested method name in `MethodRegistry`
3. If method not found, server creates error response with `METHOD_NOT_FOUND` error code
4. If method found but parameter count/types don't match, server creates error response with `INVALID_PARAMS` error code
5. Server sends JSON-RPC error response with detailed error information
6. Client receives error response and raises `RPCMethodError` with method-specific error details

This category of errors is particularly important for API usability - clear error messages help developers understand what methods are available and how to call them correctly.

Execution Error Flow

Execution errors represent exceptions thrown by the registered method during actual execution. These are application-level errors that occur after successful method dispatch but during the business logic execution.

The execution error handling sequence:

1. Server successfully looks up method and begins execution
2. Registered method raises exception during execution (divide by zero, invalid business logic, etc.)
3. Server catches exception and examines its type
4. Server creates JSON-RPC error response with `INTERNAL_ERROR` code and exception details
5. Server sends error response back to client
6. Client receives error response and raises `RPCMethodError` with execution error details

> **Design Decision: Exception Serialization**
>
> - **Context**: When user methods throw exceptions, we need to decide how much exception detail to send to clients
> - **Options Considered**:
>   1. Send full exception traceback including server file paths
>   2. Send only exception message and type
>   3. Send generic "internal error" message
> - **Decision**: Send exception type and message but not full traceback
> - **Rationale**: Provides useful debugging information without exposing server implementation details or file system structure
> - **Consequences**: Clients get actionable error information while server security is maintained

Error Response Message Format

All server-detected errors (protocol, method, and execution errors) are communicated using standardized JSON-RPC error response messages. This consistency allows clients to handle all server errors through the same mechanism.

Field	Type	Description	Example Value
jsonrpc	str	JSON-RPC version identifier	"2.0"
id	str/null	Request ID from original request	"req_12345"
error	dict	Error details object	See error object format

The error object within the response contains structured error information:

Field	Type	Description	Example Value
code	int	Standard JSON-RPC error code	-32601
message	str	Human-readable error description	"Method not found"
data	any	Additional error-specific information	{"method": "nonexistent"}

Timeout Handling

Timeout errors deserve special attention because they can occur even when both client and server are functioning correctly but network latency is high or server processing time exceeds expectations.

The client's timeout mechanism works as follows:

1. Client starts timeout timer when sending request
2. Client blocks in `_wait_for_response()` but checks elapsed time periodically
3. If timeout expires before response arrives, client raises `RPCTimeoutError`
4. Client marks request as timed out but keeps connection open (timeout doesn't necessarily indicate connection failure)
5. If response eventually arrives for timed-out request, client discards it to prevent delivering stale results

> **Important Consideration**: Timeout handling creates a potential resource leak. If the server eventually sends a response for a timed-out request, that response must be properly discarded to prevent it from being mistakenly matched to a future request with the same ID.

Error Recovery Strategies

Different error categories suggest different recovery strategies for client applications:

Error Type	Suggested Recovery	Rationale
`RPCTransportError`	Retry with exponential backoff, try alternative server	Network issues often transient
`RPCProtocolError`	Fix client code, don't retry	Protocol violations indicate programming errors
`RPCMethodError` (not found)	Fix client code, check available methods	Method name typos or API misunderstanding
`RPCMethodError` (invalid params)	Fix client code parameters	Parameter type or count mismatch
`RPCMethodError` (execution)	Retry with different parameters or fail gracefully	Server-side business logic error
`RPCTimeoutError`	Retry with longer timeout or fail	Server may be overloaded

Common Error Scenarios

Real-world RPC systems encounter several common error patterns that developers should be prepared to handle:

⚠️ **Pitfall: Ignoring Network Errors** Many developers assume network connections are reliable and don't properly handle `RPCTransportError` exceptions. In production environments, networks frequently experience transient failures, server restarts, and connection drops. Applications should catch transport errors and implement appropriate retry logic rather than crashing on the first network hiccup.

⚠️ **Pitfall: Confusing Timeout with Failure** A timeout doesn't necessarily mean the server is broken - it might just be processing a complex request that takes longer than expected. Immediately reducing timeout values or switching to alternative servers can make problems worse. Instead, applications should distinguish between timeout errors and other error types, potentially retrying with longer timeouts.

⚠️ **Pitfall: Leaking Request IDs** If request IDs are not properly managed, the client can run out of unique identifiers or accidentally reuse IDs while previous requests are still pending. This leads to response correlation errors where responses are delivered to the wrong callers. Proper cleanup in the error path is essential.

Implementation Guidance

The error handling implementation requires careful coordination between exception types, error codes, and message formats to provide a consistent developer experience.

Technology Recommendations

Component	Simple Option	Advanced Option
Error Types	Standard Python exceptions with inheritance	Custom exception hierarchy with error codes
Error Serialization	JSON with string messages	Structured error objects with metadata
Timeout Mechanism	Simple socket timeout	Configurable per-request timeouts
Retry Logic	Manual retry in application code	Automatic retry with backoff in client

Recommended File Structure

```
rpc_framework/
├── exceptions.py          ← All RPC exception classes
├── error_codes.py         ← JSON-RPC error code constants
├── client.py              ← Client with error handling
├── server.py              ← Server with error responses
└── protocol.py            ← Error message creation helpers
```

Infrastructure Starter Code

Complete Exception Hierarchy (exceptions.py)

```python
"""RPC Framework exception hierarchy for structured error handling."""


class RPCError(Exception):
    """Base exception for all RPC framework errors."""
    def __init__(self, message, error_code=None, error_data=None):
        super().__init__(message)
        self.error_code = error_code
        self.error_data = error_data


class RPCTransportError(RPCError):
    """Network communication errors - connection refused, timeout, etc."""
    pass


class RPCProtocolError(RPCError):
    """JSON-RPC protocol violations - malformed messages, version mismatch."""
    pass


class RPCMethodError(RPCError):
    """Method-related errors - not found, invalid params, execution failure."""
    pass


class RPCTimeoutError(RPCError):
    """Request timeout - server didn't respond within specified time limit."""
    def __init__(self, message, timeout_seconds):
        super().__init__(message)
        self.timeout_seconds = timeout_seconds
```

PYTHON

Error Code Constants (error_codes.py)

```python
"""JSON-RPC 2.0 standard error codes and framework-specific extensions."""


# JSON-RPC 2.0 Standard Error Codes
PARSE_ERROR = -32700      # Invalid JSON received
INVALID_REQUEST = -32600  # JSON is not valid JSON-RPC request
METHOD_NOT_FOUND = -32601 # Method does not exist or is not available
INVALID_PARAMS = -32602   # Invalid method parameter(s)
INTERNAL_ERROR = -32603   # Internal JSON-RPC error


# Framework-specific Error Codes (per JSON-RPC spec, -32000 to -32099 reserved)
TRANSPORT_ERROR = -32000  # Network communication failure
TIMEOUT_ERROR = -32001    # Request timeout
CONNECTION_ERROR = -32002 # Connection establishment failure
```

Core Error Handling Skeleton

Server Error Response Logic

```python
def process_request(self, request):
    """Process incoming RPC request and return response or error."""
    request_id = request.get('id')


    try:
        # TODO 1: Validate JSON-RPC message format (version, required fields)
        # TODO 2: Extract method name and parameters from request
        # TODO 3: Look up method in registry - raise METHOD_NOT_FOUND if missing
        # TODO 4: Validate parameter count/types against method signature
        # TODO 5: Execute method with parameters - catch any exceptions
        # TODO 6: Return create_response_message(request_id, result)
        pass
    except KeyError as e:
        # Missing required field in request
        return create_error_message(request_id, INVALID_REQUEST,
                                    f"Missing required field: {e}")
    except TypeError as e:
        # Parameter count/type mismatch
        return create_error_message(request_id, INVALID_PARAMS, str(e))
    except Exception as e:
        # Execution error in user method
        return create_error_message(request_id, INTERNAL_ERROR,
                                    f"{type(e).__name__}: {e}")
```

PYTHON

Client Error Handling Logic

```python
def _handle_response(self, response):
    """Convert JSON-RPC response to result or raise appropriate exception."""
    if 'error' in response:
        error = response['error']
        error_code = error.get('code')
        error_message = error.get('message', 'Unknown error')
        error_data = error.get('data')

        # TODO 1: Check error code and raise appropriate exception type
        # TODO 2: For PARSE_ERROR, INVALID_REQUEST -> RPCProtocolError
        # TODO 3: For METHOD_NOT_FOUND, INVALID_PARAMS -> RPCMethodError
        # TODO 4: For INTERNAL_ERROR -> RPCMethodError with execution details
        # TODO 5: For TIMEOUT_ERROR -> RPCTimeoutError
        # TODO 6: For unknown codes -> generic RPCError
        pass
    else:
        return response.get('result')
```

Milestone Checkpoints

Error Handling Verification

After implementing error handling, verify these behaviors:

1. **Network Errors**: Stop the server, make RPC call, should get `RPCTransportError`
2. **Method Not Found**: Call non-existent method, should get `RPCMethodError` with METHOD_NOT_FOUND code
3. **Invalid Parameters**: Call method with wrong parameter count, should get `RPCMethodError` with INVALID_PARAMS code
4. **Execution Errors**: Register method that throws exception, call it, should get `RPCMethodError` with exception details
5. **Timeout Errors**: Make call with very short timeout to slow method, should get `RPCTimeoutError`

Test Commands

```python
# Test script to verify error handling
client = RPCClient('localhost', 8000)

# Test 1: Network error (server not running)
try:
    client.call('add', 1, 2)
    print("ERROR: Should have raised RPCTransportError")
except RPCTransportError:
    print("✓ Network error handling works")

# Test 2: Method not found (server running)
try:
    client.call('nonexistent_method')
    print("ERROR: Should have raised RPCMethodError")
except RPCMethodError as e:
    if e.error_code == METHOD_NOT_FOUND:
        print("✓ Method not found error handling works")
```

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Generic "connection error"	Not distinguishing error types	Check exception type in handler	Use specific exception classes
Wrong error delivered to client	Response correlation failure	Log request IDs on both sides	Ensure request ID in error response matches
Timeout but server still processing	Short timeout, long operation	Monitor server logs for completion	Increase timeout or make operation async
Connection keeps breaking	Not handling partial sends/receives	Check socket error logs	Use send_all/recv_all helpers
Error details lost	Only passing error message	Check error object serialization	Include error code and data fields

Error Handling and Edge Cases

> **Milestone(s):** All milestones - error handling is critical throughout message protocol (Milestone 1), server implementation (Milestone 2), and client implementation (Milestone 3)

Mental Model: The Emergency Response System

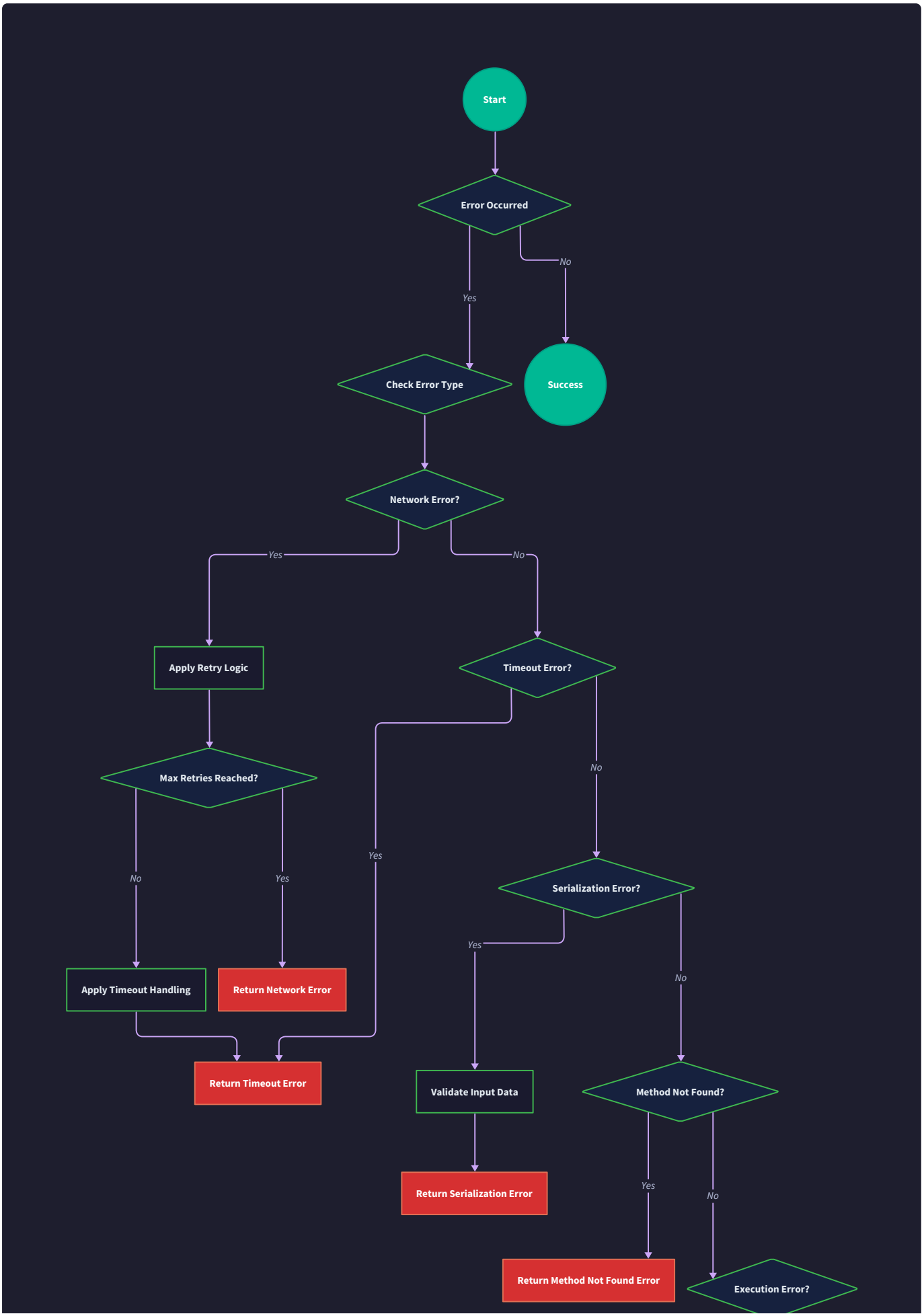
Think of RPC error handling like a well-designed emergency response system in a large organization. Just as emergency responders need clear protocols to categorize incidents (fire vs. medical vs. security), identify the appropriate response team, and escalate through proper channels, an RPC framework needs systematic error categorization, handling strategies, and propagation mechanisms.

When a fire alarm sounds, the emergency system doesn't just panic - it identifies the type of emergency, determines the severity, notifies the right responders, and follows established procedures. Similarly, when network connections fail or methods throw exceptions, our RPC framework must categorize the error, determine the appropriate recovery action, and ensure the client receives meaningful information about what went wrong and what they can do about it.

The key insight is that different types of errors require fundamentally different handling strategies. A network timeout should trigger retry logic, while a "method not found" error should immediately return to the client with diagnostic information. Just as you wouldn't send firefighters to handle a medical emergency, you shouldn't handle serialization errors the same way you handle connection failures.

Error Categories

Our RPC framework encounters errors from multiple sources and at different stages of request processing. Understanding these categories is crucial because each requires distinct handling strategies and recovery mechanisms.


Start
Error Occurred
No
Yes
Success
Check Error Type
Network Error?
Yes
No
Apply Retry Logic
Timeout Error?
Max Retries Reached?
No
Yes
No
Yes
Serialization Error?
Yes
No
Apply Timeout Handling
Return Network Error
Return Timeout Error
Validate Input Data
Method Not Found?
Return Serialization Error
Yes
No
Return Method Not Found Error
Execution Error?

The error taxonomy follows a hierarchy based on where the error originates and whether recovery is possible:

Error Category	Description	Recovery Strategy	Client Experience
Transport Errors	Network connectivity, socket operations, connection management failures	Retry with exponential backoff, connection re-establishment	`RPCTransportError` with retry suggestion
Protocol Errors	Message parsing, invalid JSON-RPC format, serialization failures	Immediate failure, diagnostic information	`RPCProtocolError` with format details
Method Errors	Method not found, parameter validation, execution exceptions	Method-specific handling, error propagation	`RPCMethodError` with method context
Timeout Errors	Request exceeds configured time limit, server unresponsive	Configurable retry or immediate failure	`RPCTimeoutError` with timeout duration
Internal Errors	Server bugs, resource exhaustion, unexpected conditions	Graceful degradation, logging, monitoring alerts	Generic error with request ID for support

Architecture Decision: Error Type Hierarchy

- **Context**: Need structured way to represent different error conditions with appropriate handling
- **Options Considered**:
  1. Single generic error class with error codes
  2. Flat error classes without inheritance
  3. Hierarchical error classes inheriting from base `RPCError`
- **Decision**: Hierarchical error classes with specific types for each category
- **Rationale**: Type system helps client code handle different errors appropriately, inheritance reduces code duplication, specific types enable targeted recovery strategies
- **Consequences**: More complex type definitions but clearer error handling logic and better debugging experience

The error class hierarchy provides both type safety and semantic clarity:

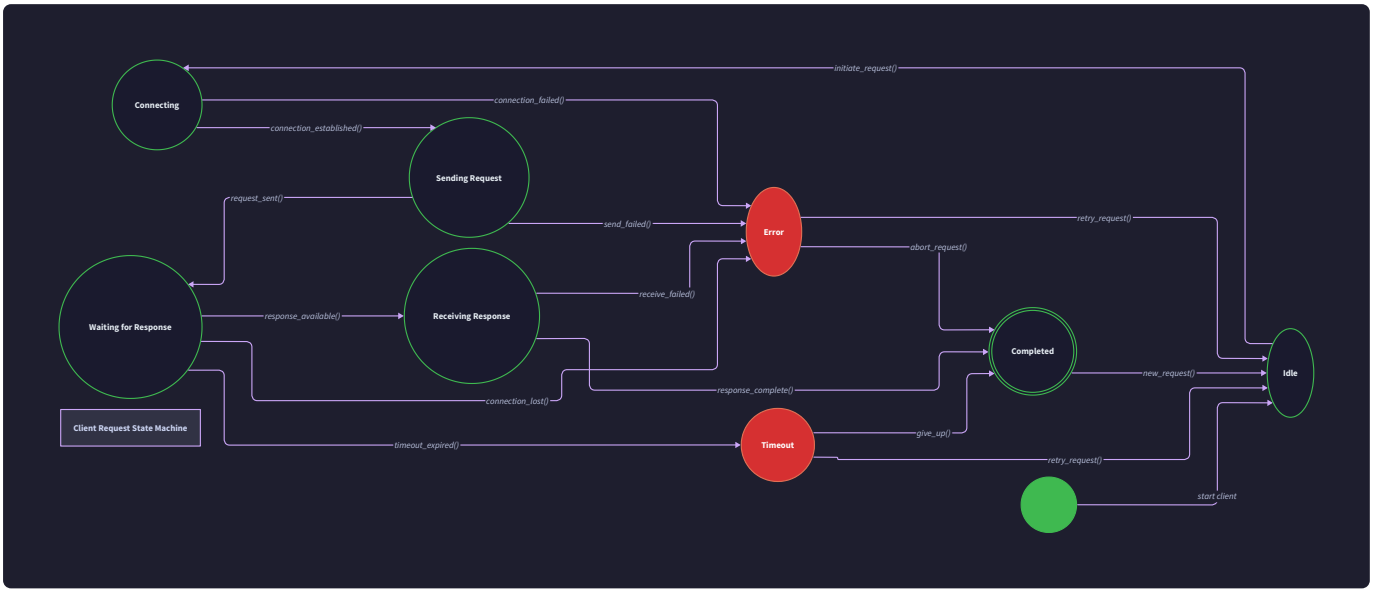
Error Class	Base Fields	Additional Fields	Usage Context
`RPCError`	`message: str`, `error_code: int`, `error_data: any`	None (base class)	Generic error handling, logging
`RPCTransportError`	Inherits base fields	`host: str`, `port: int`, `operation: str`	Connection failures, socket errors
`RPCProtocolError`	Inherits base fields	`raw_data: bytes`, `parse_stage: str`	Message format violations, encoding issues
`RPCMethodError`	Inherits base fields	`method_name: str`, `params: list`, `execution_phase: str`	Method dispatch and execution failures
`RPCTimeoutError`	Inherits base fields	`timeout_seconds: float`, `elapsed_seconds: float`	Request timeout handling

Each error type carries contextual information that enables intelligent handling. For example, `RPCTransportError` includes the host and port for connection diagnostics, while `RPCMethodError` includes the method name and parameters for debugging method invocations.

> The critical insight is that error context determines recovery strategy. Transport errors often warrant retries, while protocol errors indicate client bugs that retries cannot fix.

Network Failure Handling

Network failures represent the most common and complex error category in distributed systems. Unlike local function calls that either succeed or raise exceptions, network operations introduce a spectrum of partial failure modes that require sophisticated handling strategies.



