J. J. Strossmayer University of Osijek

Faculty of Electrical Engineering, Computing and Information Technologies Osijek
Kneza Trpimira 2b
HR-31000 Osijek
www.ferit.unios.hr

Lab Assignment 5:

Remote Procedure Calls in Distributed Computing Systems

Academic year 2025/2026.

Luka Loina, univ. mag. ing. comp.

When One Server Went Missing: Learning RPC's Real Challenges

In a real-world distributed system scenario documented by Google's Site Reliability Engineering research, a seemingly routine configuration change exposed one of the fundamental challenges with remote procedure calls: the inability to distinguish between different failure modes.

An on-call engineer managing a backend service noticed the system was approaching quota limits. To allocate additional capacity, they modified a critical configuration that was intended to add a backend server in one region. However, due to a subtle misunderstanding in the configuration tooling, the change instead ***removed an existing backend server****, fragmenting the service's traffic distribution. For 20 seconds, requests that would normally load-balance across multiple servers were now concentrated on the remaining infrastructure.*

Because the engineer assumed the configuration change was safe - after all, they had made similar changes before - they didn't actively monitor the rollout in real-time. The first indication that something had gone wrong came from automated alerting systems reporting a spike in error rates across dependent services. The error messages were cryptic: "permission-denied" exceptions from backend calls. But here's the problem specific to RPC systems: from the client's perspective, an error could mean many things. Did the request ever reach the server? Did the server process it and then crash? Did the network drop the reply? These questions were hidden from the callers making the RPCs.

*To debug the issue*Remote Procedure Calls*, engineers had to perform a breadth-first search through aggregated logs across multiple dependent services, correlating timing data with RPC latency metrics. It took hours of investigation to realize that the permission errors directly correlated with the moment of the configuration change - specifically, when one server was removed, more traffic routed to already-problematic servers, and the cascade amplified from there.*

Exercise 5: Remote Procedure Calls in Distributed Computing Systems

1 Introduction

In a traditional program running on a single machine, one part of the code calls another using a normal procedure or method call: arguments are pushed on a stack, the callee runs, and a result is returned. In a distributed system, useful functionality and data are often spread across multiple machines, yet programmers would still like to structure their code using clear calls and returns instead of manually sending and receiving messages. RPC was proposed to bridge this gap by making **remote** interactions look as much as possible like ordinary local calls. The key idea is *transparency*: when a client calls a remote procedure, the code should look almost identical to a local call, even though the call actually crosses process and machine boundaries via a network. Under the hood, the RPC system takes care of packaging up arguments, transmitting them over the network, invoking the procedure on the server, and returning results. This allows programmers to design distributed applications in terms of well-defined interfaces instead of low-level message protocols.

1.1 Learning Objectives

By the end of this lab, you will be able to:

- Understand mechanics and usage of RPC.
- Implement simple RPC system.

2 Background

Remote Procedure Calls serve several critical purposes in modern distributed computing environments. Understanding these reasons helps explain why RPC remains a foundational technology despite the emergence of alternative communication paradigms. RPC fundamentally reduces the complexity of distributed systems development by abstracting away network communication details. Developers can write code that calls remote procedures using the same syntax and semantics as local function calls, eliminating the need to manually construct and parse network messages. This abstraction allows programmers to focus on business logic rather than low-level networking protocols, making development faster and less error-prone.

Remote procedure calls enable **modular design** by allowing different components of an application to be developed independently and interact seamlessly through standardized procedure calls. Systems built with RPC maintain **loose coupling**, meaning components can be updated, replaced, or modified independently without affecting the entire application. This architectural flexibility is particularly valuable in large, complex systems where different teams may develop separate components.

Distributed applications built with RPC can easily scale by adding more systems or resources to handle increased workloads without requiring significant architectural changes. RPC enables **resource sharing**, allowing a computationally expensive service to be offloaded to a dedicated server, freeing up resources on client systems. This distribution of work across multiple machines improves overall system efficiency and allows applications to handle growing demands.

