2024 IEEE/ACIS 22nd International Conference on Software Engineering Research, Management and Applications (SERA) | 979-8-3503-9134-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/SERA61261.2024.10685565

SpeciServe, a gRPC Infrastructure Concept

Chase Carthen*, Araam Zaremehrjardi*, Zachary Estreito*, Alireza Tavakkoli*, Frederick C. Harris* Jr., Sergiu M. Dascalu*
**Computer Science and Engineering*
**University of Nevada, Reno*
Reno, Nevada
Email: {ccarthen, azaremehrjardi, zestreito, tavakkol}@unr.edu,
{fred.harris, dascalus}@cse.unr.edu

***Abstract*—Smart city projects require data to be transferred from one destination to the next using a number of different network protocols. The data pipelines involved in these smart city projects often have limited bandwidth or compute resources due to the low power nature of most embedded hardware. The data transferred between devices in these types of embedded systems are often structured in non-standard data schemata. Remote procedure calls (RPC) are implemented to transfer data between devices and switching between RPC implementations can be tricky due to the lack of standardization. There is no guarantee that an existing data schema will work with a different RPC implementation. This makes it difficult for a researcher or system developer to benchmark and compare different RPC implementations. In this paper, a conceptual infrastructure named SpeciServe is introduced where gRPC is used as a communication backbone due its support for flatbuffers and multiple server modes. Multiple software services are described to allow for dissimilar RPC implementations to be run in parallel. This system is intended to allow for researchers in machine learning, smart cities, and Internet of Things (IoT) to be able test different versions of RPCs and provide support for system developers to define the functions of an edge service.**

***Index Terms*—gRPC, data streaming, IoT, data transfer, big data, smart city, data pipeline, edge computing**

I. Introduction

There has been a notable rise in the deployment of IoT systems and the development of IoT management software. This is partly due to the sudden onset of machine learning techniques being integrated with IoT systems and development of low powered devices along with new IoT network protocols to enable communication between these devices. Some notable areas that are seeing growth with these recent changes are smart cities, agriculture, and industry. These applications would benefit greatly from a standardized networking infrastructure to handle the flow of data between devices. Because this networking infrastructure must accommodate low power devices and networks with constrained resources, software developers have created protocols such as gRPC, DDS, MQTT, and other IoT application protocols to handle communication. Developing an infrastructure with these networking protocols requires creating a set of RPCs to handle the logic of the system's various applications, and changing the underlying implementation of these RPCs can be challenging for developers.

Upgrading or changing the underlying RPCs of a distributed system's networking architecture often tasks software developers and system designers with testing, benchmarking, and comparing the new RPC implementations to previous versions. When testing these RPCs, it must be ensured that the new RPC implementation performs better or at least similar to the previous version. In some cases, software developers may wish to test multiple implementations of an RPC to see which one performs best. For example, one may wish to test various RPC implementations with a distributed machine learning system or with a distributed system that requires heavy file compression. Development and testing of new RPC implementations can be difficult to tackle as a system scales up and evolves.

In this paper, we present a concept of a gRPC infrastructure where software developers can register their RPCs over gRPC, REST, MQTT, and other network protocols. All other interactions within the system utilize gRPC as the primary communication backbone. This system consists of an RPC server that stores definitions of different RPC implementations and the schema of the data that is communicated across these RPCs. The utilization of these RPCs would be handled by an implementation server that would utilize a registration service in the RPC server. Important information about the RPCs and data schemas associated with the system will be tracked by a metadata service. Data transfers between implementation clients and implementation servers are handled by a data flow service over gRPC. All data originating from the data flow service will be cached to provide access to the data for multiple services. Figure 1 demonstrates a simplified view of the data flowing from the user of this system all the way to the implementation server.

The rest of the paper is structured as follows. Section II covers the background and related works for this paper. The systems design and implementation is described in Section III. Discussion of this paper's prototype is included in Section IV. Finally, the overall impact of this paper is discussed in Section V.

II. Background & Related Works

A. RPC Communication

Remote Procedure Call (RPC) communication was first conceptualized in the 1970s. Scalability quickly became a concern, and efforts to address the scalability of RPC were undertaken in the 1980s [1]. Given the widespread use of microservices and the ubiquitous rise of distributed computing, the paradigm of RPC communication has both matured and

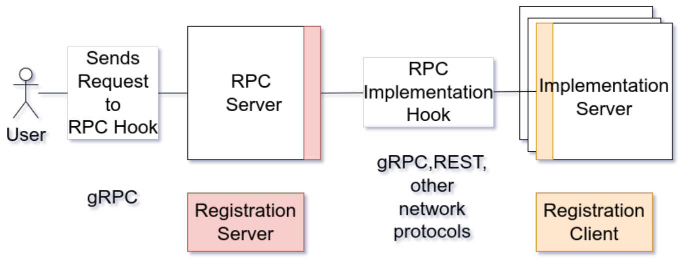

Fig. 1. Conceptual data flow from the user through the RPC server all the way to the implementation server.

shifted from its original purpose of facilitating communication between clients and servers on external devices.