The network failure landscape includes multiple failure modes, each with distinct characteristics and recovery approaches:

Failure Mode	Detection Method	Immediate Action	Recovery Strategy
Connection Refused	`socket.connect()` raises `ConnectionRefusedError`	Mark connection as failed	Exponential backoff retry, server health check
Connection Reset	`socket.send()` raises `ConnectionResetError`	Close socket, clear connection state	Re-establish connection, replay request
Connection Timeout	`socket.settimeout()` expires during operation	Abort current operation	Retry with increased timeout or different endpoint
Partial Send/Receive	`socket.send()` returns fewer bytes than requested	Continue operation from partial position	Complete the operation or abort after threshold
Sudden Disconnection	`socket.recv()` returns empty bytes	Detect connection closure	Re-establish connection, determine request status
DNS Resolution Failure	`socket.getaddrinfo()` raises `gaierror`	Cache negative result temporarily	Retry with different DNS server or cached IP

The connection state machine helps understand valid transitions and error recovery points:

Current State	Network Event	Next State	Recovery Action
`DISCONNECTED`	Connection refused	`DISCONNECTED`	Schedule retry with backoff
`CONNECTING`	Timeout during connect	`DISCONNECTED`	Try next address or increase timeout
`CONNECTED`	Socket error during send	`DISCONNECTED`	Re-queue request for retry
`WAITING_RESPONSE`	Connection reset	`DISCONNECTED`	Mark request as failed, retry if idempotent
`RECEIVING_DATA`	Partial receive timeout	`DISCONNECTED`	Abort request, connection likely dead

The `send_all` and `recv_all` functions implement robust network I/O with proper timeout handling and partial operation support. These functions abstract the complexity of socket operations while providing consistent error reporting:

Function	Responsibility	Timeout Behavior	Error Propagation
`send_all(sock, data, timeout)`	Send complete message handling partial sends	Per-operation timeout, not total timeout	Raises `RPCTransportError` with bytes sent
`recv_all(sock, size, timeout)`	Receive exact byte count handling partial receives	Cumulative timeout across all recv operations	Raises `RPCTransportError` with bytes received
`recv_message(sock, timeout)`	Receive length-prefixed message	Timeout applies to complete message	Raises `RPCProtocolError` for invalid framing

Architecture Decision: Connection Recovery Strategy

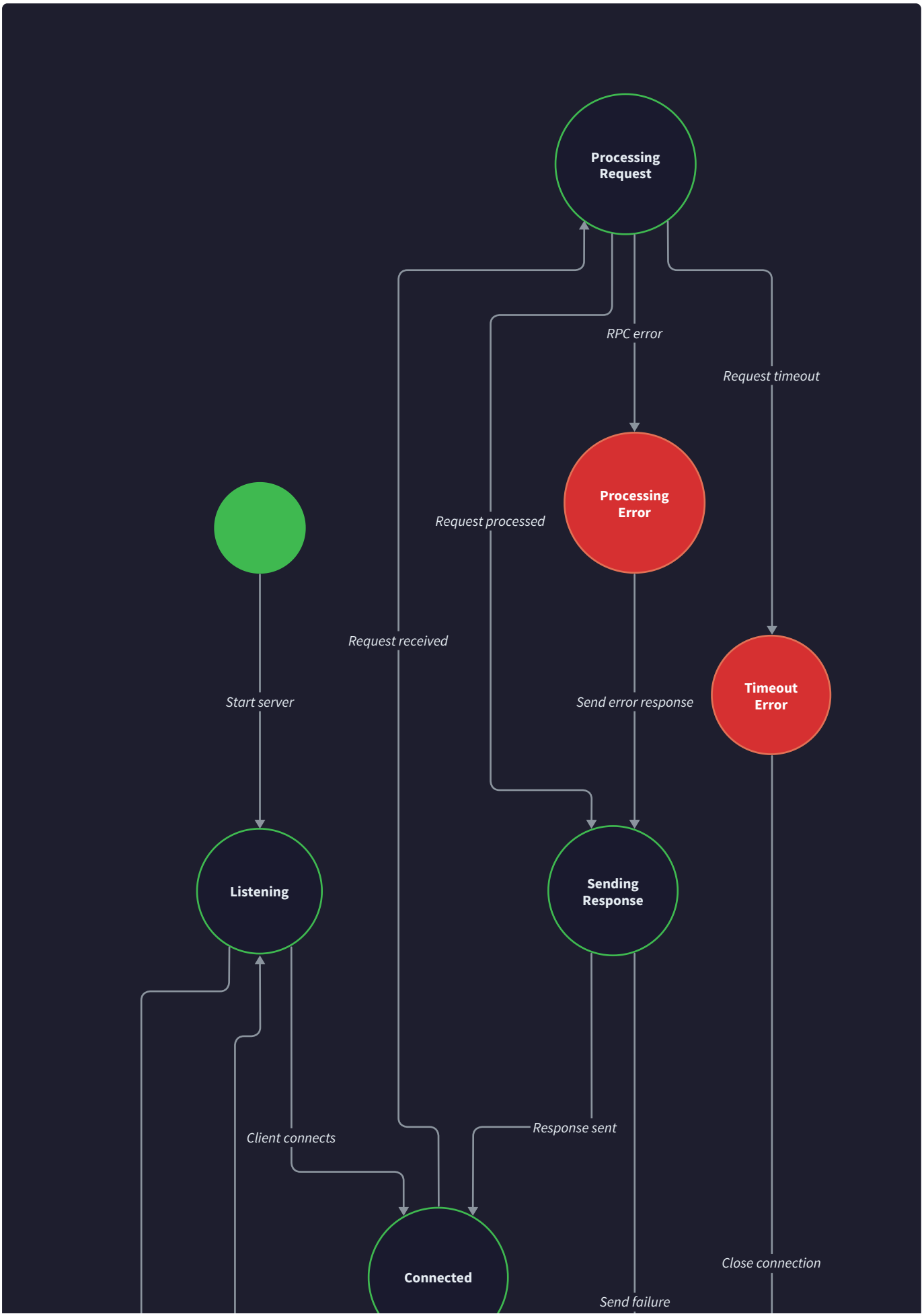
- **Context**: Network connections can fail at any point during RPC communication
- **Options Considered**:
  1. Immediate failure - no retry logic
  2. Transparent retry - hide failures from client
  3. Configurable retry - client controls retry behavior
- **Decision**: Configurable retry with exponential backoff and maximum attempt limits
- **Rationale**: Balances reliability (automatic recovery from transient failures) with control (client can tune for their use case) and prevents infinite retry loops
- **Consequences**: More complex client implementation but better resilience to network issues and clearer failure semantics

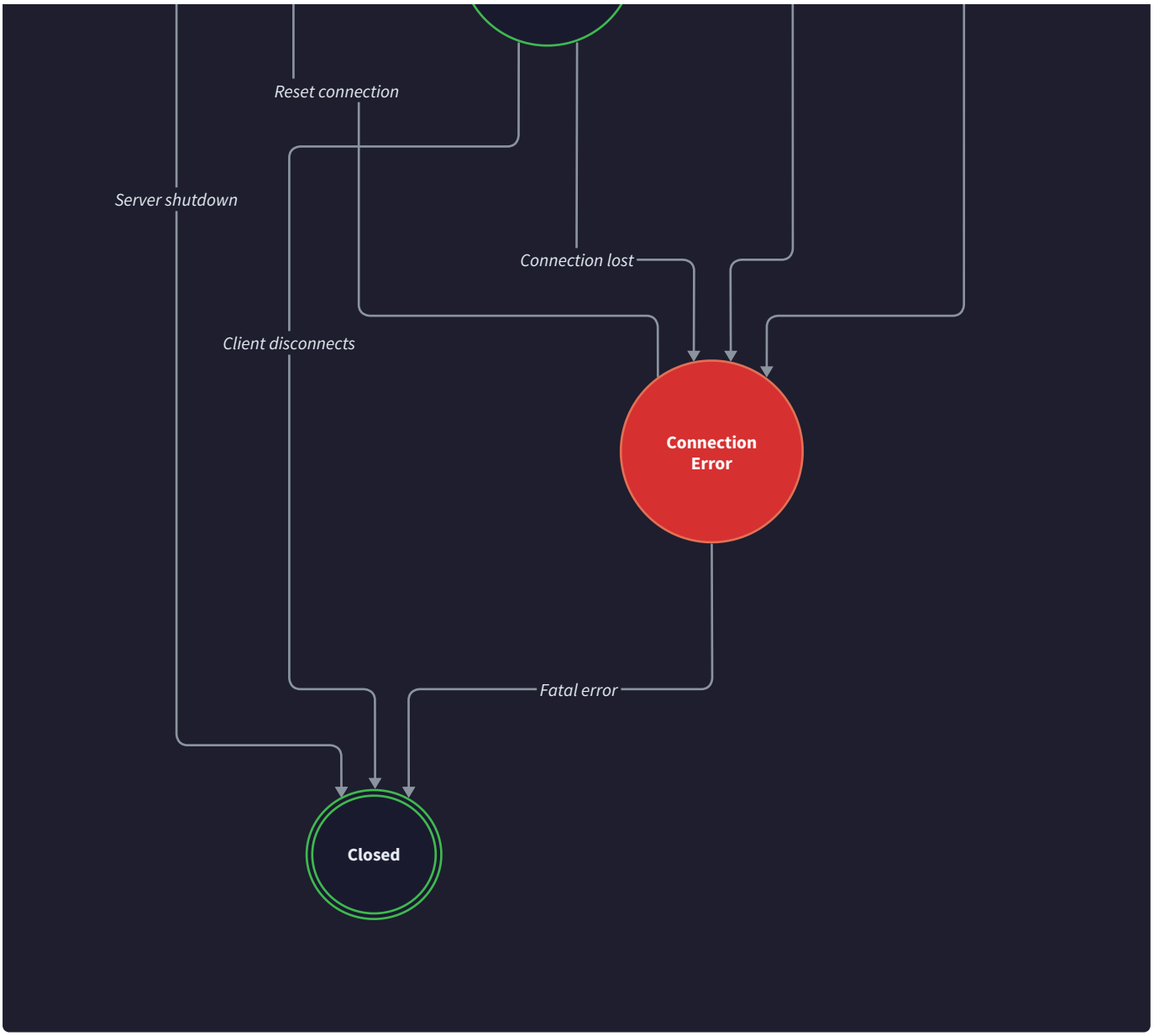
The retry mechanism uses exponential backoff to avoid overwhelming failed servers while providing reasonable recovery times for transient network issues:

Retry Attempt	Base Delay	Backoff Multiplier	Maximum Delay	Total Elapsed
1	100ms	1.0x	100ms	100ms
2	100ms	2.0x	200ms	300ms
3	100ms	4.0x	400ms	700ms
4	100ms	8.0x	800ms	1500ms
5	100ms	16.0x	5000ms (capped)	6500ms

Serialization Error Handling

Serialization errors occur when converting between Python objects and the JSON wire format. Unlike network errors that might be transient, serialization errors usually indicate programming bugs, data corruption, or incompatible message formats between client and server versions.


Processing Request
RPC error
Request timeout
Processing Error
Request processed
Request received
Timeout Error
Start server
Send error response
Listening
Sending Response
Client connects
Response sent
Connected
Close connection
Send failure



The serialization error landscape encompasses both encoding and decoding failures:

Error Type	Occurrence Point	Common Causes	Detection Method
JSON Encoding Errors	`serialize_message()` on client/server	Non-serializable objects, circular references	`json.dumps()` raises `TypeError`
JSON Decoding Errors	`deserialize_message()` on client/server	Malformed JSON, truncated messages	`json.loads()` raises `json.JSONDecodeError`
Type Conversion Errors	Parameter processing in method dispatch	Wrong parameter types, missing fields	Type validation raises `TypeError`
Message Format Errors	Request/response validation	Missing required fields, invalid structure	Schema validation fails
Encoding/Decoding Errors	Byte string conversion	Unicode issues, character encoding problems	`str.encode()` raises `UnicodeError`
Size Limit Errors	Large message handling	Messages exceeding configured limits	Length prefix validation fails

The message processing pipeline includes validation at multiple stages to catch serialization errors early and provide diagnostic information:

Processing Stage	Validation Applied	Error Type Raised	Recovery Action
Pre-Serialization	Object type checking, circular reference detection	`RPCProtocolError`	Reject request with validation details
JSON Encoding	JSON serialization compatibility	`RPCProtocolError`	Log object types, suggest alternatives
Message Framing	Length prefix validation, size limits	`RPCProtocolError`	Reject oversized messages with limit info
Network Transport	Byte encoding, socket compatibility	`RPCTransportError`	Encoding failure, suggest UTF-8
JSON Decoding	JSON parsing, structure validation	`RPCProtocolError`	Log raw data for debugging
Post-Deserialization	Message schema validation, required fields	`RPCProtocolError`	Missing field details for client

The `deserialize_message` function implements defensive parsing with comprehensive error context:

Parse Phase	Validation Check	Error Condition	Error Context Provided
Length Prefix	4-byte integer, reasonable size	Invalid prefix, oversized message	Raw bytes, expected format
JSON Parsing	Valid JSON syntax	Malformed JSON, encoding issues	Parse position, syntax error details
Schema Validation	Required fields present	Missing fields, wrong types	Field name, expected vs actual type
Content Validation	Method names, parameter types	Invalid characters, unsupported types	Specific validation rule violated

Architecture Decision: Serialization Error Recovery

- **Context**: Serialization failures can occur during encoding (client) or decoding (server/client)
- **Options Considered**:
  1. Fail fast - abort operation immediately
  2. Best effort - attempt partial serialization/skip problematic fields
  3. Structured errors - detailed diagnostics with recovery suggestions
- **Decision**: Structured errors with detailed diagnostics and no data corruption
- **Rationale**: Data integrity is more important than availability; partial serialization could cause subtle bugs; detailed errors help developers fix issues quickly
- **Consequences**: Operations fail completely on serialization errors, but developers get clear guidance for fixes

The error response format for serialization failures includes diagnostic information:

Error Field	Content	Purpose	Example Value
`error_code`	Standard JSON-RPC error code	Client error categorization	`PARSE_ERROR` (-32700)
`message`	Human-readable description	Developer debugging	"Invalid JSON syntax at position 45"
`error_data.raw_data`	Original problematic data (truncated)	Debugging the exact input	`"{\"method\": \"test\", \"params\": [1,2,3"`
`error_data.parse_stage`	Where parsing failed	Narrowing down the issue	"json_decode" or "schema_validation"
`error_data.expected_format`	What was expected	Helping fix the problem	"JSON-RPC 2.0 request object"
`error_data.suggestions`	Possible fixes	Actionable guidance	["Check for missing closing brace", "Verify UTF-8 encoding"]

⚠️ Pitfall: Logging Sensitive Data in Errors

A common mistake is including complete message content in error responses, potentially exposing sensitive information. Always truncate or sanitize error data before sending it over the network. The raw data should be logged server-side for debugging but not transmitted to clients unless specifically configured for development environments.

⚠️ Pitfall: Partial Message Processing

When message framing fails (invalid length prefix or truncated data), avoid attempting to parse partial JSON. This can lead to interpreting part of one message as the beginning of the next message, causing cascading parse errors. Always validate the complete message frame before attempting JSON parsing.

⚠️ Pitfall: Unicode Encoding Assumptions

Don't assume all messages use UTF-8 encoding. While JSON requires UTF-8, network data might be corrupted or use different encodings. Always handle `UnicodeDecodeError` exceptions and provide encoding diagnostics in error responses.

Implementation Guidance

The error handling system provides the foundation for reliable RPC communication by transforming various failure modes into structured, actionable error information that clients can handle appropriately.

Technology Recommendations

Component	Simple Option	Advanced Option
Exception Hierarchy	Built-in Python exceptions with custom messages	Custom exception classes with structured data
Error Serialization	JSON with standard fields	MessagePack with binary error data
Logging Integration	Python `logging` module with structured logs	Structured logging with correlation IDs
Network Error Detection	Socket exception handling	Connection health monitoring with heartbeats
Retry Logic	Simple exponential backoff	Circuit breaker pattern with failure tracking

Core Error Classes

```python
"""
RPC Framework Error Handling

This module provides comprehensive error handling for all RPC operations
including network failures, serialization issues, and method execution errors.
"""

import json
import socket
import time
from typing import Any, Optional, Dict, Union
from threading import Lock

class RPCError(Exception):
    """Base class for all RPC-related errors."""

    def __init__(self, message: str, error_code: int = -32603, error_data: Any = None):
        super().__init__(message)
        self.message = message
        self.error_code = error_code
        self.error_data = error_data or {}

    def to_dict(self) -> Dict[str, Any]:
        """Convert error to JSON-RPC error object format."""
        # TODO 1: Create error object with code, message, and data fields
        # TODO 2: Include error_data only if it contains useful information
        # TODO 3: Sanitize error_data to prevent sensitive information leakage
```

```
        pass


    def __str__(self) -> str:
        """Human-readable error representation for logging."""
        # TODO 1: Format error with code, message, and key data fields
        # TODO 2: Truncate large data fields to prevent log spam
        pass

class RPCTransportError(RPCError):
    """Network and transport-related errors."""


    def __init__(self, message: str, host: str = "", port: int = 0,
                 operation: str = "", original_error: Exception = None):
        error_data = {
            "host": host,
            "port": port,
            "operation": operation,
            "original_error": str(original_error) if original_error else None
        }
        super().__init__(message, -32000, error_data)
        self.host = host
        self.port = port
        self.operation = operation
        self.original_error = original_error

class RPCProtocolError(RPCError):
    """Message format and serialization errors."""
```

```
    def __init__(self, message: str, raw_data: bytes = b"", parse_stage: str = ""):
        error_data = {
            "raw_data": raw_data[:100].decode('utf-8', errors='replace') if raw_data else
"",
            "parse_stage": parse_stage,
            "data_length": len(raw_data)
        }
        super().__init__(message, -32700, error_data)
        self.raw_data = raw_data
        self.parse_stage = parse_stage

class RPCMethodError(RPCError):
    """Method execution and dispatch errors."""

    def __init__(self, message: str, method_name: str = "", params: list = None,
                 execution_phase: str = "", original_error: Exception = None):
        error_data = {
            "method_name": method_name,
            "params": params or [],
            "execution_phase": execution_phase,
            "original_error": str(original_error) if original_error else None
        }
        super().__init__(message, -32601, error_data)
        self.method_name = method_name
        self.params = params or []
        self.execution_phase = execution_phase
        self.original_error = original_error
```

```
class RPCTimeoutError(RPCError):

    """Request timeout errors."""



    def __init__(self, message: str, timeout_seconds: float, elapsed_seconds: float = 0.0):

        error_data = {

            "timeout_seconds": timeout_seconds,

            "elapsed_seconds": elapsed_seconds,

            "timed_out": elapsed_seconds >= timeout_seconds

        }

        super().__init__(message, -32001, error_data)

        self.timeout_seconds = timeout_seconds

        self.elapsed_seconds = elapsed_seconds


# Standard JSON-RPC error codes

class ErrorCode:

    PARSE_ERROR = -32700

    INVALID_REQUEST = -32600

    METHOD_NOT_FOUND = -32601

    INVALID_PARAMS = -32602

    INTERNAL_ERROR = -32603

    TRANSPORT_ERROR = -32000

    TIMEOUT_ERROR = -32001

    CONNECTION_ERROR = -32002
```

Network Operation Helpers

```python
"""
Robust network operations with comprehensive error handling.
"""

import select
import errno

class SocketHelper:
    """Utility class for robust socket operations with timeout handling."""

    @staticmethod
    def send_all(sock: socket.socket, data: bytes, timeout: float) -> None:
        """Send all data, handling partial sends and timeouts."""
        # TODO 1: Set socket to non-blocking mode for timeout control
        # TODO 2: Use select() to wait for socket writability with timeout
        # TODO 3: Handle partial sends by tracking bytes sent and continuing
        # TODO 4: Raise RPCTransportError with context on failures
        # TODO 5: Restore original socket blocking mode before returning
        # Hint: socket.send() may send fewer bytes than requested
        pass

    @staticmethod
    def recv_all(sock: socket.socket, size: int, timeout: float) -> bytes:
        """Receive exact number of bytes, handling partial receives."""
        # TODO 1: Set socket to non-blocking mode and track received data
        # TODO 2: Loop until all requested bytes are received
        # TODO 3: Use select() with remaining timeout for each recv operation
```

```
        # TODO 4: Handle partial receives by accumulating data
        # TODO 5: Detect connection closure (recv returns 0 bytes)
        # TODO 6: Raise appropriate errors for timeouts and connection issues
        pass

    @staticmethod
    def recv_message(sock: socket.socket, timeout: float) -> bytes:
        """Receive length-prefixed message."""
        # TODO 1: Receive 4-byte length prefix using recv_all
        # TODO 2: Unpack length as big-endian unsigned integer
        # TODO 3: Validate message length is reasonable (not > MAX_MESSAGE_SIZE)
        # TODO 4: Receive message body using recv_all with remaining timeout
        # TODO 5: Return complete message bytes
        # Hint: Use struct.unpack('>I', length_bytes)[0] for length
        pass

    @staticmethod
    def send_message(sock: socket.socket, message_bytes: bytes, timeout: float) -> None:
        """Send length-prefixed message."""
        # TODO 1: Create 4-byte length prefix using struct.pack('>I', len(message))
        # TODO 2: Send length prefix using send_all
        # TODO 3: Send message body using send_all
        # TODO 4: Handle any transport errors and re-raise with context
        pass

def handle_socket_error(error: Exception, operation: str, host: str = "", port: int = 0) ->
RPCTransportError:
    """Convert socket exceptions to structured RPC transport errors."""
```

```
    # TODO 1: Check error type and map to appropriate error message
    # TODO 2: Handle ConnectionRefusedError, ConnectionResetError, timeout
    # TODO 3: Handle OSError with specific errno values (EPIPE, ECONNRESET, etc.)
    # TODO 4: Provide specific guidance based on error type
    # TODO 5: Return RPCTransportError with original error context
    pass
```