By distributing components across multiple systems, RPC-based applications can be designed to handle failures more effectively. If one system fails, other systems can continue operating while the failed system is replaced or repaired, preventing complete system outages. This distributed architecture inherently provides better resilience compared to monolithic applications.

Many remote procedure call implementations support multiple programming languages and platforms, allowing developers to build heterogeneous distributed systems where different components use the best tools for their specific needs without compatibility concerns. This interoperability is crucial in modern environments where organizations typically use diverse technology stacks.

Remote procedure calls can offer **better performance than alternative approaches** like REST for certain use cases, especially when using binary protocols and compact serialization formats. This makes RPC particularly suitable for high-throughput, low-latency systems where minimizing network overhead is critical, such as in real-time trading systems or high-frequency data processing.

However, the transparency that RPC provides comes with important caveats. Network communication introduces latencies and potential failures that simply do not exist in local procedure calls. A remote call might time out, the server might be unavailable, or network packets might be lost. These failure modes require explicit handling through mechanisms like timeouts, retries, and error callbacks - concerns that are largely absent in traditional programming. Additionally, RPC systems must address questions of consistency and ordering when multiple calls interact with shared state across machines, and must decide on semantics such as whether a procedure call should be executed at-most-once or at-least-once if failures occur. These complexities explain why RPC, despite its elegance as a conceptual bridge between local and distributed programming, remains a specialized tool requiring careful consideration of its tradeoffs.

Remote procedure call operates on a **client-server model**, where a client initiates the request and a server executes the requested operation. Remote procedure call starts by client invoking **client stub,** proxy procedure that represents remote procedure locally. The stub packages (or "marshals") the procedure parameters into a network message format suitable for transmission. Depending on specific implementation of RPC, after arguments are packaged client stub ether sends the data to server or passes it to **RPC Runtime** component. RPC Runtime component manages all network communication between client and server. It is responsible for message transmission, retransmission, acknowledgment, routing, encryption, and handling the delivery of results back to the client stub. On the server side, **server stub** receives the incoming message and unpacks (or "unmarshals") the parameters, reconstructing the original arguments needed to execute the remote procedure. The server hosts the actual implementation of the remote procedure and executes it based on the received request, returning the results to the server stub for transmission back to the client. After the procedure executes, the server stub packages the results and sends them back through the RPC runtime.

A typical RPC call follows these steps:

1. The client application calls what appears to be a normal function call in the client stub, providing the necessary parameters
2. The client stub marshals (serializes) the parameters into a message format
3. The RPC runtime transmits this message to the server machine over the network
4. The server-side RPC runtime receives the message and passes it to the server stub
5. The server stub unmarshals (deserializes) the parameters
6. The actual procedure is executed on the server with the provided arguments
7. The server returns the results to the server stub
8. The server stub marshals the results into a message

9. The RPC runtime sends the results back to the client
10. The client stub receives and unmarshals the results
11. The client application receives the results and execution resumes

2.1 Key Challenges: Failures in Distributed Systems

The critical distinction between RPC and local procedure calls emerges when failures occur. Unlike local calls, where either the function executes or the program crashes, distributed systems must contend with partial failures - where some components fail while others continue operating.

Three scenarios are fundamental to RPC design:

- **Request Message Loss.** The network may drop the client's request before it reaches the server.
- **Server Failures.** The server may crash before the request arrives, while executing the request, or after computing the response but before sending it.
- **Response Message Loss.** The network may drop the server's response on its way back to the client.

These failures create semantic ambiguity: if a client doesn't receive a response, it cannot determine whether the server executed the request, whether the network failed, or whether the server crashed. This ambiguity is fundamental to distributed computing and cannot be completely resolved.

2.2 RPC Semantics: Defining Correctness

To handle failures, RPC systems must commit to one of three semantic guarantees:

At-Least-Once Semantics. If the client receives a response, the request executed at least once (possibly multiple times). If no response arrives, the request may or may not have executed. This is the simplest to implement but requires that operations be idempotent (safe to execute multiple times), such as reading data or computing a maximum value. It is used in stateless systems like DNS and early implementations of Network File System (NFS).