Specifically, RPC communication has been explored and generalized as an interprocess communication (IPC) tool for client's internal processes and distributed computing environments by the use of an interface definition language (IDL). The usage of IDLs has brought benefits in the form of an improved software developer experience by allowing code generation for RPC stubs. This may, however, incur a performance penalty depending on system requirements [2]. That said, there has been recent efforts to optimize for these performance shortcomings [3]–[5].

One solution (Wang et al., 2021) presents a immutable shared memory space where messages are passed between processes and scheduler to manage said memory space [3]. Furthermore, HatRPC further expands on using memory spaces effectively by incorporating remote direct memory access (RDMA) with Apache Thrift to allow a generalized RPC framework that implements granular optimizations for throughput and latency based upon RPC message characteristics [5]. By developing an abstraction layer between Apache Thrift and RDMA, the authors were able implement specific RMDA optimizations during compile time for client and server stubs. Additionally, specialized RPC solutions have been developed that employ specific optimizations for distributed tasks [4]. PyTorch RPC is a example of a specialized RPC framework designed to improve the scaling of key phases of a federated learning pipeline by leveraging automatic tensor optimizations when tensor-to-tensor communication is best suited and uses low-level hardware communication pipelines (e.g. NVLink) in the form of channels for IPC tasks.

B. gRPC

gRPC is an open-source RPC framework developed by Google in 2015. It has gone through minor revisions since its inception with the most recent of the protocol adopting HTTP 2 in 2020 [6]. gRPC is composed of three major aspects: serialization, the ”proto” compiler, and the RPC communication itself. Although gRPC offers the capability of using JSON for message serialization, usage with Google's Protocol Buffers is preferred with support for flat buffers.

While capable of transmitting JSON data, gRPC serialization uses Protocol Buffers by default. Protocol Buffers make use of schema files called ”proto” files that are used by the ”protoc” compiler to generate code that serializes and deserializes data [7]. The compiler is capable of generating for a wide range of programming languages such as C++, Java, C#, Python, and others. gRPC builds on top of this compiler methodology by allowing services to be defined and compiled from proto files using the protoc compiler via a plugin. gRPC services communicate with defined remote procedure call functions that include a request (called ”proto request”) and response (called ”proto response”) that define the input and output of the remote procedure call. gRPC offers four modes of communication: unary messages between client and server, client streaming to server, server streaming to client, and bidirectional streaming. Once services and their respective remote procedures are defined, the compiler generates both server and client interfaces dubbed ”stubs” in which users implement the request and response functions for their applications. gRPC has seen use for communication between microservices, a interprocess communication tool on embedded platforms, and for internetwork communication by using Software Defined Networks.

gRPC has also been implemented on a edge to client test environment for offloading image computation using the OpenCV library [8]. The authors designed the testbed environment to have a smartphone capture a image and then uses gRPC to transfer the image to a development board hosting a gRPC server that applies OpenCV filters to the image. The development of the testbed environment was for the purpose of evaluating the performance of offloading computational data across different environments and constraints such as different programming languages, operating systems, and system architectures. The authors compared the local execution times and offloading (to the server) times with the dependent variables being the platform used (Debian versus Android) and programming languages. Throughout the testing gRPC was used for both interdevice and interprocess communication on the server for a matrix multiplication and image processing application. The authors concluded from their testing that gRPC had distinct performance advantages while also providing the benefit of being an interprocess communication (IPC) tool. The IPC benefits are due to gRPC being able to create Protocol buffers for different programming languages rather easily to facilitate passing messages between applications [8]. REST, gRPC, and WebSocket were evaluated by Weerasinghe and Perera for use within a distributed environment on Amazon Web Services and they found that gRPC has the best results for interservice communication in all metrics [9]. Additionally, Park et al. present a shared backend architecture with a strong emphasis on interprocess communication for using machine learning applications on a resource constrained device such as a edge device [10]. The authors considered the use of gRPC as the backbone for allowing communication across services, providing isolation, and allocating resources for such services but reconsidered due to performance evaluations. Specifically, the authors noted a high latency and memory footprint for each new gRPC connection for their setup requirements. Thus, the authors opted to use Oracle's System V IPC rather than gRPC