Serialization Error Handling

```python
"""
Message serialization with comprehensive error handling and validation.
"""

import struct
from typing import Dict, Any

MAX_MESSAGE_SIZE = 1024 * 1024  # 1MB message limit

def serialize_message(message: Dict[str, Any]) -> bytes:
    """Serialize message to JSON with error handling."""
    try:
        # TODO 1: Validate message structure has required fields
        # TODO 2: Check for circular references in message data
        # TODO 3: Serialize to JSON string with ensure_ascii=False
        # TODO 4: Encode JSON string to UTF-8 bytes
        # TODO 5: Check message size against MAX_MESSAGE_SIZE limit
        # TODO 6: Return serialized bytes
        pass
    except (TypeError, ValueError) as e:
        # TODO 7: Analyze error and provide specific guidance
        # TODO 8: Check if error is due to non-serializable objects
        # TODO 9: Raise RPCProtocolError with diagnostic information
        pass
    except UnicodeEncodeError as e:
        # TODO 10: Handle Unicode encoding issues
        # TODO 11: Provide character position and encoding details
```

```
        pass

def deserialize_message(data: bytes) -> Dict[str, Any]:
    """Deserialize JSON message with comprehensive error handling."""
    try:
        # TODO 1: Validate input data is not empty
        # TODO 2: Decode bytes to UTF-8 string with error handling
        # TODO 3: Parse JSON string to Python dict
        # TODO 4: Validate message has required JSON-RPC fields
        # TODO 5: Return validated message dict
        pass
    except UnicodeDecodeError as e:
        # TODO 6: Handle encoding issues with position information
        # TODO 7: Suggest UTF-8 encoding and provide sample bytes
        pass
    except json.JSONDecodeError as e:
        # TODO 8: Handle JSON parsing errors with position details
        # TODO 9: Provide context around error position for debugging
        # TODO 10: Suggest common JSON syntax fixes
        pass
    except (KeyError, TypeError) as e:
        # TODO 11: Handle schema validation errors
        # TODO 12: Identify missing or wrong-type fields
        pass

def validate_request_message(message: Dict[str, Any]) -> None:
    """Validate JSON-RPC request message format."""
    # TODO 1: Check required fields: jsonrpc, method, id
```

```
    # TODO 2: Validate jsonrpc version is "2.0"
    # TODO 3: Validate method is string and not empty
    # TODO 4: Validate id is string, number, or null
    # TODO 5: Validate params is array or object if present
    # TODO 6: Raise RPCProtocolError with specific field issues
    pass

def validate_response_message(message: Dict[str, Any]) -> None:
    """Validate JSON-RPC response message format."""
    # TODO 1: Check required fields: jsonrpc, id
    # TODO 2: Validate exactly one of 'result' or 'error' is present
    # TODO 3: Validate error object has code, message, and optional data
    # TODO 4: Validate error code is integer and message is string
    # TODO 5: Raise RPCProtocolError with validation details
    pass
```

Error Recovery and Retry Logic

```python
"""
Configurable retry logic with exponential backoff for transient failures.
"""

import random
import time
from typing import Callable, Any, Optional, List, Type
from dataclasses import dataclass

@dataclass
class RetryConfig:
    """Configuration for retry behavior."""
    max_attempts: int = 3
    base_delay: float = 0.1  # 100ms
    max_delay: float = 5.0   # 5 seconds
    backoff_multiplier: float = 2.0
    jitter: bool = True
    retryable_errors: List[Type[Exception]] = None

    def __post_init__(self):
        if self.retryable_errors is None:
            self.retryable_errors = [RPCTransportError, RPCTimeoutError]

class RetryHandler:
    """Implements exponential backoff retry logic for RPC operations."""

    def __init__(self, config: RetryConfig):
```

```python
        self.config = config
        self._attempt_lock = Lock()

    def execute_with_retry(self, operation: Callable[[], Any],
                           operation_name: str = "rpc_call") -> Any:
        """Execute operation with retry logic."""
        last_error = None

        for attempt in range(1, self.config.max_attempts + 1):
            try:
                # TODO 1: Execute the operation and return result if successful
                # TODO 2: Log retry attempts for debugging (not first attempt)
                pass
            except Exception as error:
                # TODO 3: Check if error type is retryable using isinstance()
                # TODO 4: If not retryable or last attempt, re-raise immediately
                # TODO 5: Calculate delay for this attempt using exponential backoff
                # TODO 6: Add jitter if configured (random factor 0.5-1.5x)
                # TODO 7: Sleep for calculated delay before next attempt
                # TODO 8: Store error for final re-raise if all attempts fail
                pass

        # TODO 9: All attempts failed - re-raise the last error with retry context
        pass

    def calculate_delay(self, attempt: int) -> float:
        """Calculate delay for retry attempt using exponential backoff."""
```

```
        # TODO 1: Calculate base delay: base_delay * (backoff_multiplier ^ (attempt-1))
        # TODO 2: Apply maximum delay cap
        # TODO 3: Add jitter if enabled: multiply by random factor 0.5 to 1.5
        # TODO 4: Return final delay value
        pass

    def is_retryable_error(self, error: Exception) -> bool:
        """Check if error type should trigger retry logic."""
        # TODO 1: Check if error is instance of any retryable error type
        # TODO 2: For transport errors, check specific conditions (not auth failures)
        # TODO 3: For timeout errors, check if timeout was reasonable
        # TODO 4: Return True if retry is appropriate, False otherwise
        pass
```

Milestone Checkpoints

After Milestone 1 (Message Protocol):

```bash
# Test serialization error handling
python -m pytest tests/test_protocol_errors.py -v

# Expected behavior:
# - Invalid JSON messages raise RPCProtocolError with parse details
# - Oversized messages are rejected with size information
# - Missing required fields provide specific field names in errors
# - Unicode encoding issues include character position details
```

After Milestone 2 (Server Implementation):

```bash
# Test server error handling
python -m pytest tests/test_server_errors.py -v

# Expected behavior:
# - Method not found returns structured error response
# - Method execution errors are caught and returned as RPC errors
# - Network disconnections are detected and connections cleaned up
# - Invalid requests receive appropriate error codes
```

After Milestone 3 (Client Implementation):

```bash
# Test end-to-end error handling
python -m pytest tests/test_client_errors.py -v

# Expected behavior:
# - Connection failures trigger retry logic with exponential backoff
# - Timeout errors include timing information and don't hang
# - Serialization errors prevent request sending with clear diagnostics
# - Server errors are converted to appropriate client exceptions
```

Debugging Tips

Symptom	Likely Cause	Diagnosis	Fix
Client hangs indefinitely	Network timeout not properly handled	Check if `socket.settimeout()` is set	Implement proper timeout in `recv_all`
"Connection reset by peer" errors	Server closing connections abruptly	Monitor server logs for exceptions during request processing	Add exception handling in server request loop
JSON parse errors with valid JSON	Unicode encoding mismatch	Check raw bytes in error data for non-UTF8	Ensure consistent UTF-8 encoding on both ends
Intermittent "Broken pipe" errors	Client disconnecting during response send	Add connection health check before sending	Gracefully handle client disconnections
Retry logic not triggering	Error not classified as retryable	Check error type inheritance and retry configuration	Verify `RPCTransportError` inheritance chain
Memory usage growing over time	Error objects retaining large message data	Check error_data size in exception objects	Truncate large data in error serialization

Testing Strategy

> **Milestone(s):** All milestones - comprehensive testing approach for message protocol (Milestone 1), server implementation (Milestone 2), and client implementation (Milestone 3)

Mental Model: The Quality Control Factory

Think of testing your RPC framework like a quality control factory with multiple inspection stations. Each milestone represents a production line stage where components must pass specific quality checks before moving to the next station. At the message protocol station, inspectors verify that every envelope (message) has the correct addressing and contents. At the server station, they test that the factory machinery (method registry and request dispatch) processes orders correctly. At the client station, they verify that the shipping department (proxy objects and connection management) delivers results reliably. Finally, the integration station tests the entire production line end-to-end, simulating real-world customer scenarios to ensure the whole system works harmoniously.

The key insight is that testing an RPC framework requires both **component isolation** and **system integration**. Component tests verify individual pieces work correctly in controlled conditions, while integration tests validate that components collaborate properly under realistic network conditions and error scenarios.

Milestone Checkpoints

Each milestone introduces specific functionality that must be thoroughly verified before proceeding to the next stage. The checkpoint approach ensures that foundational components work correctly before building dependent layers, preventing cascading failures that become difficult to debug in later milestones.

Milestone 1: Message Protocol Checkpoints

The message protocol checkpoint verifies that request/response serialization works correctly and handles all specified parameter types and error conditions.

Test Category	What to Verify	Expected Behavior	Failure Signals
Request Serialization	`create_request_message` produces valid JSON-RPC	Method name, params, and ID present in serialized bytes	Missing fields, invalid JSON, incorrect format
Response Serialization	`create_response_message` and `create_error_message` work	Result or error paired with matching request ID	ID mismatch, malformed error structure
Message Deserialization	`deserialize_message` parses valid JSON correctly	Returns dict with expected fields and types	JSON decode errors, missing required fields
Parameter Type Handling	Strings, numbers, objects, arrays serialize correctly	Round-trip preserves original values and types	Type conversion errors, precision loss
Error Format Validation	Error messages follow JSON-RPC specification	Error code, message, and optional data fields present	Missing error codes, inconsistent format

Verification Commands:

```bash
python -m pytest tests/test_protocol.py -v

python tests/manual_protocol_test.py
```

Manual Testing Approach: Create test scripts that serialize various message types and verify the output manually. For example, create a request with complex nested parameters (lists containing objects with string

and numeric fields) and confirm that deserialization produces identical data structures.

> **Key Insight:** Protocol testing must validate both the happy path and edge cases. Test with empty parameter lists, null values, very large messages approaching `MAX_MESSAGE_SIZE`, and malformed JSON to ensure robust error handling.

Signs of Success:

- All message types serialize to valid JSON that matches the JSON-RPC specification format
- Deserialization of serialized messages produces identical data structures (round-trip integrity)
- Error messages contain all required fields and use standard error codes from `ErrorCode` constants
- Large messages (approaching 1MB) serialize without truncation or memory issues

Troubleshooting Common Issues:

⚠️ **Pitfall: JSON Serialization Type Errors** Symptom: TypeError when calling `serialize_message` with certain parameter types Cause: Python objects that aren't JSON-serializable (like custom classes or datetime objects) Fix: Implement custom JSON encoder or validate parameter types before serialization

⚠️ **Pitfall: Unicode Encoding Problems** Symptom: UnicodeDecodeError when deserializing messages containing non-ASCII characters Cause: Inconsistent UTF-8 encoding/decoding between serialization and deserialization Fix: Explicitly specify UTF-8 encoding in all string-to-bytes conversions

Milestone 2: Server Implementation Checkpoints

The server checkpoint validates that method registration, request dispatch, and error handling work correctly for concurrent client connections.

Test Category	What to Verify	Expected Behavior	Failure Signals
Method Registry	`register_method` stores callable functions correctly	Methods accessible by name, callable with correct signatures	KeyError when calling registered methods, signature mismatches
TCP Server Startup	`start_server` listens on specified host and port	Server accepts incoming connections, binds to correct endpoint	Address already in use, permission denied, socket errors
Request Parsing	`handle_connection` parses incoming JSON-RPC requests	Valid requests deserialize correctly, invalid requests return parse errors	Silent failures, missing error responses
Method Dispatch	`process_request` invokes registered methods with parameters	Method execution with provided arguments returns expected results	Method not found errors, parameter count mismatches
Error Response Format	Server returns properly formatted JSON-RPC errors	Error responses contain matching request ID and standard error codes	Missing request IDs, invalid error format
Concurrent Handling	Multiple clients can connect and make requests simultaneously	Each request processed independently without interference	Request mixing, shared state corruption

Verification Commands:

```bash
python -m pytest tests/test_server.py -v

python tests/server_integration_test.py

python -c "import rpc_server; server = rpc_server.RPCServer('localhost', 8080);
server.start_server()"
```

Manual Testing Approach: Start the server in one terminal, then use telnet or a simple client script to send raw JSON-RPC requests. Verify that the server responds with correctly formatted responses and handles malformed requests gracefully.

Test Method Registry:

```python
# Register a simple test method
def add(a, b):
    return a + b

server.register_method("add", add)

# Verify method is callable through registry
result = server.registry.methods["add"](5, 3)
assert result == 8
```

Test Concurrent Connections: Use multiple client connections simultaneously to verify that the thread-per-connection model handles concurrent requests without blocking or corrupting shared state.

> **Key Insight:** Server testing requires verifying both functional correctness and concurrent behavior. The `MethodRegistry` must be thread-safe, and each client connection should operate independently without affecting other concurrent requests.

Signs of Success:

- Server starts successfully and listens on the specified port without binding errors
- Method registry stores and retrieves callable functions correctly with thread safety
- Valid JSON-RPC requests execute registered methods and return properly formatted responses
- Invalid requests (malformed JSON, method not found, parameter errors) return standard JSON-RPC error responses
- Multiple concurrent clients can connect and make requests simultaneously without interference
- Server handles client disconnections gracefully without crashing or resource leaks

Troubleshooting Common Issues:

⚠️ **Pitfall: Socket Address Reuse Problems** Symptom: "Address already in use" error when restarting server quickly Cause: Operating system holding socket in TIME_WAIT state after server shutdown Fix: Set `SO_REUSEADDR` socket option and implement proper server shutdown cleanup

⚠️ **Pitfall: Thread Resource Exhaustion** Symptom: Server stops accepting new connections after handling many clients Cause: Creating new threads without proper cleanup when connections end Fix: Use thread cleanup or implement connection pooling to limit concurrent threads

⚠️ **Pitfall: Shared State Corruption** Symptom: Method calls return unexpected results when multiple clients connect Cause: Registered methods accessing shared global variables without synchronization Fix: Use thread-local storage or proper locking for shared state in registered methods

Milestone 3: Client Implementation Checkpoints

The client checkpoint validates that method proxying, connection management, and timeout handling work correctly under various network conditions.

Test Category	What to Verify	Expected Behavior	Failure Signals
Server Connection	`call` establishes TCP connection to server	Client connects to specified host and port successfully	Connection refused, timeout errors, network unreachable
Method Proxying	`RPCProxy.__getattr__` creates callable methods	Proxy objects allow natural method call syntax	AttributeError, method calls don't send requests
Request Transmission	`call` sends properly formatted JSON-RPC requests	Requests include method name, parameters, and unique IDs	Malformed requests, missing request IDs
Response Correlation	Client matches responses to correct pending requests	Responses return to the originating method call	Request/response ID mismatches, blocking forever
Timeout Handling	`call` respects timeout parameter and raises `RPCTimeoutError`	Calls abort after specified timeout duration	Hanging indefinitely, premature timeouts
Connection Reuse	Multiple calls use the same TCP connection efficiently	Single connection handles multiple sequential requests	New connection for every request, connection leaks
Error Propagation	Server errors propagate to client as appropriate exceptions	JSON-RPC errors raise `RPCMethodError` with server details	Silent failures, generic exceptions without context

Verification Commands:

```bash
python -m pytest tests/test_client.py -v

python tests/client_integration_test.py

python tests/client_server_integration.py
```

Manual Testing Approach: Start a test server with known methods, then create client instances and verify that proxy method calls work as expected. Test timeout behavior by calling methods that intentionally delay longer than the timeout setting.

Test Method Proxying:

```python
# Create client and proxy
client = RPCClient("localhost", 8080)
proxy = RPCProxy(client)

# Call methods using natural syntax
result = proxy.add(10, 20)
assert result == 30

# Verify timeout behavior
try:
    proxy.slow_method(timeout=1.0)  # Method takes 5 seconds
    assert False, "Should have timed out"
except RPCTimeoutError as e:
    assert e.timeout_seconds == 1.0
```

Test Connection Management:

```python
client = RPCClient("localhost", 8080)

# Verify connection reuse
result1 = client.call("method1", [])
result2 = client.call("method2", [])
# Both calls should use same socket connection

# Verify connection cleanup
client._disconnect()
assert client._connected == False
```

> **Key Insight:** Client testing must validate both the programming interface (method proxying feels natural) and the network behavior (connection reuse, timeout handling, error propagation). The proxy pattern should be transparent to the user while handling all RPC complexity internally.

Signs of Success:

- Client connects to server successfully and establishes stable TCP connection
- Proxy objects allow method calls with natural Python syntax (proxy.method_name(args))
- Method calls send correctly formatted JSON-RPC requests and receive matching responses
- Request IDs are generated uniquely and responses correlate to the correct pending requests
- Timeout handling works reliably, aborting slow requests after the specified duration
- Connection reuse functions correctly, using a single TCP connection for multiple sequential calls
- Server errors propagate as appropriate client-side exceptions with diagnostic information

Troubleshooting Common Issues:

⚠️ **Pitfall: Request/Response ID Correlation Errors** Symptom: Method calls return results from different requests or hang indefinitely Cause: Request ID generation not unique or response matching logic incorrect Fix: Use UUID for request IDs and verify response correlation logic handles concurrent requests

⚠️ **Pitfall: Connection State Corruption** Symptom: Client errors after first successful call or intermittent connection failures Cause: Socket state not properly managed between requests Fix: Implement proper connection state tracking and recovery logic in `_ensure_connected`

⚠️ **Pitfall: Timeout Not Interrupting Socket Operations** Symptom: Timeout parameter ignored, client hangs on unresponsive server Cause: Socket operations (send/recv) not respecting timeout settings Fix: Set socket timeout before each operation and handle socket timeout exceptions properly

Integration Test Scenarios

Integration testing verifies that the complete RPC framework functions correctly under realistic conditions, testing the interactions between all components and handling real-world edge cases that don't appear in unit tests.

End-to-End RPC Call Scenarios

These scenarios test the complete flow from client method invocation through network transport to server execution and response handling.

Scenario	Setup	Expected Behavior	Validation Points
Basic Method Call	Server with `add(a, b)` method, client calls `proxy.add(5, 3)`	Returns 8, request/response IDs match	Message format, method execution, response correlation
Complex Parameter Types	Server method accepting nested objects and arrays	Parameters preserve types and structure through serialization	JSON round-trip integrity, type preservation
Method Not Found	Client calls `proxy.nonexistent_method()` on server without that method	Raises `RPCMethodError` with `METHOD_NOT_FOUND` error code	Error propagation, standard error codes
Server Method Exception	Registered method raises Python exception during execution	Client receives `RPCMethodError` with exception details	Error handling, exception message preservation
Large Message Transfer	Method with parameters approaching `MAX_MESSAGE_SIZE`	Successful transfer without truncation or memory errors	Message framing, size limits, memory efficiency
Multiple Sequential Calls	Client makes 10 method calls using same connection	All calls succeed with connection reuse	Connection management, request correlation
Concurrent Client Connections	5 clients simultaneously calling server methods	All requests processed correctly without interference	Thread safety, concurrent request handling

Basic End-to-End Test Implementation:

The following test scenario validates that a complete RPC call works correctly from client method invocation to server response:

1. Start `RPCServer` with a registered `add` method that returns the sum of two parameters
2. Create `RPCClient` and `RPCProxy` connecting to the server
3. Call `proxy.add(10, 20)` using natural method syntax
4. Verify that the call returns 30 and the request/response correlation worked correctly
5. Check that the TCP connection remains open for subsequent calls
6. Validate that the JSON-RPC message format was used correctly throughout the exchange

Complex Parameter Handling Test:

This scenario ensures that complex data structures survive the serialization round-trip correctly:

1. Register server method `process_data(data)` that returns a modified version of the input data structure
2. Create test data containing nested objects, arrays, strings, numbers, and null values
3. Call `proxy.process_data(complex_data)` and verify the response preserves all data types and structure
4. Test edge cases like empty arrays, deeply nested objects, and Unicode strings
5. Verify that the serialization process doesn't introduce type conversion errors or data loss

Error Handling Integration Test:

This scenario validates that errors propagate correctly from server to client with proper diagnostic information:

1. Register server method that intentionally raises various types of exceptions
2. Call methods that trigger different error conditions: method not found, parameter validation errors, execution exceptions
3. Verify that each error type results in the appropriate client-side exception with accurate error codes and messages
4. Test error handling during network failures: server disconnection, timeout scenarios, malformed responses
5. Ensure that error conditions don't corrupt the connection state for subsequent requests

Network Failure and Recovery Scenarios

These scenarios test how the RPC framework handles various network failure modes and recovery situations.