At-Most-Once Semantics. If the client receives a response, the request executed exactly once. If no response arrives, the request may or may not have executed. This semantic is implemented by having clients tag each request with a unique sequence number and having servers track which requests have been executed, caching responses to avoid re-execution on duplicates. This approach is more complex but enables stateful operations.

Exactly-Once Semantics. If a response is received, the request executed exactly once. If no response arrives, the client retransmits indefinitely until it receives a response. This provides the strongest guarantee but is often impractical because it may block indefinitely if the server has truly crashed permanently.

2.3 Serialization and Data Representation

Remote procedure calls require converting in-memory data structures into byte sequences for network transmission - a process called serialization. Primitive types (integers, floats, booleans) serialize straightforwardly. Complex data structures require more careful handling:

Pointers. A pointer value itself is meaningless on a remote machine—the memory address it references exists only in the caller's address space, not on the remote server. RPC systems cannot simply transmit raw pointer values across the network. Instead, they employ one of three strategies. The most common approach is to **marshal** (serialize) the data that the pointer references, not the pointer itself. For scenarios requiring persistent server-side state—such as open file handles, database connections, or multi-step transactions—the RPC system uses **opaque references** (or handles). Rather than transmitting data, the server creates the actual object and returns a unique reference identifier (such as a file handle number) to the client. The client includes this reference in subsequent RPC calls, and the server maintains an internal table mapping references to objects. This allows the server to maintain state across multiple related calls. Modern RPC frameworks take a simpler approach: **pointer parameters are not permitted**. All arguments and return values must be serializable value types—primitive types, strings, arrays, and structs composed of these. This eliminates the ambiguity and complexity of pointer handling at the cost of less flexibility for representing certain complex scenarios. The three strategies represent different trade-offs. Data **marshaling** (Strategy 1) is simple but stateless. **Opaque references** (Strategy 2) enable stateful operations and are widely used in production systems like NFS and Java RMI. **Disallowing pointers** (Strategy 3) provides maximum simplicity and language-agnostic interoperability, making it the standard in modern RPC frameworks.

Complex Objects. Structs, arrays, and nested data structures must be flattened into a linear byte sequence. Modern RPC frameworks use standardized serialization formats like Protocol Buffers, JSON, or language-native serialization mechanisms.

Type Information. The RPC framework must preserve type information to correctly deserialize bytes on the receiving end. This is why RPC systems typically require interface definitions specifying the names, argument types, and return types of all callable procedures - usually expressed in an Interface Definition Language (IDL).

3 Comparison and Experiments

In this section, you will design and perform experiments, collect data, and analyze the results to better understand trade-offs of different approaches to data replication.

3.1 Goal

Your objective is to:

- Create simple RPC system.
- Implement key-value using your RPC system.

3.2 Student Tasks

Assignment 1. **Implement RPC by using structured data communicated over the network.**

1. Start with base system for RPC.
2. Implement stub function for set and get operations for key-value store on server.
    - You can store data for keys and values in-memory by using unordered map (“std::unordered_map<std::string, std::string>”).
    - You can modify RPC_message structure and RPC_procedure enumeration in rpc.h file to add parameters and values you need for your RPC calls.
3. Implement stub functions for set and get operations on client.
4. Test your implementation.

Assignment 2. **Implement buffer based marshaling and unmarshaling of RPC request and response.**

1. Replace usage of RPC_message structure in your RPC implementation with flat buffer.
   - Use **fixed-size type**, like `uint32_t`, `int32_t` or `uint16_t` from `<stdint.h>`, for your buffer instead of platform-dependent types like `int` or `long`. This ensures the same size and representation across all architectures.
2. Test implementation of set and get functions.

Assignment 3. **Expand marshaling/unmarshaling logic to handle arbitrary sized data in requests and responses.**

1. Expand buffer based marshaling and unmarshaling logic from Assignment 2. to handle arbitrary sized data without wasting memory and network bandwidth.

3.3 Final Deliverables

By the end of this section, you should have:

1. RPC based key-value store with buffer based marshaling and unmarshaling