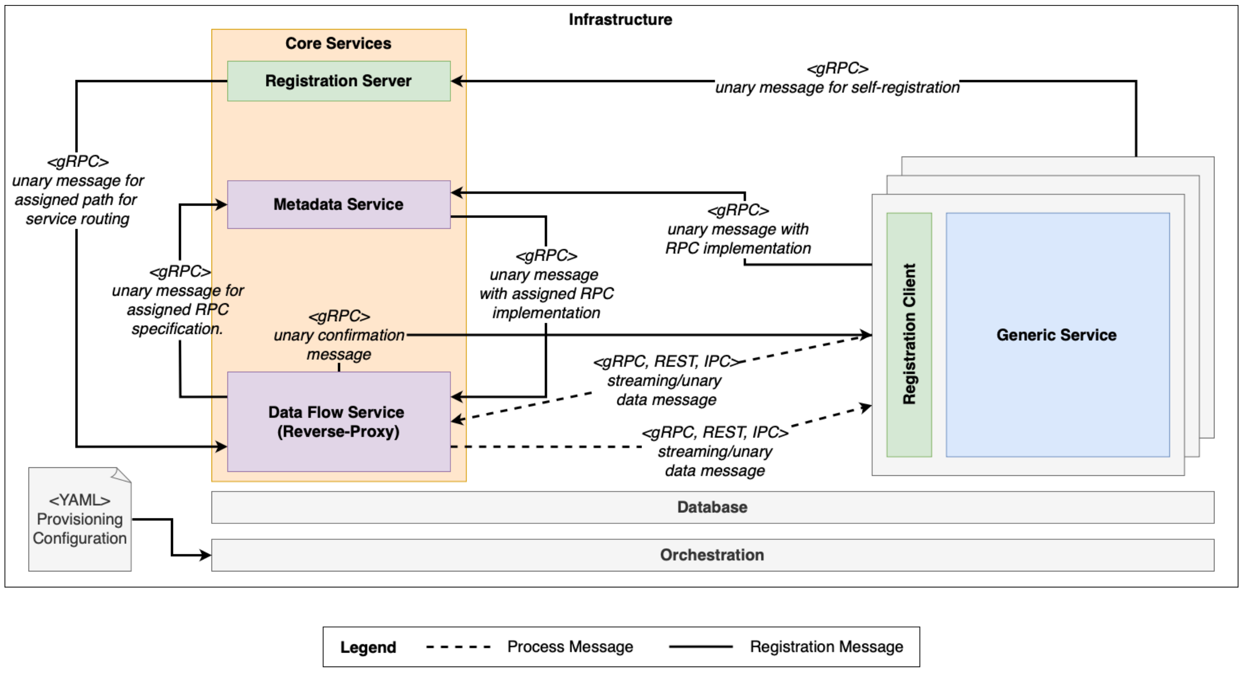
Infrastructure
Core Services
Registration Server
Metadata Service
Data Flow Service (Reverse-Proxy)
<gRPC> unary message for self-registration
<gRPC> unary message for assigned path for service routing
<gRPC> unary message with RPC implementation
<gRPC> unary message with assigned RPC implementation
<gRPC> unary message for assigned RPC specification.
<gRPC> unary confirmation message
<gRPC, REST, IPC> streaming/unary data message
<gRPC, REST, IPC> streaming/unary data message
Registration Client
Generic Service
<YAML> Provisioning Configuration
Database
Orchestration
Legend
Process Message
Registration Message

Fig. 2. The conceptual SpeciServe system as a whole.

for their shared back-end configuration.

III. Design

SpeciServe will consist of several data different services with different services being available in the RPC Server to allow for other services that are implementations to the RPCs stored in the RPC server. Figure 2 shows a figure of the system as a whole and its individual components. Communication between different services will be handled by IPC within the RPC server to ensure fast communication between the services. SpeciServe's RPC architecture begins with the creation of a yaml configuration file with the following responsibilities: defining core services paths and generic services with their respective RPC functions. Furthermore, the yaml configuration file is responsible for establishing the operating parameters of the core services that will be used in an orchestration applications such as Kubernetes and Docker-Compose. Core service paths are the statically bound in the infrastructure, while generic services make themselves known using the registration service and are dynamically bound. By providing a provisioning configuration, the necessary forethought for overall system interoperability is taken care through the metadata and registration service.

Within the RPC server a metadata service is used to maintain what RPC clients are connected to a RPCs, data schemas, and the RPCs within the whole system. The metadata service will act as a service that allows for system developers to see what a given system can support and what can be added. Within the metadata service it will encapsulate what gRPC server modes are used within connections. This will allow for the system to support different network infrastructures where certain network modes may be advantageous. Data schemas within the metadata server will be stored as flatbuffers or protocol buffers to ensure strong typing within the system. RPCs will be stored as a yaml file that will use any data schemas and predefined types within flatbuffers and protocol buffers.

In order to allow for different implementations to connect to the RPC server, a registration service will handle adding new implementations to the RPC server and validating whether the implementation matches any present RPCs stored within the service. It will also update the metadata service with the details of the client being added to the service as it is added. This service will be responsible for adding new services to the system and ensures that it matches the current data model present in the metadata service. A thin client in tandem with this registration client will make it possible for implementation server or generic service to connect to the RPC server. This registration process can occur over REST, IPC, or other network protocols.