Failure Mode	Trigger Condition	Expected Behavior	Recovery Validation
Server Disconnect During Call	Server process killed while client waiting for response	Client raises `RPCTransportError` with connection details	Subsequent calls detect disconnection and attempt reconnection
Connection Timeout	Server accepts connection but doesn't respond to requests	Client raises `RPCTimeoutError` after specified timeout	Connection marked as failed, new connection established for next call
Partial Message Transmission	Network interruption during large message transfer	Client detects incomplete message and raises transport error	Connection reset, retry logic handles partial transfers
Server Overload	Server cannot accept new connections due to resource limits	Client receives connection refused error with retry suggestion	Client implements exponential backoff for connection retries
Intermittent Network Issues	Connection drops randomly during request/response cycle	Framework detects connection failures and recovers gracefully	Automatic reconnection without user intervention

Connection Failure Recovery Test:

This scenario validates that the client handles server disconnection gracefully and recovers for subsequent requests:

1. Start server and establish client connection with successful initial request
2. Kill server process while client has a pending request waiting for response
3. Verify that client detects connection failure and raises `RPCTransportError` with appropriate error details
4. Restart server and make another client request
5. Confirm that client establishes new connection automatically and request succeeds
6. Validate that connection state tracking correctly reflects the disconnection and reconnection

Timeout Handling Integration Test:

This scenario ensures that timeout handling works correctly under realistic network conditions:

1. Configure server method that intentionally delays longer than client timeout setting
2. Make client request with specific timeout value shorter than method execution time
3. Verify that client raises `RPCTimeoutError` with accurate timeout duration and elapsed time
4. Ensure that timed-out connection is properly cleaned up and doesn't interfere with subsequent requests
5. Test timeout handling with various timeout values to ensure accuracy across different durations

Stress and Performance Scenarios

These scenarios validate that the RPC framework maintains correctness under load and performs adequately for typical use cases.

Load Scenario	Configuration	Success Criteria	Performance Expectations
High Request Rate	Single client making 1000 sequential requests	All requests complete successfully with correct responses	Average latency under 10ms for simple methods
Multiple Concurrent Clients	20 clients each making 100 requests simultaneously	No request mixing, all responses correlate correctly	Server handles concurrent load without blocking
Large Message Stress	Requests with payloads near maximum message size	All large messages transfer completely without corruption	Memory usage remains bounded, no memory leaks
Long-Running Server	Server handling requests continuously for 1 hour	No resource leaks, consistent response times	Memory and file descriptor usage remains stable
Connection Pool Stress	Rapid connection/disconnection cycles	No socket resource exhaustion, proper cleanup	Operating system socket limits not exceeded

Concurrent Client Load Test:

This scenario validates that the server's thread-per-connection model handles concurrent requests correctly:

1. Create 10 client instances, each running in a separate thread
2. Each client makes 50 requests to different server methods simultaneously
3. Verify that all 500 total requests complete successfully with correct results
4. Check that request/response correlation works correctly across concurrent clients
5. Monitor server resource usage to ensure thread creation and cleanup work properly
6. Validate that no shared state corruption occurs between concurrent requests

Memory and Resource Stress Test:

This scenario ensures that the framework doesn't leak resources under sustained load:

1. Run client making continuous requests for 30 minutes with periodic garbage collection
2. Monitor memory usage of both client and server processes throughout the test
3. Verify that socket file descriptors are properly closed after use
4. Check that thread resources are cleaned up when client connections end
5. Ensure that message buffers and connection state don't accumulate over time

> **Key Insight:** Integration testing reveals issues that don't appear in unit tests, particularly around resource management, concurrent access, and network failure handling. These scenarios test the framework's behavior under realistic conditions rather than ideal laboratory settings.

Error Recovery and Resilience Scenarios

These scenarios test how well the framework handles and recovers from various error conditions that occur in production environments.

Error Condition	Setup	Expected Recovery	Validation
Serialization Failure	Method parameters that can't be JSON-serialized	Client raises appropriate error before sending request	No malformed data sent to server
Server Method Crash	Registered method with unhandled exception	Server sends error response, remains available for other requests	Server doesn't crash, other clients unaffected
Network Partition	Temporary network disconnection during active session	Client detects failure, retries when network recovers	Automatic recovery without manual intervention
Server Resource Exhaustion	Server running out of memory or file descriptors	Graceful degradation with appropriate error responses	Server remains stable, doesn't crash completely
Message Size Limit Exceeded	Request larger than `MAX_MESSAGE_SIZE`	Client or server rejects message before processing	Clear error message about size limits

Serialization Error Handling Test:

This scenario validates that serialization errors are caught and handled appropriately:

1. Attempt to call server method with parameters containing non-serializable objects (e.g., file handles, custom classes)
2. Verify that client detects serialization failure before sending request to server
3. Ensure that serialization error raises `RPCProtocolError` with diagnostic information about the problematic data
4. Test that client connection remains usable after serialization error for subsequent valid requests

Server Resilience Test:

This scenario ensures that server errors don't affect other clients or crash the server:

1. Register server method that raises various types of exceptions (ValueError, KeyError, custom exceptions)
2. Create multiple client connections, some calling problematic methods and others calling normal methods

3. Verify that method exceptions are caught, converted to JSON-RPC error responses, and sent to the appropriate client
4. Confirm that server continues running and handling requests from other clients normally
5. Test that exception details are preserved in error responses for debugging purposes

Implementation Guidance

This section provides practical testing infrastructure and verification procedures to help implement comprehensive testing for the RPC framework.

Testing Technology Recommendations

Component	Simple Option	Advanced Option
Unit Testing	Python unittest (built-in)	pytest with fixtures and parameterized tests
Integration Testing	Manual test scripts	pytest with test containers
Network Testing	Local loopback connections	Docker containers with network simulation
Load Testing	Threading module with simple loops	locust or concurrent.futures for realistic load
Mocking	unittest.mock (built-in)	pytest-mock with advanced stubbing

Test File Structure

Organize test files to match the component structure and separate unit tests from integration tests:

```
rpc_framework/
  src/
    rpc/
      protocol.py          ← Message protocol implementation
      server.py            ← RPC server implementation
      client.py            ← RPC client implementation
      errors.py            ← Error classes
  tests/
    unit/
      test_protocol.py     ← Protocol serialization tests
      test_server.py       ← Server component tests
      test_client.py       ← Client component tests
      test_errors.py       ← Error handling tests
    integration/
      test_end_to_end.py   ← Full RPC call scenarios
      test_failure_modes.py ← Network failure tests
      test_load.py         ← Performance and stress tests
    helpers/
      test_server.py       ← Test server utilities
      test_client.py       ← Test client utilities
      fixtures.py          ← Common test data
```

Unit Test Infrastructure Starter Code

Complete testing utilities that provide the foundation for all milestone testing:

PYTHON

```python
# tests/helpers/test_server.py

import threading

import time

import socket

from typing import Optional, Callable, Dict, Any

from src.rpc.server import RPCServer


class TestServerHelper:

    """Helper class for managing test servers in unit and integration tests."""



    def __init__(self, host: str = "localhost", port: int = 0):

        self.host = host

        self.port = port

        self.server: Optional[RPCServer] = None

        self.server_thread: Optional[threading.Thread] = None

        self._methods: Dict[str, Callable] = {}



    def register_method(self, name: str, method: Callable):

        """Register a method for the test server."""

        self._methods[name] = method



    def start(self) -> int:

        """Start the test server and return the actual port number."""

        # TODO 1: Create RPCServer instance with host and port

        # TODO 2: Register all methods from self._methods

        # TODO 3: Start server in background thread

        # TODO 4: Wait for server to start listening
```

```
        # TODO 5: Return actual port number (important when port=0)
        pass

    def stop(self):
        """Stop the test server and clean up resources."""
        # TODO 1: Signal server to stop
        # TODO 2: Wait for server thread to complete
        # TODO 3: Clean up server resources
        pass

    def is_running(self) -> bool:
        """Check if the test server is currently running."""
        # TODO: Check server state and thread status
        pass

# tests/helpers/test_client.py
import time
from typing import Any, List, Dict
from src.rpc.client import RPCClient
from src.rpc.errors import RPCError

class TestClientHelper:
    """Helper for testing client behavior and collecting metrics."""

    def __init__(self, host: str, port: int):
        self.host = host
        self.port = port
        self.client = RPCClient(host, port)
```

```
        self.call_history: List[Dict[str, Any]] = []



    def call_with_timing(self, method_name: str, params: List[Any], timeout: float = 5.0) -
> Dict[str, Any]:

        """Make an RPC call and record timing and result information."""

        start_time = time.time()

        error = None

        result = None



        try:

            result = self.client.call(method_name, *params, timeout=timeout)

        except Exception as e:

            error = e



        end_time = time.time()



        call_record = {

            'method': method_name,

            'params': params,

            'start_time': start_time,

            'end_time': end_time,

            'duration': end_time - start_time,

            'result': result,

            'error': error,

            'success': error is None

        }
```

```
        self.call_history.append(call_record)


        if error:
            raise error
        return result


    def get_average_latency(self) -> float:
        """Calculate average latency for successful calls."""
        # TODO: Calculate average duration from call_history for successful calls
        pass


    def get_error_rate(self) -> float:
        """Calculate percentage of calls that resulted in errors."""
        # TODO: Calculate ratio of failed calls to total calls
        pass


    def reset_metrics(self):
        """Clear call history and reset metrics."""
        self.call_history.clear()
```

Milestone Checkpoint Implementation

Complete test cases that verify each milestone's core functionality:

PYTHON

```python
# tests/unit/test_protocol.py

import pytest

import json

from src.rpc.protocol import create_request_message, create_response_message,
serialize_message, deserialize_message

from src.rpc.errors import ErrorCode


class TestProtocolMilestone:

    """Tests for Milestone 1 - Message Protocol implementation."""



    def test_create_request_message_basic(self):

        """Verify request message contains required fields."""

        # TODO 1: Create request with method name, parameters, and request ID

        # TODO 2: Assert message contains 'jsonrpc', 'method', 'params', 'id' fields

        # TODO 3: Verify field values match input parameters

        pass



    def test_request_response_serialization_roundtrip(self):

        """Verify messages can be serialized and deserialized without data loss."""

        # TODO 1: Create request with complex nested parameters

        # TODO 2: Serialize to bytes using serialize_message

        # TODO 3: Deserialize bytes back to dict using deserialize_message

        # TODO 4: Assert original and deserialized data are identical

        pass



    @pytest.mark.parametrize("params,expected_type", [

        ([1, 2, 3], list),

        ({"key": "value"}, dict),
```

```
        ("test string", str),
        (42.5, float),
        (None, type(None))
    ])
    def test_parameter_type_preservation(self, params, expected_type):
        """Verify different parameter types are preserved through serialization."""
        # TODO 1: Create request message with given params
        # TODO 2: Serialize and deserialize the message
        # TODO 3: Assert parameter types are preserved exactly
        pass

    def test_error_message_format(self):
        """Verify error messages follow JSON-RPC specification."""
        # TODO 1: Create error message with code, message, and data
        # TODO 2: Assert error structure contains required fields
        # TODO 3: Verify error codes match ErrorCode constants
        pass

# tests/integration/test_end_to_end.py
import pytest
import threading
import time
from tests.helpers.test_server import TestServerHelper
from tests.helpers.test_client import TestClientHelper

class TestEndToEndScenarios:
    """Integration tests for complete RPC framework functionality."""
```

```
    def setup_method(self):
        """Set up test server with sample methods."""
        self.server_helper = TestServerHelper()

        # Register test methods
        self.server_helper.register_method("add", lambda a, b: a + b)
        self.server_helper.register_method("echo", lambda msg: msg)
        self.server_helper.register_method("slow_method", lambda: time.sleep(2))

        # Start server and get port
        self.port = self.server_helper.start()
        self.client_helper = TestClientHelper("localhost", self.port)

    def teardown_method(self):
        """Clean up test server and client."""
        self.server_helper.stop()

    def test_basic_method_call(self):
        """Verify basic RPC call works end-to-end."""
        # TODO 1: Make simple method call using client helper
        # TODO 2: Verify correct result is returned
        # TODO 3: Check that call completed in reasonable time
        pass

    def test_multiple_sequential_calls(self):
        """Verify connection reuse for multiple calls."""
        # TODO 1: Make 10 sequential method calls
```

```
    # TODO 2: Verify all calls succeed with correct results
    # TODO 3: Check that connection was reused (timing analysis)
    pass


def test_timeout_handling(self):
    """Verify timeout behavior for slow methods."""
    # TODO 1: Call slow_method with short timeout
    # TODO 2: Assert RPCTimeoutError is raised
    # TODO 3: Verify timeout duration is respected
    pass


def test_concurrent_clients(self):
    """Verify server handles multiple concurrent clients."""
    # TODO 1: Create multiple client threads
    # TODO 2: Each client makes multiple requests simultaneously
    # TODO 3: Verify all requests complete successfully
    # TODO 4: Assert no request mixing or correlation errors
    pass
```

Load and Stress Testing Infrastructure

```python
# tests/integration/test_load.py

import pytest

import threading

import time

import statistics

from concurrent.futures import ThreadPoolExecutor, as_completed

from tests.helpers.test_server import TestServerHelper

from tests.helpers.test_client import TestClientHelper


class TestLoadScenarios:

    """Stress and performance tests for RPC framework."""


    def setup_method(self):

        self.server_helper = TestServerHelper()


        # Register performance test methods

        self.server_helper.register_method("fast_add", lambda a, b: a + b)

        self.server_helper.register_method("cpu_intensive", self._cpu_work)

        self.server_helper.register_method("large_response", self._generate_large_data)


        self.port = self.server_helper.start()


    def _cpu_work(self, iterations: int = 1000):

        """CPU-intensive method for performance testing."""

        # TODO: Implement method that does meaningful CPU work

        pass
```

```python
def _generate_large_data(self, size_kb: int = 100):
    """Generate large response data for testing message size limits."""
    # TODO: Generate dictionary with specified size in KB
    pass


def test_high_request_rate(self):
    """Test server performance under high request rate."""
    client = TestClientHelper("localhost", self.port)

    # TODO 1: Make 1000 requests as quickly as possible
    # TODO 2: Record latency for each request
    # TODO 3: Assert average latency is under threshold
    # TODO 4: Verify no requests failed
    pass


def test_concurrent_client_load(self):
    """Test server with multiple concurrent clients."""
    num_clients = 20
    requests_per_client = 50

    def client_worker(client_id):
        # TODO 1: Create client helper for this thread
        # TODO 2: Make specified number of requests
        # TODO 3: Return performance metrics
        pass
```

```
    # TODO 1: Create thread pool with client workers
    # TODO 2: Execute all workers concurrently
    # TODO 3: Collect and analyze results
    # TODO 4: Assert performance requirements are met
    pass


def test_memory_stability(self):
    """Verify no memory leaks during sustained operation."""
    import psutil
    import os

    process = psutil.Process(os.getpid())
    initial_memory = process.memory_info().rss

    # TODO 1: Run continuous requests for 5 minutes
    # TODO 2: Monitor memory usage throughout test
    # TODO 3: Assert memory growth is bounded
    # TODO 4: Verify garbage collection works correctly
    pass
```

Debugging and Verification Tools

```python
# tests/helpers/debug_tools.py

import json

import socket

from typing import Dict, Any, List

from src.rpc.protocol import serialize_message, deserialize_message


class RPCDebugger:

    """Tools for debugging RPC framework issues."""



    @staticmethod

    def validate_message_format(message_dict: Dict[str, Any]) -> List[str]:

        """Validate JSON-RPC message format and return list of issues."""

        issues = []



        # TODO 1: Check for required JSON-RPC fields

        # TODO 2: Validate field types and values

        # TODO 3: Return list of format violations found

        pass



    @staticmethod

    def trace_rpc_call(host: str, port: int, method: str, params: List[Any]):

        """Make RPC call with detailed tracing of message flow."""

        print(f"=== RPC Call Trace: {method}({params}) ===")



        # TODO 1: Create and display request message

        # TODO 2: Show serialized bytes
```

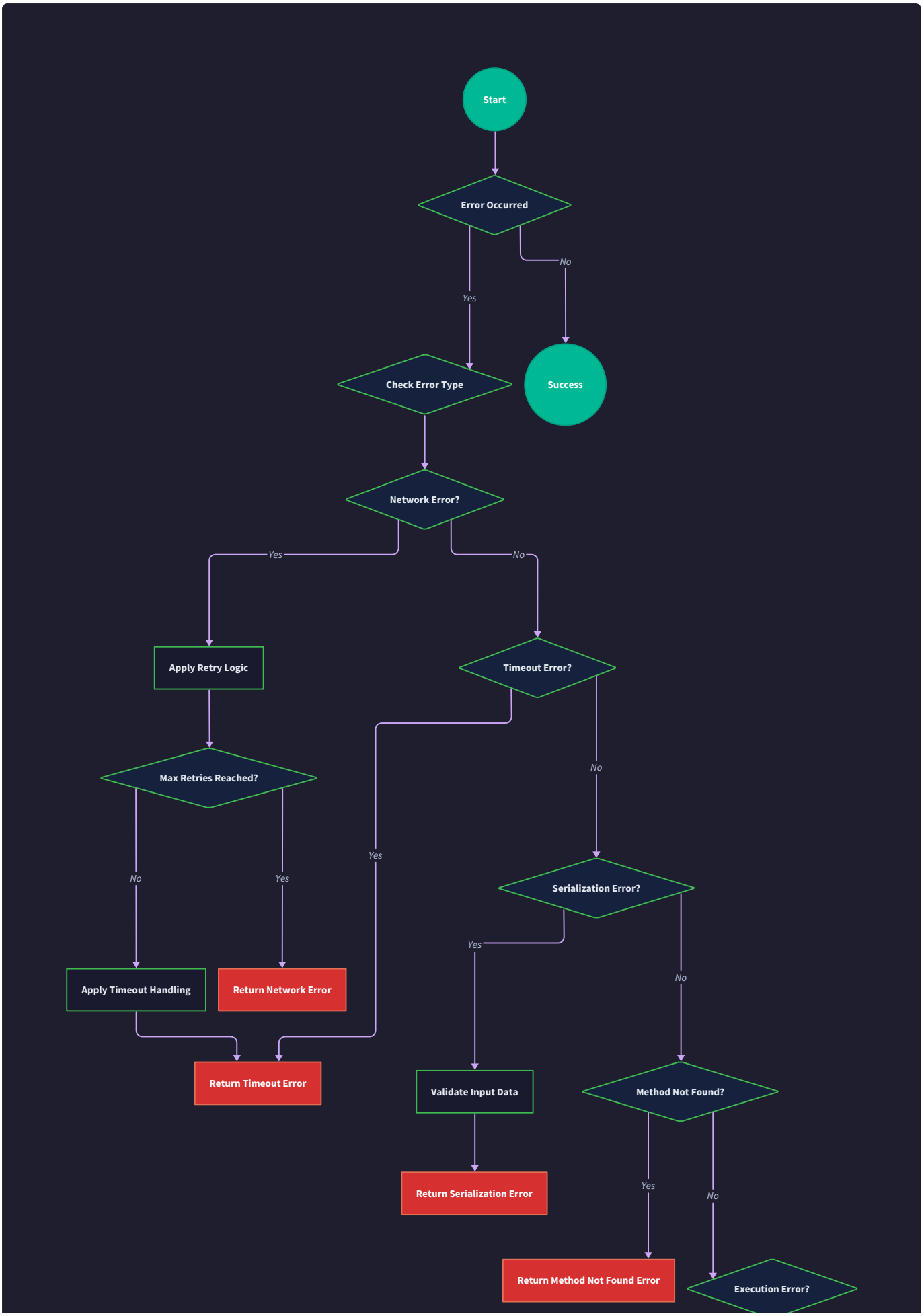
```
        # TODO 3: Connect to server and trace network communication
        # TODO 4: Display response and any errors
        pass

    @staticmethod
    def test_server_connectivity(host: str, port: int, timeout: float = 5.0) -> Dict[str, Any]:
        """Test basic connectivity to RPC server."""
        result = {
            'host': host,
            'port': port,
            'connected': False,
            'error': None,
            'latency_ms': None
        }

        # TODO 1: Attempt TCP connection
        # TODO 2: Measure connection latency
        # TODO 3: Test basic socket communication
        # TODO 4: Return diagnostic information
        pass
```

This comprehensive testing strategy ensures that each milestone is properly validated before proceeding to the next, while integration tests verify that all components work together correctly under realistic conditions. The provided infrastructure code gives learners a solid foundation for implementing their own test cases without getting bogged down in testing framework setup.

Debugging Guide

> **Milestone(s):** All milestones - debugging techniques apply throughout message protocol (Milestone 1), server implementation (Milestone 2), and client implementation (Milestone 3), and their integration

Building an RPC framework involves multiple layers of complexity - network communication, serialization, concurrency, and error propagation. When something goes wrong, the symptoms often manifest far from the root cause, making debugging challenging for developers new to distributed systems. This guide provides systematic approaches to identify, diagnose, and resolve the most common issues encountered when implementing our RPC framework.