All RPC calls and the result of those are handled by a dataflow service. This service is responsible for handling the traffic between an outsider user and the generic services over gRPC. As data is sent out and received by the dataflow service, the data will be stored into a database for caching and allowing the user to retrieve previous results. This database may be utilized by external users or by other implementations to get information about past messages.

IV. Discussion

This SpeciServe conceptual framework is designed to allow for developers to test and implement their edge services with ease. However, this conceptual framework may not be as fast as other RPC implementations such as Kogias et al.'s implementation where their RPC implementation is designed to speed up several unique types of services across different programs [11]. Although, there exists performance disadvantages within the conceptualized framework we rather aim to emphasize compatibility for infrastructures that may incorporate different protocols for communication. By prioritizing compatibility, we improve the developer experience and system extensibility at the cost of system efficiently. These goals are achieved through the registration server and client in which allow the conceptualized system to receive and transmit these new data streams during runtime by using services' provided data schema. Furthermore, our implementation is meant to serve as a way for researchers in different fields to be able to test their own implementations. In effect this conceptual system will be more suited for near-realtime applications. One field that may benefit greatly from this is machine learning where multiple different versions of an ML solution are often compared against each other. This conceptual framework could be used as a tool for different researchers to test different algorithms with the same set goal. Beyond testing different versions of RPCs, this system could allow for testing similar sensors from different manufacturers.

gRPC is proposed as a backbone to this system as it allows for more flexibility and scalability in terms of the data format and its underlying features. It has the capability for more threads to be added if the demand of the networking increases. It has several different server modes unary, bidirectional, client send/receive, and server send/receive modes. Flatbuffers, a format supported by gRPC to be sent, has an option for data to be compressed within the format itself. Both flatbuffer and protocol buffers impose a strongly typed data schema that most systems are designed for. These different server modes allow for versatility in that the RPC Server can be connected to generic services in many different types of networks.

V. Conclusion

In the future we plan to implement a proof of concept based on this proposed infrastructure. While the implementation of this conceptual framework will have more performance bottlenecks than regular RPC implementations or other similar implementations, it may prove useful for system developers and researchers to test, benchmark, and develop their algorithms. This concept may be worth trying in many different types of networks including in the cloud, at the edge, or both. Given that this conceptual framework is suitable for testing and designing different RPCs. The SpeciServe framework will be used to develop several example applications and benchmark them against existing implementations. Along with the development of example these applications, we will test interfacing and interacting with devices of differing designs when considering high latency protocols like LoRa.

Acknowledgment

This material is in part upon work supported by the National Science Foundation under grants OAC-2209806, OIA-2019609, and OIA-2148788. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of The National Science Foundation.

References

[1] B. Bershad, D. Ching, E. Lazowska, J. Sanislo, and M. Schwartz, "A remote procedure call facility for interconnecting heterogeneous computer systems," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 8, pp. 880–894, 1987.

[2] N. Feske, "A case study on the cost and benefit of dynamic rpc marshalling for low-level system components," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 4, pp. 40–48, Jul. 2007.

[3] S. Wang, B. Hindman, and I. Stoica, "In reference to rpc: It's time to add distributed memory," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '21, Ann Arbor, Michigan: Association for Computing Machinery, 2021, pp. 191–198.

[4] P. Damania, S. Li, A. Desmaison, *et al.*, "Pytorch rpc: Distributed deep learning built on tensor-optimized remote procedure calls," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[5] T. Li, H. Shi, and X. Lu, "Hatrpc: Hint-accelerated thrift rpc over rdma," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[6] Google. "Grpc - a high performance, open source universal rpc framework." (2016), [Online]. Available: https://grpc.io (visited on 03/24/2024).

[7] Google. "Protocol buffers - protocol buffers are language-neutral, platform-neutral extensible mechanisms for serializing structured data." (), [Online]. Available: https://protobuf.dev (visited on 03/04/2024).

[8] M. Araújo, M. E. Maia, P. A. Rego, and J. N. De Souza, "Performance analysis of computational offloading on embedded platforms using the grpc framework," in *8th International Workshop on ADVANCEs in ICT Infrastructures and Services (ADVANCE 2020)*, 2020, pp. 1–8.

[9] L. Weerasinghe and I. Perera, "Evaluating the inter-service communication on microservice architecture," in *2022 7th International Conference on Information Technology Research (ICITR)*, 2022, pp. 1–6.

[10] M. Park, K. Bhardwaj, and A. Gavrilovska, "Pocket: Ml serving from the edge," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 46–62.

[11] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion, "R2P2: Making RPCs first-class datacenter citizens," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA: USENIX Association, Jul. 2019, pp. 863–880.