Start
Error Occurred
No
Yes
Success
Check Error Type
Network Error?
Yes
No
Apply Retry Logic
Timeout Error?
Max Retries Reached?
No
Yes
No
Yes
Serialization Error?
Yes
No
Apply Timeout Handling
Return Network Error
Return Timeout Error
Validate Input Data
Method Not Found?
Return Serialization Error
Yes
No
Return Method Not Found Error
Execution Error?

Mental Model: The Detective's Investigation Process

Think of debugging RPC issues like a detective investigating a crime scene. You start with symptoms (the "crime") - perhaps a client call hangs indefinitely, or the server crashes with a cryptic error. Like a detective, you must gather evidence systematically, form hypotheses about what went wrong, and test each theory until you find the root cause. The key is following a structured process rather than randomly changing code and hoping for the best.

Just as a detective looks for patterns (multiple similar crimes suggest a serial perpetrator), RPC debugging benefits from recognizing common failure patterns. Network issues have different "fingerprints" than serialization problems, which look different from timing issues. Learning to read these patterns accelerates diagnosis.

The detective also knows that the most obvious suspect isn't always guilty. In RPC debugging, the error message you see might be a secondary effect. A "connection refused" error might actually be caused by the server crashing due to a serialization bug, not a networking problem. Always dig deeper than the surface symptoms.

Connection and Network Issues

Network problems are often the first roadblock when building RPC systems, because they involve multiple moving parts - sockets, TCP protocol behavior, firewalls, and operating system networking stacks. These issues can be particularly frustrating because they often work fine in development but fail in different environments.

> **Key Insight**: Network issues usually manifest as exceptions during connection establishment or data transfer, but the root cause might be configuration, timing, or resource exhaustion rather than actual network connectivity problems.

Common Network Failure Patterns

The following table catalogs the most frequent network-related issues and their diagnostic approaches:

Symptom	Likely Root Cause	Diagnostic Steps	Resolution Strategy
`ConnectionRefusedError` on client	Server not running or wrong port	`netstat -an | grep PORT`, `telnet HOST PORT`	Verify server startup, check port binding
Client hangs on `connect()`	Firewall blocking, wrong IP	`ping HOST`, `traceroute HOST`, check firewall rules	Network configuration, security groups
`BrokenPipeError` during send	Server closed connection unexpectedly	Check server logs for crashes, monitor server process	Fix server-side errors causing disconnection
`ConnectionResetError` mid-call	Server or network dropped connection	Network monitoring, server resource usage	Implement connection health checks, retry logic
Intermittent `TimeoutError`	Network latency spikes or server overload	Measure round-trip times, server CPU usage	Increase timeouts, add server capacity
`AddressAlreadyInUse` on server start	Previous server instance still bound to port	`lsof -i :PORT`, check for zombie processes	Clean shutdown, SO_REUSEADDR socket option

Socket State Debugging

Understanding TCP socket states helps diagnose connection issues. The `RPCClient` and `RPCServer` maintain socket connections that transition through predictable states. When debugging, check the actual socket state against the expected state:

Expected Client State	Socket State	Diagnostic Command	Common Mismatches
Disconnected	No socket entry	`netstat -an`	Leaked sockets from previous connections
Connecting	`SYN_SENT`	`ss -t state syn-sent`	Stuck in connecting due to firewall
Connected	`ESTABLISHED`	`ss -t state established`	Socket shows established but RPC calls fail
Disconnecting	`FIN_WAIT1`, `FIN_WAIT2`	`ss -t state fin-wait-1`	Clean shutdown taking too long

Server Connection Handling

The `RPCServer` accepts incoming connections and spawns threads to handle each client. Common issues arise from resource limits, thread management, and proper socket cleanup:

Connection Accept Loop Issues:

1. **Server stops accepting new connections**: Usually caused by the accept loop exiting due to an unhandled exception. Check server logs for stack traces around the last successful connection.
2. **`accept()` returns immediately with `EMFILE`**: The server has exhausted file descriptors. This happens when client connections aren't properly closed, leading to fd leaks.
3. **New connections hang in `SYN_RECEIVED`**: The server's accept backlog is full. Increase the backlog parameter in `listen()` or process connections faster.

Thread Management Problems:

Server threading issues often manifest as resource exhaustion or hanging connections. The thread-per-connection model can consume significant resources:

Problem	Symptoms	Diagnosis	Solution
Thread leak	Memory usage grows indefinitely	`ps -eLf | grep python | wc -l`	Ensure threads join after client disconnect
Too many threads	`OSError: can't create thread`	Monitor thread count vs system limits	Connection pooling, async I/O
Deadlock in threads	Some connections hang indefinitely	Thread dump, check `MethodRegistry` lock usage	Review locking order, use timeouts

⚠️ Pitfall: Forgetting to set `daemon=True` on handler threads

When creating threads to handle client connections, forgetting to set `daemon=True` means the server process won't exit cleanly when the main thread ends. The process hangs waiting for handler threads to complete, which never happens if clients are still connected. Always use `daemon=True` for connection handler threads, and implement graceful shutdown if you need to wait for requests to complete.

Client Connection Management

The `RPCClient` maintains a persistent connection to the server, which introduces complexity around connection lifecycle management:

Connection Establishment Issues:

1. **Initial connection succeeds but subsequent calls fail**: The client socket might be in an inconsistent state. Implement connection health checks by sending a small test message before important RPC calls.

2. **Connection works locally but fails in production**: Different network environments (NAT, load balancers, proxies) can interfere with TCP connections. Log the actual IP addresses being used to ensure correct routing.
3. **Connection timeouts are too short/long**: Network conditions vary dramatically. Implement adaptive timeouts that adjust based on measured round-trip times.

Connection Reuse Problems:

The `_ensure_connected()` method determines when to establish or reuse connections. Common mistakes include:

- Not detecting broken connections before reuse
- Racing between connection cleanup and new requests
- Sharing connections across threads without proper synchronization

Network Timeout Behavior

Timeouts in network programming have subtle behavior that often surprises developers. The `RPCClient` must handle multiple timeout scenarios:

Timeout Type	What It Controls	Default Behavior	Debugging Tips
Connection timeout	Time to establish socket connection	OS default (often 60+ seconds)	Use `socket.settimeout()` before `connect()`
Send timeout	Time to send complete message	Blocks indefinitely	Use `select()` or async I/O
Receive timeout	Time to receive complete response	Blocks indefinitely	Set socket timeout, handle `socket.timeout`
RPC call timeout	End-to-end call completion	Application-defined	Track time from request send to response

> **Design Insight**: Network timeouts should be layered - a short connection timeout (5-10 seconds), medium send/receive timeouts (30 seconds), and longer RPC call timeouts (60+ seconds). This provides fast feedback for obvious failures while allowing legitimate long-running operations to complete.

Testing Network Edge Cases

Network issues are often intermittent and environment-dependent. Systematic testing helps identify problems before they occur in production:

Network Simulation Techniques:

1. **Latency injection**: Use `tc` (traffic control) on Linux to add artificial network delays: `tc qdisc add dev lo root netem delay 100ms`
2. **Packet loss simulation**: Simulate unreliable networks: `tc qdisc add dev lo root netem loss 1%`
3. **Bandwidth limiting**: Test behavior under slow connections: `tc qdisc add dev lo root tbf rate 1kbit burst 1600 limit 3000`
4. **Connection interruption**: Use `iptables` to drop connections mid-stream and test recovery logic.

Serialization and Protocol Issues

Serialization problems occur when converting between Python objects and the JSON wire format. These issues often manifest as parsing errors, type mismatches, or corrupted data. Unlike network issues that typically fail fast, serialization problems can cause subtle data corruption that's discovered much later.

> **Key Insight**: Serialization bugs often stem from assumptions about data types that hold during testing but break with real-world data variations. Always test with edge cases like empty values, very large numbers, Unicode strings, and nested data structures.

JSON Serialization Edge Cases

JSON serialization seems straightforward but has numerous edge cases that can break RPC communication:

Data Type	JSON Limitation	Failure Mode	Workaround
`datetime` objects	No native datetime type	`TypeError: datetime not JSON serializable`	Convert to ISO strings, custom serializer
`Decimal` numbers	Only supports float precision	Precision loss, rounding errors	String encoding for exact decimals
`bytes` objects	No binary data support	`TypeError: bytes not JSON serializable`	Base64 encoding
`set` collections	No set type	`TypeError: set not JSON serializable`	Convert to list, document ordering loss
`None` vs empty string	Both can represent "missing"	Logic errors in parameter validation	Explicit null checking
Very large integers	Implementation limits vary	Precision loss in JavaScript clients	String encoding for big integers
NaN/Infinity floats	Not part of JSON spec	Varies by JSON library	Replace with null or string representation

⚠️ Pitfall: Assuming JSON round-trip fidelity

A common mistake is assuming that `deserialize_message(serialize_message(data))` always returns data identical to the original. JSON serialization loses type information and precision. For example, Python tuples become lists, Decimal numbers become floats, and dictionary keys must be strings. Design your RPC interface to handle these transformations gracefully.

Message Format Validation

The `validate_request_message()` and `validate_response_message()` functions check JSON-RPC format compliance, but validation failures often indicate deeper issues:

Request Validation Failures:

Validation Error	Potential Cause	Investigation Steps
Missing `method` field	Client serialization bug	Check `create_request_message()` logic
Invalid `params` type	Parameter marshaling error	Log raw parameters before serialization
Missing `id` field	Request ID generation failure	Verify `generate_request_id()` function
Extra fields in request	Version mismatch, debugging code	Compare client/server protocol versions

Response Validation Failures:

Response validation errors usually indicate server-side problems in request processing or response generation:

1. **Missing `result` and `error` fields**: The server's request dispatcher isn't following JSON-RPC protocol. Check the `process_request()` method for edge cases where neither result nor error is set.
2. **Wrong `id` field**: Response correlation failure. This serious bug means responses can be delivered to the wrong pending request. Debug the request tracking logic in `_track_request()` and `_wait_for_response()`.
3. **Invalid error format**: The server's error handling doesn't conform to JSON-RPC error structure. Verify that all error paths use `create_error_message()` with proper error codes.

Parameter Type Marshaling

RPC calls involve converting Python function signatures to JSON arrays and back. Type mismatches cause failures that can be hard to debug:

Common Type Marshaling Issues:

1. **Positional vs keyword arguments**: JSON-RPC traditionally uses positional parameters (JSON arrays), but Python functions often use keyword arguments. The `call()` method must decide how to marshal mixed args/kwargs.

2. **Complex nested objects**: While simple types serialize cleanly, custom objects require explicit serialization logic. Attempting to serialize objects with circular references will cause infinite recursion.
3. **Function signature validation**: The server should validate that the provided parameters match the registered function's signature. Mismatches should return `INVALID_PARAMS` errors, not Python exceptions.

Message Framing Errors

The length-prefix message framing protocol can fail in subtle ways, especially under high load or with large messages:

Framing Protocol Issues:

Symptom	Root Cause	Diagnostic Approach
`recv()` returns fewer bytes than expected	Network fragmentation or slow sender	Log actual vs expected byte counts
Length prefix reads as impossibly large number	Byte order issues or corruption	Check endianness, validate length bounds
Message appears truncated	Premature connection close	Monitor socket state during receive
JSON parsing fails on seemingly valid data	UTF-8 encoding issues	Hex dump received bytes, check encoding

Message Size Validation:

The `MAX_MESSAGE_SIZE` constant protects against memory exhaustion from malicious or corrupted length prefixes. However, legitimate large messages can trigger this protection:

1. **Large parameter lists**: RPC calls with many parameters or large string/binary data can exceed size limits. Consider pagination or chunking for large data transfers.
2. **Detailed error messages**: Error responses containing full stack traces can be surprisingly large. Truncate or summarize error details for production systems.

Debugging Serialization Problems

Systematic debugging of serialization issues requires visibility into the data transformation pipeline:

Data Flow Tracing:

1. **Pre-serialization logging**: Log the exact Python objects before JSON conversion to verify the input data integrity.
2. **Wire format inspection**: Log the raw bytes sent over the network to check for encoding issues or corruption.

3. **Post-deserialization validation**: After parsing JSON, log the reconstructed Python objects to verify round-trip accuracy.

JSON Library Behavior:

Different JSON libraries have subtly different behavior that can cause interoperability issues:

- Standard `json` module: Strict JSON compliance, good error messages
- `orjson`: High performance, but slightly different float formatting
- `ujson`: Fast parsing, but less robust error handling

> **Design Principle**: Use the standard `json` module during development for better error messages, then consider performance alternatives only if profiling shows JSON parsing as a bottleneck.

Timing and Concurrency Issues

Timing and concurrency bugs are among the most challenging to debug in RPC systems because they're often non-deterministic, environment-dependent, and may only manifest under specific load conditions. These issues arise from the inherent concurrency in network programming - multiple clients connecting simultaneously, overlapping requests and responses, and the need to correlate asynchronous operations.

> **Key Insight**: Concurrency bugs in RPC systems often involve shared state that seems fine during single-threaded testing but breaks when multiple operations access it simultaneously. The key to debugging is making these race conditions reproducible through stress testing and careful instrumentation.

Request ID Correlation Problems

The request ID mechanism allows clients to match responses with their originating requests when multiple calls are in flight. Bugs in this correlation system can cause responses to be delivered to the wrong callers or lost entirely:

Request ID Generation Issues:

Problem	Symptoms	Debugging Approach	Solution
Duplicate request IDs	Responses delivered to wrong calls	Log all generated IDs, check for duplicates	Use UUIDs instead of counters
ID wraparound in counters	Old responses match new requests	Monitor ID space exhaustion	64-bit IDs or UUID4 generation
Thread-unsafe ID generation	Race conditions in ID assignment	Stress test with concurrent calls	Thread-safe ID generator

⚠️ Pitfall: Using simple incrementing counters for request IDs

A common mistake is using a simple counter like `self._next_id += 1` to generate request IDs. This creates two problems: thread safety (multiple threads can get the same ID) and eventual wraparound (after 2^32 requests, IDs repeat). Use `uuid.uuid4()` or a thread-safe counter with sufficient bit width.

Request Tracking Race Conditions:

The `_pending_requests` dictionary tracks active requests, but concurrent access creates race conditions:

1. **Lost response handling**: A response arrives just as the timeout handler is cleaning up the request. Both threads try to modify `_pending_requests` simultaneously.
2. **Double response processing**: Network duplication causes the same response to arrive twice. Without proper cleanup, both copies might be processed.
3. **Memory leaks from orphaned requests**: If request cleanup fails due to exceptions, entries remain in `_pending_requests` indefinitely.

Request Tracking Debug Strategy:

```
# Effective debugging requires logging request lifecycle events:
# 1. Request creation and tracking registration
# 2. Response arrival and correlation
# 3. Timeout expiration and cleanup
# 4. Manual cleanup (disconnect/shutdown)
```

Timeout Handling Complexity

Timeout handling involves multiple concurrent operations: sending requests, waiting for responses, and cleaning up expired requests. These operations must coordinate without blocking each other:

Timeout Implementation Challenges:

Timeout Type	Concurrency Challenge	Failure Mode
Fixed timeouts	All requests use same duration	Fast operations wait unnecessarily, slow operations timeout prematurely
Per-request timeouts	Complex timeout tracking	Race between timeout and response arrival
Adaptive timeouts	Requires RTT measurement	Feedback loops can cause instability

Timeout Race Conditions:

The most common timeout bug occurs when a response arrives just as the timeout expires:

1. **Thread A**: Waits for response, timeout expires, starts cleanup

2. **Thread B**: Receives response from network, starts processing
3. **Race condition**: Both threads try to remove the request from `_pending_requests`

The solution requires atomic operations and careful ordering:

```
# Timeout handler must:
# 1. Atomically remove request from tracking (only if still present)
# 2. Signal waiting thread with timeout error
# 3. Avoid double-signaling if response already arrived

# Response handler must:
# 1. Atomically remove request from tracking (only if still present)
# 2. Signal waiting thread with response
# 3. Gracefully handle case where request was already timed out
```

Server Concurrency Issues

The `RPCServer` handles multiple client connections concurrently, creating shared state access patterns that can cause race conditions:

Method Registry Thread Safety:

The `MethodRegistry` allows dynamic registration and execution of RPC methods. Concurrent access creates several race conditions:

Operation Combination	Race Condition	Impact
Register + Execute	Method added while call in progress	Call might use old or new version inconsistently
Unregister + Execute	Method removed while call in progress	Call fails with method-not-found despite being valid
Multiple Registers	Same method name registered by different threads	Undefined which implementation wins

> **Architecture Decision: Method Registry Locking Strategy**
>
> - **Context**: The MethodRegistry needs thread-safe access for reading (method lookup) and writing (registration)
> - **Options Considered**:
>   1. Single mutex for all operations
>   2. Read-write lock allowing concurrent reads
>   3. Copy-on-write with atomic pointer swap
> - **Decision**: Read-write lock (`threading.RLock`)
> - **Rationale**: Method execution (reads) vastly outnumbers registration (writes), so allowing concurrent reads improves performance significantly
> - **Consequences**: More complex lock management, but better scalability under load

Connection Handler Thread Lifecycle:

Each client connection gets its own handler thread, but thread management introduces timing issues:

1. **Thread creation overhead**: Creating threads for each connection is expensive. Under high connection rates, the server spends more time creating threads than processing requests.
2. **Thread cleanup timing**: Handler threads must clean up resources when clients disconnect. If the client closes the connection abruptly, the cleanup code might not execute properly.
3. **Graceful shutdown coordination**: When the server shuts down, it must wait for all handler threads to complete their current requests before exiting.

Deadlock Detection and Prevention

Deadlocks in RPC systems typically involve multiple locks being acquired in different orders by different threads. The most common scenario involves the method registry lock and request tracking locks:

Common Deadlock Patterns:

Thread 1 Operations	Thread 2 Operations	Deadlock Scenario
1. Acquire registry lock 2. Wait for tracking lock	1. Acquire tracking lock 2. Wait for registry lock	Classic two-lock deadlock
1. Process request (registry) 2. Send response (socket)	1. Handle timeout (tracking) 2. Update registry stats	Nested lock acquisition with I/O

Deadlock Prevention Strategies:

1. **Lock ordering**: Always acquire locks in the same order across all code paths. Document the lock hierarchy clearly.

2. **Lock timeouts**: Use timed lock acquisition to detect potential deadlocks. If a lock can't be acquired within a reasonable time, log a warning and abort the operation.
3. **Minimize lock scope**: Hold locks for the shortest time possible. Don't perform I/O operations while holding locks.

Load Testing for Concurrency Issues

Concurrency bugs often only appear under specific load patterns. Systematic load testing helps expose race conditions that don't occur during development:

Effective Load Testing Patterns:

Test Pattern	Purpose	Implementation
Burst connections	Test connection handling limits	Open 100+ connections simultaneously
Concurrent method calls	Expose request tracking races	Multiple threads calling same method
Mixed read/write operations	Test registry lock contention	Register methods while handling calls
Connection drops during calls	Test cleanup logic	Kill client connections mid-request

Concurrency Bug Indicators:

Watch for these symptoms during load testing:

1. **Inconsistent test results**: The same test passes sometimes and fails others
2. **Resource leaks**: Memory or file descriptor usage grows over time
3. **Hanging operations**: Requests never complete or timeout
4. **Assertion failures**: Race conditions causing invalid state

⚠️ Pitfall: Testing only single-threaded scenarios

Many developers test RPC functionality with a single client making sequential calls. This approach misses most concurrency bugs. Always include multi-threaded and multi-client test scenarios. Use tools like `threading.Barrier` to coordinate multiple threads and create worst-case race condition scenarios.

Performance Monitoring and Profiling

Timing issues often manifest as performance problems rather than functional failures. Monitoring key metrics helps identify concurrency bottlenecks:

Key Timing Metrics:

Metric	What It Reveals	Monitoring Approach
Request latency distribution	Lock contention, queueing delays	Histogram of response times
Thread pool utilization	Resource exhaustion, blocking operations	Active thread count over time
Lock acquisition time	Contention hotspots	Time spent waiting for locks
Connection establishment rate	Network or threading bottlenecks	Connections per second

> **Debugging Principle**: Performance problems and correctness problems in concurrent systems are often related. A performance degradation might indicate race conditions, lock contention, or resource leaks that will eventually cause functional failures.

Implementation Guidance

This section provides practical tools and techniques for systematically debugging RPC framework issues. The debugging utilities help isolate problems quickly and provide detailed diagnostic information.

Technology Recommendations

Debugging Category	Simple Option	Advanced Option
Network diagnostics	`netstat`, `telnet`, basic socket logging	`tcpdump`, `wireshark`, network monitoring tools
Concurrency debugging	`threading` module logging, manual trace points	Thread profilers, deadlock detection tools
Serialization validation	JSON pretty-printing, manual data inspection	Schema validation libraries, fuzzing tools
Performance monitoring	Basic timing with `time.time()`	`cProfile`, `py-spy`, APM tools

Debugging Infrastructure

Complete Socket Diagnostic Helper:

PYTHON

```python
import socket

import time

import threading

from typing import Dict, Any, Optional

from dataclasses import dataclass


@dataclass

class ConnectionDiagnostics:

    """Diagnostic information about a socket connection."""

    local_address: tuple

    remote_address: tuple

    socket_state: str

    bytes_sent: int

    bytes_received: int

    connection_time: float

    last_activity: float


class SocketDiagnosticsHelper:

    """Helper for debugging socket-related issues."""



    def __init__(self):

        self._connections: Dict[int, ConnectionDiagnostics] = {}

        self._lock = threading.Lock()



    def register_connection(self, sock: socket.socket) -> None:

        """Register a socket for diagnostic tracking."""

        with self._lock:

            try:
```

```
            local_addr = sock.getsockname()
            remote_addr = sock.getpeername()
            sock_fd = sock.fileno()

            self._connections[sock_fd] = ConnectionDiagnostics(
                local_address=local_addr,
                remote_address=remote_addr,
                socket_state="CONNECTED",
                bytes_sent=0,
                bytes_received=0,
                connection_time=time.time(),
                last_activity=time.time()
            )
        except OSError as e:
            print(f"Failed to register socket: {e}")

    def record_send(self, sock: socket.socket, byte_count: int) -> None:
        """Record bytes sent on a socket."""
        # TODO: Update connection diagnostics with send activity
        # TODO: Update last_activity timestamp
        # TODO: Increment bytes_sent counter
        # TODO: Handle case where socket is not registered
        pass

    def record_receive(self, sock: socket.socket, byte_count: int) -> None:
        """Record bytes received on a socket."""
        # TODO: Update connection diagnostics with receive activity
```

```
        # TODO: Update last_activity timestamp
        # TODO: Increment bytes_received counter
        # TODO: Handle case where socket is not registered
        pass


    def test_connectivity(self, host: str, port: int, timeout: float = 5.0) -> Dict[str,
Any]:
        """Test basic connectivity to a server."""
        # TODO: Create test socket with timeout
        # TODO: Attempt connection to host:port
        # TODO: Measure connection time
        # TODO: Test basic send/receive capability
        # TODO: Return diagnostic dictionary with results
        # Hint: Include success/failure, timing, and any error details
        pass
```

Message Serialization Debugger:

PYTHON

```python
import json

import traceback

from typing import Any, Dict, List, Optional


class SerializationDebugger:

    """Helper for debugging JSON serialization issues."""



    @staticmethod

    def validate_serializable(data: Any, path: str = "root") -> List[str]:

        """Validate that data can be JSON serialized and return issues."""

        issues = []



        # TODO: Recursively check data structure for serialization problems

        # TODO: Check for non-serializable types (datetime, bytes, etc.)

        # TODO: Check for circular references

        # TODO: Validate that dictionary keys are strings

        # TODO: Check for very large numbers that might lose precision

        # TODO: Return list of issues with their paths in the data structure



        return issues



    @staticmethod

    def safe_serialize(data: Any) -> Optional[bytes]:

        """Attempt serialization with detailed error reporting."""

        try:

            # TODO: Try to serialize data to JSON

            # TODO: Return encoded bytes if successful
```

```
            # TODO: Catch and log specific serialization errors
            # TODO: Return None if serialization fails
            pass
        except Exception as e:
            print(f"Serialization failed: {e}")
            traceback.print_exc()
            return None

    @staticmethod
    def compare_round_trip(original: Any) -> Dict[str, Any]:
        """Test round-trip serialization and report differences."""
        # TODO: Serialize original data to JSON
        # TODO: Deserialize back to Python object
        # TODO: Compare original with round-trip result
        # TODO: Report type changes, precision loss, etc.
        # TODO: Return detailed comparison report
        pass
```

Request Tracking Diagnostics:

PYTHON

```python
import threading

import time

from typing import Dict, Any

from dataclasses import dataclass, field


@dataclass

class RequestDiagnostics:

    """Diagnostic information about an RPC request."""

    request_id: str

    method_name: str

    created_time: float

    timeout_seconds: float

    thread_id: int

    state: str = "PENDING"

    response_time: Optional[float] = None

    error_info: Optional[str] = None


class RequestTrackingDiagnostics:

    """Helper for debugging request correlation and timeout issues."""



    def __init__(self):

        self._active_requests: Dict[str, RequestDiagnostics] = {}

        self._completed_requests: List[RequestDiagnostics] = []

        self._lock = threading.RLock()

        self._max_completed_history = 100



    def register_request(self, request_id: str, method_name: str, timeout_seconds: float) -
> None:
```

```
        """Register a new request for tracking."""
        # TODO: Create RequestDiagnostics object with current time and thread
        # TODO: Add to active_requests dictionary
        # TODO: Use lock for thread safety
        pass

    def mark_response_received(self, request_id: str) -> bool:
        """Mark request as having received a response."""
        # TODO: Find request in active_requests
        # TODO: Update state to "COMPLETED" and set response_time
        # TODO: Move from active to completed list
        # TODO: Return True if found, False if not
        # TODO: Handle case where request was already cleaned up
        pass

    def mark_timeout(self, request_id: str) -> bool:
        """Mark request as timed out."""
        # TODO: Find request in active_requests
        # TODO: Update state to "TIMEOUT" and set error_info
        # TODO: Move from active to completed list
        # TODO: Return True if found, False if already cleaned up
        pass

    def get_diagnostics_summary(self) -> Dict[str, Any]:
        """Get summary of request tracking state."""
        with self._lock:
            current_time = time.time()
```

```
        active_count = len(self._active_requests)


        # TODO: Calculate statistics on active requests
        # TODO: Find oldest active request and check if it's overdue
        # TODO: Count requests by state in completed history
        # TODO: Calculate average response times
        # TODO: Return comprehensive diagnostic dictionary


        return {
            "active_requests": active_count,
            "completed_requests": len(self._completed_requests),
            "timestamp": current_time
            # Add more diagnostic fields
        }
```

Milestone Checkpoints

After Milestone 1 (Message Protocol):

Test serialization robustness with edge cases:

```bash
# Run this test to validate message protocol

python -c "

from your_rpc.protocol import create_request_message, serialize_message,
deserialize_message

import json


# Test edge cases

edge_cases = [

    ('test_method', []),  # Empty parameters

    ('test_method', [None, '', 0, False]),  # Falsy values

    ('test_method', ['unicode: 你好', 'emoji: 😀']),  # Unicode strings

    ('test_method', [1.7976931348623157e+308]),  # Very large float

]


for method, params in edge_cases:

    msg = create_request_message(method, params, 'test-id')

    serialized = serialize_message(msg)

    deserialized = deserialize_message(serialized)

    print(f'✓ {method} with {params} serialized successfully')

"
```

Expected output: All edge cases should serialize without exceptions.

After Milestone 2 (Server Implementation):

Test server concurrency and error handling:

```bash
# Test concurrent connections to the server
python -c "
import threading
import socket
import time

def test_connection():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        sock.connect(('localhost', 8000))
        sock.send(b'test message')
        sock.close()
        print('✓ Connection successful')
    except Exception as e:
        print(f'✗ Connection failed: {e}')

# Start 10 concurrent connections
threads = [threading.Thread(target=test_connection) for _ in range(10)]
for t in threads:
    t.start()
for t in threads:
    t.join()
"
```

Expected output: All 10 connections should succeed without server errors.

After Milestone 3 (Client Implementation):

Test timeout handling and request correlation:

```bash
# Test client timeout behavior
python -c "
from your_rpc.client import RPCClient
import threading
import time

client = RPCClient('localhost', 8000)

def make_call(method, delay):
    try:
        result = client.call(method, delay, timeout=2.0)
        print(f'✓ {method} completed: {result}')
    except Exception as e:
        print(f'✗ {method} failed: {e}')

# Test concurrent calls with different timeouts
threads = [
    threading.Thread(target=make_call, args=('fast_method', 0.5)),
    threading.Thread(target=make_call, args=('slow_method', 5.0)),
    threading.Thread(target=make_call, args=('medium_method', 1.5)),
]

for t in threads:
    t.start()
for t in threads:
    t.join()
"
```

Expected output: Fast and medium calls succeed, slow call times out after 2 seconds.

Common Debugging Scenarios

Symptom: Client calls hang indefinitely

Diagnosis Step	Command/Technique	Expected Result
Check server running	`netstat -an | grep 8000`	Should show `LISTEN` state
Test basic connectivity	`telnet localhost 8000`	Should connect successfully
Check request ID tracking	Add logging to `_track_request()`	Should show request being registered
Monitor response correlation	Add logging to response handler	Should show if responses arrive

Symptom: Intermittent serialization errors

Diagnosis Step	Investigation Technique	What to Look For
Log pre-serialization data	Add logging before `json.dumps()`	Non-serializable types, circular refs
Test with simple data first	Replace complex params with basic types	Isolate problematic data structures
Check Unicode handling	Test with various string encodings	UTF-8 encoding issues
Validate JSON compliance	Use strict JSON parser	Invalid JSON constructs

Symptom: Server handles some requests but not others

Diagnosis Area	Investigation Steps
Method registry	Log all registered methods, check method name matching
Parameter validation	Log received parameters vs expected function signature
Exception handling	Ensure all exceptions in method handlers are caught
Thread synchronization	Check for deadlocks in method registry access

Performance Debugging Tools

Request Latency Tracker:

```python
import time

import collections

from typing import DefaultDict, List


class LatencyTracker:

    """Track RPC call latencies for performance debugging."""



    def __init__(self):

        self._method_times: DefaultDict[str, List[float]] = collections.defaultdict(list)

        self._request_starts: Dict[str, float] = {}



    def start_request(self, request_id: str, method_name: str) -> None:

        """Record request start time."""

        # TODO: Store current timestamp for request_id

        # TODO: Associate with method_name for categorization

        pass



    def end_request(self, request_id: str, method_name: str) -> None:

        """Record request completion and calculate latency."""

        # TODO: Calculate elapsed time from start

        # TODO: Add to method_times list for the method

        # TODO: Remove from request_starts

        # TODO: Handle case where request_id not found (late response)

        pass



    def get_latency_stats(self, method_name: str = None) -> Dict[str, float]:

        """Get latency statistics for debugging."""
```

```
        # TODO: Calculate min, max, mean, median, 95th percentile
        # TODO: Return for specific method or all methods
        # TODO: Handle case where no data is available
        pass
```

This debugging guide provides systematic approaches to identify and resolve the most common issues in RPC framework development. The key is following structured diagnostic processes rather than random troubleshooting, using proper instrumentation to make problems visible, and testing edge cases that don't occur during normal development.

Future Extensions

> **Milestone(s):** Beyond Milestones 1, 2, and 3 - advanced features that build upon the basic RPC framework foundation

The basic RPC framework provides a solid foundation for remote procedure calls, but real-world systems require additional capabilities for performance, reliability, and scalability. This section explores natural extensions that transform our educational framework into a production-ready system. These enhancements follow the same design principles we've established while addressing the operational challenges that emerge when RPC systems face high load, unreliable networks, and security requirements.

Mental Model: The Growing City Analogy

Think of our basic RPC framework as a small town with a simple postal service. The town has grown successful, and now needs urban infrastructure improvements. The postal service (our message protocol) needs express lanes for high-priority mail (performance extensions). The town needs security systems and backup services (reliability extensions). Each extension addresses specific growing pains while preserving the fundamental architecture that made the original system work.

The extensions fall into two primary categories: **performance extensions** that make the system faster and more efficient, and **reliability extensions** that make the system more robust and secure. Like urban planning, these improvements must be carefully designed to work together without creating conflicts or compromising the system's core simplicity.

Performance Extensions

Performance extensions focus on reducing latency, increasing throughput, and minimizing resource consumption. These improvements are essential when the RPC framework must handle thousands of concurrent requests or when network efficiency becomes critical.

Connection Pooling

Connection pooling transforms the client from establishing a new TCP connection for each RPC call to maintaining a reusable pool of persistent connections. This eliminates the overhead of TCP handshaking and connection teardown that becomes significant under high request volumes.

Current Limitation: The basic `RPCClient` establishes a single connection and reuses it for sequential requests. This creates a bottleneck when multiple threads need to make concurrent RPC calls, as they must wait for the shared connection to become available.

> **Decision: Connection Pool Architecture**
>
> - **Context**: Multiple threads making concurrent RPC calls are serialized by the single shared connection, creating a throughput bottleneck and blocking threads unnecessarily
> - **Options Considered**:
>   - Thread-local connections (one connection per thread)
>   - Shared connection pool with borrowing/returning semantics
>   - Connection-per-request with caching
> - **Decision**: Shared connection pool with borrowing/returning semantics
> - **Rationale**: Thread-local connections can create too many connections under high thread counts, while connection-per-request loses reuse benefits. A shared pool provides optimal balance of connection reuse and concurrency.
> - **Consequences**: Requires connection lifecycle management and pool sizing configuration, but enables true concurrent RPC calls without thread blocking

Design Aspect	Thread-Local Connections	Shared Connection Pool	Connection-Per-Request
Concurrency	Excellent	Good	Excellent
Resource Usage	High (many connections)	Medium	Low
Connection Reuse	Good	Excellent	Poor
Implementation Complexity	Medium	High	Low
Chosen	No	**Yes**	No

The connection pool maintains a collection of established TCP connections that can be borrowed for RPC calls and returned when complete. The pool handles connection health checking, automatic reconnection for stale connections, and pool size management based on demand.

Connection Pool Components:

Component	Type	Description
`connections`	`List[socket.socket]`	Available connections ready for use
`in_use`	`Set[socket.socket]`	Connections currently borrowed for active requests
`max_size`	`int`	Maximum number of connections to maintain
`min_size`	`int`	Minimum connections to keep alive
`idle_timeout`	`float`	Seconds before closing idle connections
`health_check_interval`	`float`	Seconds between connection health checks
`pool_lock`	`threading.RLock`	Protects pool state during concurrent access

Connection Borrowing Algorithm:

1. Acquire the pool lock to ensure thread-safe access to pool state
2. Check if any healthy connections are available in the connections list
3. If available connection exists, move it from connections to in_use and return it
4. If no connections available but pool size under maximum, create new connection
5. Establish TCP connection to server with timeout and connection validation
6. Add new connection to in_use set and return it to caller
7. If pool at maximum size, block waiting for connection to be returned
8. Implement timeout mechanism to prevent infinite blocking on pool exhaustion

Connection Health Checking: The pool periodically validates that connections are still usable by sending lightweight ping messages or checking socket status. Broken connections are automatically removed from the pool and replaced with fresh connections.

> The key insight for connection pooling is that the complexity of pool management is isolated within the connection pool component, while the RPC calling interface remains unchanged. Existing code using `call()` methods continues to work transparently.

Asynchronous Support

Asynchronous support transforms the blocking RPC client into a non-blocking system that can handle thousands of concurrent requests without dedicating a thread to each pending call. This is essential for building responsive applications that make many RPC calls concurrently.

Current Limitation: The basic `RPCClient.call()` method blocks the calling thread until the response arrives or timeout occurs. Applications making hundreds of concurrent RPC calls must create hundreds of threads, leading to resource exhaustion and context switching overhead.

Decision: Async/Await Integration

- **Context**: Blocking RPC calls force applications to use thread-per-request patterns that don't scale well under high concurrency loads
- **Options Considered**:
  - Callback-based asynchronous interface
  - Future/Promise-based interface
  - Python asyncio integration with async/await
- **Decision**: Python asyncio integration with async/await syntax
- **Rationale**: Async/await provides the most natural programming model for asynchronous code, avoiding callback hell and making error handling straightforward
- **Consequences**: Requires asyncio event loop and separate async client implementation, but enables single-threaded high concurrency

Async Approach	Learning Curve	Error Handling	Debugging	Performance
Callbacks	High	Complex	Difficult	Good
Futures/Promises	Medium	Good	Medium	Good
Async/Await	**Medium**	**Excellent**	**Good**	**Excellent**

The asynchronous client uses Python's `asyncio` library to manage concurrent RPC calls without blocking threads. Instead of blocking on socket operations, the client registers interest in socket readiness with the event loop and yields control to other coroutines.

Async Client Components:

Component	Type	Description
`_event_loop`	`asyncio.AbstractEventLoop`	Event loop managing async operations
`_connection_pool`	`AsyncConnectionPool`	Non-blocking connection management
`_pending_requests`	`Dict[str, asyncio.Future]`	Futures awaiting responses
`_response_handler_task`	`asyncio.Task`	Background task processing incoming responses
`_request_semaphore`	`asyncio.Semaphore`	Limits concurrent request count

Async Call Flow:

1. Create request message with unique ID and serialize to bytes
2. Acquire connection from async connection pool (may await if pool full)

3. Send request bytes over connection using asyncio socket operations
4. Create Future object for this request ID and store in pending requests
5. Return the Future to caller for awaiting (call is now non-blocking)
6. Background response handler receives responses and resolves matching Futures
7. When response arrives, lookup Future by request ID and set result/exception
8. Calling code awaits the Future and receives the result when available

Background Response Handler: A dedicated asyncio task continuously reads responses from all active connections and routes them to the appropriate Future objects. This single task can handle responses from thousands of concurrent requests efficiently.

> The async transformation maintains the same logical request/response model while eliminating thread blocking. The programming model changes from `result = client.call(method, args)` to `result = await async_client.call(method, args)`.

Binary Serialization

Binary serialization replaces JSON encoding with more efficient binary formats like MessagePack or Protocol Buffers. This reduces message size and serialization overhead, especially important for large payloads or high-frequency RPC calls.

Current Limitation: JSON serialization is human-readable and easy to debug, but produces large messages and requires significant CPU time for encoding/decoding. For RPC calls with large data structures or high call volumes, serialization becomes a performance bottleneck.

> **Decision: MessagePack as Primary Binary Format**
>
> - **Context**: JSON serialization creates large messages and high CPU overhead for complex data structures with nested objects and arrays
> - **Options Considered**:
>   - MessagePack (schema-less binary JSON equivalent)
>   - Protocol Buffers (schema-based with code generation)
>   - Apache Avro (schema-based with runtime schema evolution)
> - **Decision**: MessagePack as primary format with JSON fallback
> - **Rationale**: MessagePack maintains JSON's schema-less flexibility while providing binary efficiency, and requires no schema management or code generation
> - **Consequences**: Smaller messages and faster serialization, but loses human readability and requires binary debugging tools

Serialization Format	Message Size	CPU Overhead	Schema Management	Debugging
JSON	Large	High	None	Excellent
MessagePack	**Small**	**Low**	**None**	**Fair**
Protocol Buffers	Smallest	Lowest	Complex	Poor
Apache Avro	Small	Medium	Medium	Fair

Binary Serialization Architecture: The framework supports multiple serialization formats through a pluggable serialization interface. Clients and servers negotiate the serialization format during connection establishment, with fallback to JSON for compatibility.

Serialization Interface:

Method	Parameters	Returns	Description
`serialize_message(message, format)`	`Dict, str`	`bytes`	Encode message using specified format
`deserialize_message(data, format)`	`bytes, str`	`Dict`	Decode bytes using specified format
`get_supported_formats()`	None	`List[str]`	List available serialization formats
`negotiate_format(client_formats, server_formats)`	`List[str], List[str]`	`str`	Select optimal compatible format

Format Negotiation Process:

1. Client sends connection handshake including list of supported serialization formats
2. Server responds with its supported formats and selects best mutual format
3. Both client and server use negotiated format for all subsequent messages
4. If no mutual format exists, fall back to JSON as universal compatibility format
5. Format selection considers efficiency, with preference for binary formats over JSON

MessagePack Integration Benefits:

- Message size typically 30-50% smaller than equivalent JSON
- Serialization speed 2-3x faster than JSON for complex objects
- Direct mapping from JSON data types (no schema required)
- Preserves type information (distinguishes integers from floats)
- Support for binary data without base64 encoding overhead

> Binary serialization provides the most significant performance improvement for RPC systems with large payloads or high call volumes, often reducing network bandwidth by 40% and serialization CPU usage by 60%.

Reliability Extensions

Reliability extensions focus on making the RPC framework robust against failures, secure against attacks, and resilient under adverse conditions. These improvements are essential for production systems that must operate continuously despite network failures, server crashes, and malicious actors.

Authentication and Authorization

Authentication verifies the identity of RPC clients, while authorization determines which methods each authenticated client is permitted to invoke. This prevents unauthorized access and enables fine-grained permission control over RPC services.

Current Limitation: The basic RPC framework has no security mechanisms. Any client that can connect to the server can invoke any registered method, creating serious security vulnerabilities in production environments.

> **Decision: Token-Based Authentication with Method-Level Authorization**
>
> - **Context**: Production RPC services need to verify client identity and restrict access to sensitive methods based on client permissions
> - **Options Considered**:
>   - HTTP Basic Authentication (simple but sends credentials repeatedly)
>   - JWT tokens (stateless but require signature verification)
>   - Session-based authentication (stateful but requires session storage)
> - **Decision**: JWT token authentication with method-level authorization rules
> - **Rationale**: JWT tokens are stateless, self-contained, and can carry authorization claims, eliminating the need for server-side session storage while supporting fine-grained permissions
> - **Consequences**: Adds token verification overhead to each request, but provides robust security without server-side state management

Authentication Method	Stateless	Performance	Security	Implementation Complexity
Basic Auth	Yes	Excellent	Poor	Low
JWT Tokens	**Yes**	**Good**	**Excellent**	**Medium**
Session-Based	No	Good	Good	High

Authentication Architecture: Authentication is implemented as a middleware layer that intercepts requests before they reach the method registry. The middleware extracts authentication tokens, validates them, and enriches the request context with identity and permission information.

Authentication Components:

Component	Type	Description
`token_validator`	`JWTValidator`	Validates JWT token signatures and expiration
`permission_store`	`PermissionStore`	Maps user identities to method permissions
`auth_middleware`	`AuthMiddleware`	Intercepts requests and enforces authentication
`client_identity`	`ClientIdentity`	Represents authenticated client with permissions

Authentication Request Flow:

1. Client includes JWT token in request message header or dedicated auth field
2. Server's auth middleware extracts token from incoming request message
3. Middleware validates token signature using configured public key or shared secret
4. Middleware checks token expiration time and other standard JWT claims
5. Middleware extracts user identity and roles from token's custom claims
6. Permission store lookup determines which methods this identity can invoke
7. If target method is authorized, request continues to method registry for execution
8. If authentication fails or method unauthorized, return error without invoking method

JWT Token Structure:

```
Header: {"alg": "HS256", "typ": "JWT"}
Payload: {
  "sub": "user_id",
  "exp": 1234567890,
  "iat": 1234567800,
  "roles": ["read_user", "write_orders"],
  "permissions": ["get_user_info", "create_order", "update_order"]
}
```

Method-Level Authorization: Each registered method can specify required permissions through decorators or configuration. The auth middleware checks that the authenticated client possesses all required permissions before allowing method execution.

Permission Model	Granularity	Management Overhead	Flexibility
Role-Based	Medium	Low	Good
Permission-Based	**High**	**Medium**	**Excellent**
Method-Level ACLs	Highest	High	Good

> Authentication and authorization add security at the cost of performance overhead. Token validation typically adds 1-2ms per request, which is acceptable for most applications but may require caching optimizations for extremely high-throughput systems.

Encryption and Transport Security

Encryption protects RPC messages from eavesdropping and tampering during network transmission. This is essential when RPC calls contain sensitive data or traverse untrusted networks like the public internet.

Current Limitation: The basic RPC framework uses plain TCP sockets that transmit messages in cleartext. Network attackers can read all RPC traffic and potentially modify messages in transit.

> **Decision: TLS 1.3 for Transport Encryption**
>
> - **Context**: RPC messages may contain sensitive data and travel across untrusted networks where eavesdropping and tampering are possible
> - **Options Considered**:
>   - Application-level encryption (encrypt message payloads)
>   - TLS transport encryption (encrypt entire TCP connection)
>   - VPN-based network encryption (encrypt at network layer)
> - **Decision**: TLS 1.3 for transport-level encryption
> - **Rationale**: TLS provides proven security with minimal application code changes, handles key exchange automatically, and encrypts all traffic including metadata
> - **Consequences**: Adds TLS handshake overhead and CPU cost for encryption/decryption, but provides comprehensive protection against network-level attacks

Encryption Approach	Protection Scope	Performance Impact	Implementation Effort
Application-Level	Message Content	Low	High
TLS Transport	**Complete Traffic**	**Medium**	**Low**
Network-Level VPN	Complete Traffic	Low	External

TLS Integration Architecture: TLS encryption is implemented by wrapping the TCP sockets in TLS sockets that handle encryption/decryption transparently. The existing RPC message protocol operates unchanged over the encrypted transport.

TLS Configuration Components:

Component	Type	Description
`tls_context`	`ssl.SSLContext`	TLS configuration including cipher suites and certificates
`server_certificate`	`ssl.Certificate`	Server's public key certificate for client verification
`private_key`	`ssl.PrivateKey`	Server's private key for TLS handshake
`ca_certificates`	`List[ssl.Certificate]`	Certificate authorities trusted for client certificates
`cipher_preferences`	`List[str]`	Ordered list of acceptable cipher suites

TLS Handshake Process:

1. Client initiates TCP connection to server as before
2. Client immediately starts TLS handshake by sending ClientHello message
3. Server responds with ServerHello, certificate, and key exchange information
4. Client validates server certificate against trusted certificate authorities
5. Client and server complete key exchange to establish shared encryption keys
6. Both sides send Finished messages to confirm successful handshake
7. All subsequent RPC messages are encrypted using established session keys
8. RPC protocol operates normally over the encrypted TLS connection

Certificate Management: Production deployments require proper certificate management with certificate rotation, certificate authority validation, and secure private key storage. Development environments can use self-signed certificates for testing.

> TLS encryption typically adds 5-10ms latency for the initial handshake and 10-20% CPU overhead for encryption/decryption. Connection reuse amortizes the handshake cost across many RPC calls.

Retry Mechanisms and Circuit Breakers

Retry mechanisms automatically re-execute failed RPC calls, while circuit breakers prevent cascading failures by temporarily stopping calls to unresponsive services. Together, they make RPC systems resilient to transient network failures and server overload.

Current Limitation: The basic RPC framework treats all failures as permanent. A temporary network glitch or momentary server overload causes RPC calls to fail immediately, even though a retry seconds later would

likely succeed.

> **Decision: Exponential Backoff Retry with Circuit Breaker Protection**
>
> - **Context**: Transient network failures and temporary server overload should not cause permanent RPC failures, but naive retry can amplify problems during outages
> - **Options Considered**:
>   - Fixed interval retry (simple but can overwhelm recovering servers)
>   - Exponential backoff retry (reduces load on recovering servers)
>   - Circuit breaker only (prevents cascading failures but no retry)
> - **Decision**: Exponential backoff retry with circuit breaker protection
> - **Rationale**: Exponential backoff handles transient failures gracefully while circuit breakers prevent retry storms during extended outages
> - **Consequences**: Improves reliability but adds complexity and potential for longer response times during failures

Resilience Pattern	Transient Failures	Extended Outages	Implementation Complexity
Fixed Retry	Good	Poor	Low
Exponential Backoff	**Excellent**	**Fair**	**Medium**
Circuit Breaker	Poor	Excellent	Medium
Combined Approach	**Excellent**	**Excellent**	**High**

Retry Configuration:

Parameter	Type	Description
`max_attempts`	`int`	Maximum retry attempts before permanent failure
`base_delay`	`float`	Initial delay before first retry (seconds)
`max_delay`	`float`	Maximum delay between retries (seconds)
`backoff_multiplier`	`float`	Multiplier for exponential delay increase
`jitter_enabled`	`bool`	Add randomization to prevent thundering herd
`retryable_errors`	`Set[ErrorCode]`	Which error types should trigger retry

Exponential Backoff Algorithm:

1. Execute initial RPC call attempt and capture any resulting error

2. Check if error type is in the configured set of retryable errors
3. If not retryable (e.g., authentication failure), return error immediately
4. If retryable and attempts remain, calculate delay as base_delay * (multiplier ^ attempt)
5. Add random jitter to delay to prevent synchronized retry storms
6. Sleep for calculated delay period before attempting next retry
7. Increment attempt counter and repeat from step 1 until success or max attempts
8. If all attempts exhausted, return the last error to the caller

Circuit Breaker States:

State	Request Handling	Transition Conditions	Purpose
`CLOSED`	Pass all requests	Failure rate exceeds threshold → OPEN	Normal operation
`OPEN`	Reject immediately	Timeout expires → HALF_OPEN	Prevent cascading failures
`HALF_OPEN`	Allow single test request	Success → CLOSED, Failure → OPEN	Test service recovery

Circuit Breaker Algorithm:

1. Track success and failure rates using sliding window of recent requests
2. When failure rate exceeds threshold (e.g., 50% failures in last 100 requests), trip to OPEN
3. In OPEN state, immediately return circuit breaker error without attempting RPC call
4. After timeout period (e.g., 60 seconds), transition to HALF_OPEN state
5. In HALF_OPEN, allow single test request to determine if service has recovered
6. If test request succeeds, reset failure counters and return to CLOSED state
7. If test request fails, return to OPEN state and restart timeout period

> The combination of retry and circuit breaker provides comprehensive resilience: retries handle brief failures while circuit breakers prevent retry storms during extended outages. This is essential for building robust distributed systems.

⚠️ **Common Pitfall: Retry Without Idempotency Consideration** Many developers implement retry logic without considering whether the RPC methods are idempotent (safe to call multiple times). Retrying non-idempotent operations like "transfer money" or "send email" can cause duplicate actions. The solution is to either ensure all retried methods are idempotent, or implement request deduplication using unique request IDs to detect and ignore duplicate calls.

⚠️ **Common Pitfall: Circuit Breaker Per Client Instead of Per Service** Implementing circuit breakers at the client instance level rather than per remote service can cause false positives. If one service is down but others

are healthy, a per-client circuit breaker might prevent all RPC calls. Instead, maintain separate circuit breaker state for each remote service endpoint.

Implementation Guidance

These performance and reliability extensions build upon the basic RPC framework established in earlier milestones. The implementation approach follows a modular design where extensions can be added incrementally without breaking existing functionality.

Technology Recommendations

Extension Category	Simple Option	Advanced Option
Connection Pooling	Thread-safe queue with basic pooling	`asyncio` connection pool with health checks
Async Support	Threading with futures	Full `asyncio` integration
Binary Serialization	MessagePack with fallback to JSON	Protocol Buffers with schema evolution
Authentication	Simple API keys	JWT tokens with role-based permissions
Encryption	TLS with self-signed certificates	TLS with proper CA certificates
Retry Logic	Fixed delay retry	Exponential backoff with jitter
Circuit Breakers	Simple failure counting	Sliding window with configurable thresholds

Recommended File Structure

```
rpc-framework/
├── rpc/
│   ├── extensions/
│   │   ├── __init__.py
│   │   ├── connection_pool.py      ← Connection pooling implementation
│   │   ├── async_client.py         ← Asyncio-based RPC client
│   │   ├── serialization.py        ← Pluggable serialization formats
│   │   ├── auth/
│   │   │   ├── __init__.py
│   │   │   ├── middleware.py       ← Authentication middleware
│   │   │   ├── jwt_validator.py    ← JWT token validation
│   │   │   └── permissions.py      ← Permission management
│   │   ├── security/
│   │   │   ├── __init__.py
│   │   │   ├── tls_config.py       ← TLS configuration helpers
│   │   │   └── certificates.py     ← Certificate management
│   │   └── resilience/
│   │       ├── __init__.py
│   │       ├── retry_handler.py    ← Retry logic with exponential backoff
│   │       └── circuit_breaker.py  ← Circuit breaker implementation
│   ├── core/                       ← Basic RPC framework from earlier milestones
│   │   ├── protocol.py
│   │   ├── server.py
│   │   └── client.py
│   └── utils/
│       ├── diagnostics.py          ← Enhanced debugging for extensions
│       └── testing.py              ← Test helpers for extensions
├── examples/
│   ├── pooled_client_example.py    ← Connection pooling demo
│   ├── async_client_example.py     ← Async RPC calls demo
│   ├── secure_server_example.py    ← TLS + authentication demo
│   └── resilient_client_example.py ← Retry + circuit breaker demo
└── tests/
    ├── test_extensions/
    │   ├── test_connection_pool.py
    │   ├── test_async_client.py
    │   ├── test_authentication.py
    │   └── test_resilience.py
    └── integration/
        └── test_production_scenarios.py
```

Connection Pool Infrastructure

```python
# rpc/extensions/connection_pool.py

import threading

import socket

import time

from typing import List, Set, Optional

from queue import Queue, Empty

from contextlib import contextmanager


class ConnectionPool:

    """Thread-safe connection pool for RPC clients."""


    def __init__(self, host: str, port: int, max_size: int = 10,

                 min_size: int = 2, idle_timeout: float = 300.0):

        # TODO 1: Initialize pool state with empty connections and in_use sets

        # TODO 2: Store connection parameters (host, port) for creating new connections

        # TODO 3: Set up threading primitives (lock, condition variable for waiting)

        # TODO 4: Start background thread for connection health checking and cleanup

        pass


    @contextmanager

    def borrow_connection(self, timeout: float = 30.0):

        """Borrow connection from pool, automatically return when done."""

        # TODO 1: Acquire lock and look for available connection in pool

        # TODO 2: If no connection available, try to create new one if under max_size

        # TODO 3: If at max_size, wait for connection to be returned (with timeout)

        # TODO 4: Move borrowed connection from available to in_use set
```

```
        # TODO 5: Yield connection to caller for use in with statement
        # TODO 6: In finally block, return connection to pool and notify waiters
        pass


    def _create_connection(self) -> socket.socket:
        """Create new TCP connection to RPC server."""
        # TODO 1: Create TCP socket with appropriate options (SO_REUSEADDR, etc.)
        # TODO 2: Set socket timeout for connection establishment
        # TODO 3: Connect to configured host and port
        # TODO 4: Validate connection is working (send/receive test message)
        # TODO 5: Return established socket ready for RPC communication
        pass


    def _health_check_worker(self):
        """Background thread that validates connection health and manages pool size."""
        # TODO 1: Run infinite loop with periodic sleep (health_check_interval)
        # TODO 2: Acquire pool lock and iterate through available connections
        # TODO 3: Test each connection health (try sending ping or check socket status)
        # TODO 4: Remove dead connections from pool and close their sockets
        # TODO 5: If pool below min_size, create new connections to reach minimum
        pass
```

Async Client Core Implementation

```python
# rpc/extensions/async_client.py

import asyncio

import json

from typing import Dict, Any, Optional

from rpc.core.protocol import create_request_message, serialize_message


class AsyncRPCClient:

    """Asyncio-based RPC client for high-concurrency applications."""



    def __init__(self, host: str, port: int, max_concurrent: int = 1000):

        # TODO 1: Store connection parameters and initialize async state

        # TODO 2: Create semaphore to limit concurrent requests (prevent resource exhaustion)

        # TODO 3: Initialize pending requests dict for tracking futures by request ID

        # TODO 4: Set up connection state (reader, writer, connected flag)

        pass



    async def call(self, method_name: str, *args, timeout: float = 30.0, **kwargs) -> Any:

        """Make async RPC call, returns coroutine that resolves to result."""

        # TODO 1: Generate unique request ID for correlation with response

        # TODO 2: Create request message with method name and parameters

        # TODO 3: Serialize message to bytes for network transmission

        # TODO 4: Acquire semaphore to limit concurrent requests

        # TODO 5: Ensure connection is established (call _ensure_connected)

        # TODO 6: Send serialized request over asyncio stream writer

        # TODO 7: Create Future for this request and store in pending_requests dict

        # TODO 8: Set up timeout task to cancel future if response doesn't arrive
```

```
        # TODO 9: Return awaitable future that caller can await for result
        pass

    async def _ensure_connected(self):
        """Establish asyncio connection if not already connected."""
        # TODO 1: Check if already connected (self._connected flag)
        # TODO 2: If not connected, use asyncio.open_connection to connect
        # TODO 3: Store reader and writer streams for sending/receiving
        # TODO 4: Start background task for reading responses (_response_handler)
        # TODO 5: Set connected flag and handle connection errors appropriately
        pass

    async def _response_handler(self):
        """Background coroutine that reads responses and resolves pending futures."""
        # TODO 1: Run infinite loop reading from asyncio stream reader
        # TODO 2: Read length-prefixed messages from stream (handle partial reads)
        # TODO 3: Deserialize received bytes to response message dict
        # TODO 4: Extract request ID from response to find matching pending future
        # TODO 5: Check if response contains result or error and resolve future
appropriately
        # TODO 6: Remove completed request from pending_requests dict
        # TODO 7: Handle connection errors by rejecting all pending futures
        pass
```

Authentication Middleware Foundation

```python
# rpc/extensions/auth/middleware.py

from typing import Dict, Any, Optional, Set

import jwt

from rpc.core.protocol import create_error_message, ErrorCode


class AuthMiddleware:

    """Authentication middleware that validates JWT tokens and enforces permissions."""



    def __init__(self, jwt_secret: str, permission_store: 'PermissionStore'):

        # TODO 1: Store JWT validation secret and permission store reference

        # TODO 2: Configure JWT validation options (algorithms, expiration checking)

        # TODO 3: Initialize any caching for token validation results

        pass



    def process_request(self, request_message: Dict[str, Any]) -> Optional[Dict[str, Any]]:

        """Process incoming request, return error dict if authentication fails."""

        # TODO 1: Extract authentication token from request (header or dedicated field)

        # TODO 2: If no token present, return authentication required error

        # TODO 3: Validate JWT token signature and expiration using jwt library

        # TODO 4: Extract user identity and roles/permissions from token claims

        # TODO 5: Check if user has permission to invoke the requested method

        # TODO 6: If authorized, add user context to request and return None (success)

        # TODO 7: If unauthorized, return error message with appropriate error code

        pass



    def _validate_jwt_token(self, token: str) -> Optional[Dict[str, Any]]:
```

```
        """Validate JWT token and return claims dict, or None if invalid."""
        # TODO 1: Use jwt.decode() to validate token signature and expiration
        # TODO 2: Check standard claims (exp, iat, etc.) for validity
        # TODO 3: Return decoded claims dict if token is valid
        # TODO 4: Return None if token is expired, malformed, or signature invalid
        # TODO 5: Log authentication failures for security monitoring
        pass

    def _check_method_permission(self, user_claims: Dict[str, Any],
                                 method_name: str) -> bool:
        """Check if authenticated user has permission to call method."""
        # TODO 1: Extract user roles/permissions from JWT claims
        # TODO 2: Query permission store for required permissions for method
        # TODO 3: Check if user has all required permissions (intersection check)
        # TODO 4: Return True if authorized, False if missing permissions
        pass
```

Circuit Breaker Implementation

```python
# rpc/extensions/resilience/circuit_breaker.py

import threading

import time

from enum import Enum

from typing import Callable, Any

from collections import deque


class CircuitState(Enum):

    CLOSED = "closed"      # Normal operation

    OPEN = "open"          # Rejecting requests

    HALF_OPEN = "half_open"  # Testing recovery


class CircuitBreaker:

    """Circuit breaker to prevent cascading failures in RPC calls."""



    def __init__(self, failure_threshold: float = 0.5, window_size: int = 100,

                 timeout_seconds: float = 60.0):

        # TODO 1: Initialize circuit state to CLOSED (normal operation)

        # TODO 2: Set up failure tracking with sliding window (use deque)

        # TODO 3: Store configuration (failure threshold, window size, timeout)

        # TODO 4: Initialize threading primitives for concurrent access

        # TODO 5: Set up timing variables for OPEN -> HALF_OPEN transitions

        pass



    def call(self, func: Callable, *args, **kwargs) -> Any:

        """Execute function with circuit breaker protection."""

        # TODO 1: Check current circuit state and handle OPEN state (immediate failure)
```

```
        # TODO 2: If HALF_OPEN, allow only single test request through
        # TODO 3: If CLOSED or test request, execute function and capture result/exception
        # TODO 4: Record success or failure in sliding window
        # TODO 5: Update circuit state based on recent failure rate
        # TODO 6: Return result or raise exception as appropriate
        pass

    def _update_circuit_state(self):
        """Update circuit state based on recent failure rate."""
        # TODO 1: Calculate current failure rate from sliding window
        # TODO 2: If CLOSED and failure rate exceeds threshold, transition to OPEN
        # TODO 3: If OPEN and timeout expired, transition to HALF_OPEN
        # TODO 4: If HALF_OPEN and test succeeded, transition to CLOSED
        # TODO 5: If HALF_OPEN and test failed, return to OPEN with new timeout
        pass

    def _record_result(self, success: bool):
        """Record success or failure in sliding window."""
        # TODO 1: Acquire lock for thread-safe access to sliding window
        # TODO 2: Add success/failure boolean to sliding window deque
        # TODO 3: If window exceeds max size, remove oldest entry
        # TODO 4: Update failure rate calculation based on new window contents
        pass
```

Milestone Checkpoints for Extensions

Connection Pool Checkpoint: After implementing connection pooling:

1. Run `python -m pytest tests/test_extensions/test_connection_pool.py` - all tests should pass

2. Start test server with `python examples/pooled_server_example.py`
3. Run concurrent client test: `python examples/concurrent_client_test.py`
4. Expected behavior: 100+ concurrent RPC calls complete in under 5 seconds
5. Check that connection count stays reasonable (under max_pool_size)

Async Client Checkpoint: After implementing async support:

1. Run `python -m pytest tests/test_extensions/test_async_client.py`
2. Start server and run: `python examples/async_client_example.py`
3. Expected behavior: 1000 concurrent async calls complete in under 3 seconds
4. Memory usage should remain stable (no memory leaks from futures)

Authentication Checkpoint: After implementing auth middleware:

1. Start secure server: `python examples/secure_server_example.py`
2. Test with valid token: Should allow method calls
3. Test without token: Should return "Authentication required" error
4. Test with expired token: Should return "Token expired" error
5. Test unauthorized method: Should return "Insufficient permissions" error

Circuit Breaker Checkpoint: After implementing circuit breaker:

1. Start unreliable server (randomly fails 60% of requests)
2. Run client with circuit breaker enabled
3. Expected behavior: Initial requests fail, circuit opens, subsequent requests fail fast
4. After timeout, circuit allows test request through
5. If server recovers, circuit closes and normal operation resumes

Glossary

> **Milestone(s):** Foundation for all milestones - provides essential terminology and concepts used throughout message protocol (Milestone 1), server implementation (Milestone 2), and client implementation (Milestone 3)

This glossary defines the key terms, concepts, and technical vocabulary used throughout the RPC Framework design document. Understanding these terms is essential for implementing the framework successfully and communicating about RPC systems effectively.

Mental Model: The Technical Dictionary

Think of this glossary as a specialized technical dictionary for our RPC framework domain. Just as a medical dictionary defines terms like "hypertension" and "myocardial infarction" that have precise meanings in

healthcare, this glossary defines terms like "method proxying" and "connection pooling" that have specific meanings in distributed systems.

Each term represents a concept, pattern, or technique that solves a particular problem in remote procedure calls. When we say "request correlation," we're referring to a specific mechanism for matching responses to their originating requests - not just any kind of matching or correlation. This precision in terminology helps developers communicate clearly about complex distributed system behaviors.

Core RPC Concepts

The foundational concepts that define how remote procedure calls work and what problems they solve.

Term	Definition	Example Usage
JSON-RPC	Lightweight remote procedure call protocol using JSON for message encoding. Defines standard request/response message formats with method names, parameters, and unique identifiers.	"Our framework implements JSON-RPC 2.0 specification for cross-language compatibility."
Remote Procedure Call (RPC)	Programming paradigm where a program can call functions or methods that execute on a different machine or process, appearing as if they were local function calls.	"The client uses RPC to invoke the `calculate_taxes` method running on the accounting server."
Method Proxying	Technique where a client-side proxy object converts method calls into RPC requests transparently, making remote methods appear as local methods to calling code.	"The `RPCProxy` class implements method proxying through Python's `__getattr__` mechanism."
Request ID	Unique identifier assigned to each RPC request that allows correlation between requests and their corresponding responses in async or concurrent scenarios.	"Request ID 'req_12345' ensures the client matches the tax calculation response to the correct calling thread."
Wire Format	The specific serialized message format sent over the network between RPC client and server, including both message content and any framing or delimiting information.	"Our wire format uses 4-byte length prefix followed by JSON-encoded message body."
Method Registry	Server-side mapping that associates function names (strings) with callable handler functions, enabling request dispatch to the appropriate implementation.	"The server's method registry maps 'user.create' to the `create_user_handler` function."
Request Dispatch	Server process of receiving an RPC request, looking up the requested method in the registry, and routing the call to the appropriate handler function.	"Request dispatch failed because 'calculate_orbit' method was not found in the registry."

Network Communication Terms

Technical vocabulary for the underlying network protocols and communication patterns that enable RPC functionality.

Term	Definition	Example Usage
Message Framing	Technique for delimiting individual messages within a continuous byte stream, solving the problem of where one message ends and another begins.	"TCP provides a byte stream, so we need message framing to identify complete RPC messages."
Length Prefix	Message framing approach where each message begins with a fixed-size header indicating the total message size, allowing the receiver to read exactly the right number of bytes.	"Our 4-byte length prefix supports messages up to 4GB, though we limit them to 1MB for safety."
Connection Management	Strategies for handling TCP socket lifecycle including establishment, reuse, health checking, and graceful cleanup to optimize performance and reliability.	"Proper connection management prevents socket descriptor leaks and reduces connection setup overhead."
Connection Reuse	Performance optimization where a single persistent TCP connection carries multiple RPC requests/responses instead of creating new connections for each call.	"Connection reuse reduced average call latency from 15ms to 3ms by eliminating TCP handshake overhead."
Connection Pooling	Advanced connection management pattern maintaining a pool of reusable connections to a server, allowing concurrent requests while limiting total connection count.	"The connection pool maintains 5-20 connections to the database server based on current load."
Thread-per-Connection	Server threading model where each client connection is handled by a dedicated thread, providing isolation and simplicity at the cost of memory overhead.	"Thread-per-connection model supports 1000 concurrent clients with 1000 server threads."
Send All	Network programming pattern ensuring complete message transmission by repeatedly calling send() until all bytes are transmitted, handling partial sends gracefully.	"The `send_all` function prevents message truncation when TCP buffers are full."
Receive All	Network programming pattern ensuring complete message reception by repeatedly calling receive() until the expected number of bytes have been read.	"The `recv_all` function blocks until the complete 1024-byte message is received."

Concurrency and State Management

Terms related to handling multiple simultaneous RPC calls and managing shared state safely across threads.

Term	Definition	Example Usage
Request Tracking	Mechanism for correlating responses with their originating requests, typically using request IDs and pending request data structures.	"Request tracking prevents responses from being delivered to the wrong calling thread."
Response Correlation	Process of matching incoming responses to pending requests using request IDs, enabling proper delivery in concurrent environments.	"Response correlation failed when request ID 'abc123' was not found in pending requests map."
Timeout Handling	Strategy for preventing indefinite blocking when remote servers are unresponsive, including timeout detection and appropriate error response generation.	"Timeout handling raises `RPCTimeoutError` when no response arrives within 30 seconds."
Pending Requests	Client-side data structure tracking active RPC calls that are waiting for responses, typically mapping request IDs to calling thread information.	"The pending requests dictionary grew to 500 entries during the load test spike."
Connection Health Check	Periodic validation that pooled or persistent connections are still usable, detecting dead connections before they cause RPC failures.	"Connection health checks detected 3 stale connections and removed them from the pool."
Race Condition	Timing-dependent bug in concurrent code where the outcome depends on the relative timing of events, often causing intermittent failures.	"The race condition occurred when two threads modified the pending requests map simultaneously."
Deadlock	Circular waiting condition where two or more threads are blocked forever, each waiting for resources held by the others.	"Deadlock happened when thread A held the registry lock while waiting for connection lock held by thread B."

Error Handling and Reliability

Terminology for managing failures, errors, and edge cases in distributed RPC systems.

Term	Definition	Example Usage
Error Propagation	Mechanism for transmitting error information from the server back to the client, including error codes, messages, and diagnostic data.	"Error propagation ensures that database connection failures are reported to the calling client."
Structured Errors	Error objects with consistent fields and diagnostic information, making them easier to handle programmatically and debug effectively.	"Structured errors include error codes, human-readable messages, and optional diagnostic data."
Error Categorization	Classification system for different types of errors (network, serialization, application logic) that enables appropriate handling strategies for each category.	"Error categorization routes network errors to retry logic and application errors to user feedback."
Graceful Degradation	System design principle where the system continues operating with reduced functionality during partial failures rather than complete shutdown.	"Graceful degradation allows read-only operations to continue when the write database is unavailable."
Exponential Backoff	Retry delay strategy that increases the wait time between retry attempts exponentially, preventing overwhelming of failing services.	"Exponential backoff waits 1s, 2s, 4s, 8s between retry attempts for the failing payment service."
Retry Logic	Automatic re-execution strategy for failed operations, typically with limits on attempt count and delays between attempts.	"Retry logic attempts the account lookup operation up to 3 times with exponential backoff."
Circuit Breaker	Failure prevention pattern that stops making calls to an unresponsive service for a period, allowing it time to recover.	"The circuit breaker opened after 10 consecutive failures to the recommendation service."
Transport Error	Network-level error related to TCP connections, socket operations, or message transmission, distinct from application-level errors.	"Transport error indicates the connection was reset by peer during message transmission."
Serialization Error	Error occurring during conversion between objects and their wire format representation, often due to unsupported data types or encoding issues.	"Serialization error occurred when trying to encode the `datetime` object to JSON."
Protocol Error	Error in message format or structure that violates the RPC protocol specification, making the message unparseable or invalid.	"Protocol error: request message missing required 'method' field in JSON-RPC format."

Testing and Quality Assurance

Terms related to validating RPC system behavior and ensuring reliability through testing strategies.

Term	Definition	Example Usage
Integration Testing	Testing approach that validates interactions between multiple components under realistic conditions, ensuring they work together correctly.	"Integration testing revealed that client timeouts were too short for database-heavy operations."
End-to-End Testing	Comprehensive testing that validates complete workflows from client method invocation through network transport to server execution and response.	"End-to-end testing confirmed that the entire user authentication flow works across service boundaries."
Load Testing	Performance validation technique that measures system behavior under expected traffic patterns and user loads.	"Load testing showed the server handles 1000 concurrent clients before response times degrade."
Stress Testing	Testing approach that pushes the system beyond normal operating conditions to identify breaking points and failure modes.	"Stress testing with 10,000 concurrent connections revealed memory leaks in connection handling."
Checkpoint Verification	Testing milestone that validates specific behaviors after implementing each component, providing incremental validation during development.	"Checkpoint verification confirmed that message serialization works before implementing network transport."
Test Double	Testing pattern using fake implementations of dependencies (mocks, stubs, fakes) to isolate components under test.	"The test double simulates network failures to verify client retry behavior."

Performance and Optimization

Technical vocabulary related to optimizing RPC system performance and resource utilization.

Term	Definition	Example Usage
Latency	Time delay between initiating an RPC call and receiving the response, including network transmission and server processing time.	"RPC latency increased from 5ms to 50ms when the database server was moved to a different datacenter."
Throughput	Number of RPC requests the system can process per unit time, typically measured in requests per second.	"System throughput peaked at 10,000 requests per second during the benchmark test."
Connection Overhead	Resource costs associated with establishing and maintaining TCP connections, including memory usage and setup time.	"Connection overhead was reduced by 60% when we switched from per-request to persistent connections."
Serialization Overhead	Performance cost of converting objects to and from wire format, including CPU time and memory allocation.	"JSON serialization overhead accounts for 20% of total RPC call time in our profiling data."
Binary Serialization	Efficient message encoding using binary formats like MessagePack or Protocol Buffers instead of text-based formats like JSON.	"Binary serialization reduced message size by 40% and encoding time by 60%."
Async Support	Non-blocking RPC implementation using asynchronous programming patterns, allowing clients to perform other work while waiting for responses.	"Async support enables the web server to handle 1000 concurrent RPC calls without blocking threads."
Connection Multiplexing	Advanced technique allowing multiple concurrent RPC calls over a single TCP connection, improving efficiency and reducing connection overhead.	"Connection multiplexing allows 100 concurrent requests over just 5 TCP connections."

Security and Authentication

Terms related to securing RPC communications and controlling access to remote methods.

Term	Definition	Example Usage
JWT Authentication	Stateless authentication mechanism using JSON Web Tokens that embed user claims and permissions within the token itself.	"JWT authentication eliminates the need for server-side session storage in our RPC service."
Method-Level Authorization	Fine-grained access control system that validates permissions for individual RPC methods based on user credentials.	"Method-level authorization prevents regular users from calling administrative methods."
TLS Transport Security	Encryption of network traffic using Transport Layer Security protocol, protecting RPC messages from eavesdropping and tampering.	"TLS transport security ensures that sensitive financial data is encrypted during RPC calls."
Token Validation	Process of verifying JWT tokens including signature validation, expiration checking, and claims verification.	"Token validation rejected the request because the JWT had expired 5 minutes ago."
Permission Store	Repository mapping users to their allowed permissions and methods to their required permissions, enabling authorization decisions.	"The permission store indicates that 'admin' role can call 'user.delete' but 'user' role cannot."

Development and Implementation

Terms related to the practical aspects of building and organizing RPC framework code.

Term	Definition	Example Usage
Receptionist Pattern	Design pattern where the server acts as a receptionist, routing incoming requests to the appropriate department (handler function).	"The receptionist pattern centralizes request routing logic in the server's main dispatch method."
Secretary Pattern	Design pattern where the client acts as a secretary, handling communication complexity on behalf of the calling code.	"The secretary pattern hides network failures and retries from application code making RPC calls."
Method Cache	Performance optimization storing references to proxy methods to avoid recreating them on each attribute access.	"The method cache prevents creating new proxy functions every time `client.remote_method` is accessed."
Proxy Object	Client-side object that intercepts method calls and converts them into RPC requests, providing transparent remote method invocation.	"The proxy object makes calling `remote.calculate(x, y)` look identical to local function calls."
Handler Registration	Process of associating function names with callable implementations in the server's method registry.	"Handler registration maps the string 'math.add' to the `addition_handler` function."
Request Dispatcher	Server component responsible for parsing incoming requests and invoking the appropriate registered handler functions.	"The request dispatcher validates the message format before looking up the requested method."
Response Builder	Utility component that constructs properly formatted RPC response messages with results, errors, and correlation IDs.	"The response builder ensures all success responses include the original request ID."

Debugging and Diagnostics

Terms for troubleshooting RPC systems and understanding their runtime behavior.

Term	Definition	Example Usage
Request Tracing	Diagnostic technique that follows a request through all stages of processing, providing visibility into timing and decision points.	"Request tracing showed the tax calculation spent 2 seconds waiting for the database query."
Connection Diagnostics	Monitoring and debugging information about network connections including state, byte counts, and error history.	"Connection diagnostics revealed that 30% of connections were being closed by network timeouts."
Latency Tracking	Performance monitoring that measures and records timing information for RPC calls, enabling identification of slow operations.	"Latency tracking identified that user lookup calls take 500ms on average during peak hours."
Error Correlation	Diagnostic technique for linking related errors across client and server logs using request IDs and timestamps.	"Error correlation matched the client timeout with the server's database deadlock exception."
Socket State Monitoring	Tracking the current state and health of TCP sockets used for RPC communication.	"Socket state monitoring detected 15 connections in CLOSE_WAIT state, indicating a resource leak."

Implementation Guidance

This section provides practical guidance for implementing the concepts defined in this glossary using Python.

Core Implementation Classes

The following table shows the main classes you'll implement and their essential attributes:

Class Name	Key Attributes	Primary Responsibility
`RPCServer`	`host`, `port`, `registry`, `server_socket`, `running`	Accept connections and dispatch requests
`RPCClient`	`host`, `port`, `default_timeout`, `_socket`, `_pending_requests`	Send requests and manage responses
`MethodRegistry`	`methods`, `lock`, `descriptions`	Store and lookup registered methods
`RPCProxy`	`_client`, `_timeout`, `_method_cache`	Provide transparent method call interface
`ConnectionPool`	`host`, `port`, `max_size`, `connections`, `pool_lock`	Manage reusable connections efficiently

Error Hierarchy Implementation

Create a comprehensive error class hierarchy to support structured error handling:

PYTHON

```python
class RPCError(Exception):

    """Base class for all RPC-related errors."""

    def __init__(self, message: str, error_code: int, error_data: any = None):

        super().__init__(message)

        self.message = message

        self.error_code = error_code

        self.error_data = error_data


class RPCProtocolError(RPCError):

    """Error in message format or protocol compliance."""

    def __init__(self, message: str, raw_data: bytes, parse_stage: str):

        super().__init__(message, PARSE_ERROR)

        self.raw_data = raw_data

        self.parse_stage = parse_stage


class RPCTransportError(RPCError):

    """Network-level communication error."""

    def __init__(self, message: str, host: str, port: int, operation: str):

        super().__init__(message, TRANSPORT_ERROR)

        self.host = host

        self.port = port

        self.operation = operation


class RPCTimeoutError(RPCError):

    """Request timeout error."""

    def __init__(self, message: str, timeout_seconds: float, elapsed_seconds: float):

        super().__init__(message, TIMEOUT_ERROR)

        self.timeout_seconds = timeout_seconds
```

```
        self.elapsed_seconds = elapsed_seconds


class RPCMethodError(RPCError):
    """Error during method execution."""
    def __init__(self, message: str, method_name: str, params: list, execution_phase: str):
        super().__init__(message, INTERNAL_ERROR)
        self.method_name = method_name
        self.params = params
        self.execution_phase = execution_phase
```

Standard Error Codes

Define error codes following JSON-RPC specification:

```python
class ErrorCode:
    PARSE_ERROR = -32700      # Invalid JSON received
    INVALID_REQUEST = -32600   # JSON is not a valid request object
    METHOD_NOT_FOUND = -32601  # Method does not exist
    INVALID_PARAMS = -32602    # Invalid method parameters
    INTERNAL_ERROR = -32603    # Internal JSON-RPC error
    TRANSPORT_ERROR = -32000   # Network/transport error
    TIMEOUT_ERROR = -32001     # Request timeout
    CONNECTION_ERROR = -32002  # Connection failed
```

Message Utilities

Essential functions for message handling:

```python
import json

import uuid

import struct

import socket

from typing import Dict, Any, Optional


def generate_request_id() -> str:

    """Generate unique request identifier using UUID4."""

    return str(uuid.uuid4())


def create_request_message(method: str, params: list, request_id: str) -> Dict[str, Any]:

    """Create JSON-RPC 2.0 request message."""

    return {

        "jsonrpc": "2.0",

        "method": method,

        "params": params,

        "id": request_id

    }


def create_response_message(request_id: str, result: Any) -> Dict[str, Any]:

    """Create JSON-RPC 2.0 success response message."""

    return {

        "jsonrpc": "2.0",

        "result": result,

        "id": request_id

    }


def create_error_message(request_id: str, code: int, message: str, data: Any = None) ->
Dict[str, Any]:
```

```
    """Create JSON-RPC 2.0 error response message."""
    error_obj = {"code": code, "message": message}
    if data is not None:
        error_obj["data"] = data

    return {
        "jsonrpc": "2.0",
        "error": error_obj,
        "id": request_id
    }

def serialize_message(message: Dict[str, Any]) -> bytes:
    """Convert message dictionary to JSON bytes."""
    json_str = json.dumps(message, separators=(',', ':'))
    return json_str.encode('utf-8')

def deserialize_message(data: bytes) -> Dict[str, Any]:
    """Parse JSON bytes to message dictionary."""
    try:
        json_str = data.decode('utf-8')
        return json.loads(json_str)
    except (UnicodeDecodeError, json.JSONDecodeError) as e:
        raise RPCProtocolError(f"Failed to deserialize message: {e}", data, "json_parsing")
```

Network Helper Functions

Low-level socket operations with proper error handling:

```python
import time

import select


MAX_MESSAGE_SIZE = 1048576  # 1MB maximum message size


def send_all(sock: socket.socket, data: bytes, timeout: float = 30.0) -> None:

    """Send complete message handling partial sends."""

    sock.settimeout(timeout)

    bytes_sent = 0

    total_bytes = len(data)

    start_time = time.time()



    while bytes_sent < total_bytes:

        if time.time() - start_time > timeout:

            raise RPCTimeoutError(f"Send timeout after {timeout}s", timeout, time.time() -
start_time)



        try:

            sent = sock.send(data[bytes_sent:])

            if sent == 0:

                raise RPCTransportError("Socket connection broken during send",

                                        sock.getpeername()[0], sock.getpeername()[1], "send")

            bytes_sent += sent

        except socket.error as e:

            raise RPCTransportError(f"Send failed: {e}",

                                    sock.getpeername()[0], sock.getpeername()[1], "send")


def recv_all(sock: socket.socket, size: int, timeout: float = 30.0) -> bytes:

    """Receive exact byte count with timeout."""
```

```
    sock.settimeout(timeout)

    chunks = []

    bytes_received = 0

    start_time = time.time()



    while bytes_received < size:

        if time.time() - start_time > timeout:

            raise RPCTimeoutError(f"Receive timeout after {timeout}s", timeout, time.time()
- start_time)



        try:

            chunk = sock.recv(min(size - bytes_received, 4096))

            if not chunk:

                raise RPCTransportError("Socket connection closed during receive",

                                        sock.getpeername()[0], sock.getpeername()[1], "recv")

            chunks.append(chunk)

            bytes_received += len(chunk)

        except socket.error as e:

            raise RPCTransportError(f"Receive failed: {e}",

                                    sock.getpeername()[0], sock.getpeername()[1], "recv")



    return b''.join(chunks)


def send_message(sock: socket.socket, message_bytes: bytes, timeout: float = 30.0) -> None:

    """Send length-prefixed message."""

    if len(message_bytes) > MAX_MESSAGE_SIZE:

        raise RPCProtocolError(f"Message too large: {len(message_bytes)} bytes",

                               message_bytes, "size_check")
```

```
    # Send 4-byte length prefix in network byte order
    length_prefix = struct.pack('!I', len(message_bytes))
    send_all(sock, length_prefix, timeout)
    send_all(sock, message_bytes, timeout)

def recv_message(sock: socket.socket, timeout: float = 30.0) -> bytes:
    """Receive length-prefixed message."""
    # Read 4-byte length prefix
    length_data = recv_all(sock, 4, timeout)
    message_length = struct.unpack('!I', length_data)[0]

    if message_length > MAX_MESSAGE_SIZE:
        raise RPCProtocolError(f"Message too large: {message_length} bytes",
                               length_data, "length_check")

    # Read message body
    return recv_all(sock, message_length, timeout)
```

File Structure Recommendation

Organize your RPC framework code using this structure:

```
rpc_framework/
├── __init__.py                # Package exports
├── protocol.py                # Message formats and serialization
├── server.py                  # RPCServer implementation
├── client.py                  # RPCClient and RPCProxy implementation
├── registry.py                # MethodRegistry implementation
├── errors.py                  # Error class hierarchy
├── transport.py               # Network helper functions
├── utils.py                   # Utility functions
└── tests/
    ├── __init__.py
    ├── test_protocol.py       # Message protocol tests
    ├── test_server.py         # Server component tests
    ├── test_client.py         # Client component tests
    └── test_integration.py    # End-to-end tests
```

This structure separates concerns cleanly while keeping related functionality together. Each module has a clear responsibility and can be developed and tested independently.