Improving Load Balancing of Long-lived Streaming RPCs for gRPC-enabled Inter-service Communication

Christopher Starck[1,†], Javad Ghofrani[1,*,†]

[1]*Institute of computer engineering, University of Luebeck, Luebeck, Germany*

Abstract

Many companies adopted gRPC since 2015 for their microservices architecture due to its efficient communication method, particularly regarding streaming data. However, streaming requires long-lived connections that load-balancing solutions fail to address. These problems include unexpectedly overloading services, which may result in a chain of server failures. This paper describes the load-balancing problems with long-lived streaming RPCs and proposes a flow control-inspired approach that shifts the control of incoming requests from the load balancers to the application servers. Results of our experiments show that utilizing this method leads to more server resiliency in the case of long-lived streaming RPCs.

Keywords

microservice architecture, inter-service communication, gRPC-streaming, load balancing

1. Introduction

gRPC is an open-source Remote Procedure Call (RPC) framework introduced by Google in 2015 that provides a high-performance alternative to REST [1]. Companies reported improved developer productivity with gRPC, as it allows for easier integration with existing legacy systems[1] and provides automatic code generation for various programming languages [2]. It utilizes HTTP/2 [3] for binary communication, which is more efficient than the text-based protocol used by REST. The built-in flow control and error-handling features make gRPC more reliable and a viable choice for inter-service communication in microservice architectures [4].

gRPC provides streaming RPCs between servers and clients, enabling them to keep the connection and continue with bidirectional communication once they opened it. This functionality makes the current load balancing mechanisms [5] insufficient since the load balancer will be involved only in the connection establishment step. After starting the connection, the load balancer will have no control over requests and loads on the server. This way, the clients can suddenly increase the load on the server and cause the server to fail.

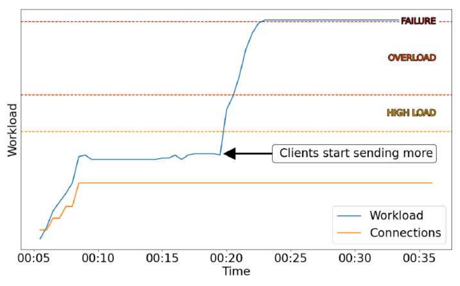
15th Central European Workshop on Services and their Composition

*Corresponding author.

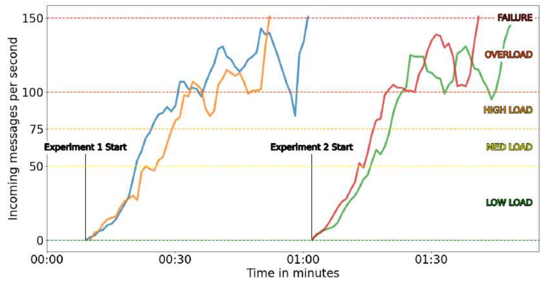
†These authors contributed equally.

christopher.starck@googlemail.com (C. Starck); javad.ghofrani@gmail.com (J. Ghofrani)
https://iti.uni-luebeck.de/ (J. Ghofrani)
0000-0001-7298-2080 (C. Starck); 0000-0002-9249-7434 (J. Ghofrani)

[1]https://www.cncf.io/case-studies/netflix/



(a) Sudden workload increase due to streaming RPCs.



(b) Servers when one of them crashes

Figure 1: Example of Sudden workload change in clients which leads to cascading failure of the servers

This paper highlights the load-balancing challenge of streaming RPCs in gRPC. We devise a flow control-inspired solution to handle this challenge. Handling such cases is important since crashing the streaming servers can lead to cascading server failure, loss of sensitive data, and decreased service quality and customer satisfaction.

This paper is structured as follows: We explain the problem of long-lived streaming RPCs with the load balancing mechanism in Section 2. In Section 3, we review the existing literature on gRPC. In Section 4, we propose our approach inspired by flow control mechanisms, and finally, we summarize our findings and recommendations for future research in Section 5.

2. Load Balancing Problems with Long-lived Streaming RPCs

There is no clear definition for when a streaming RPC can be considered long-lived. Depending on the use case, long-lived RPCs may last a few seconds in real-time scenarios or for several days in business scenarios. Streaming RPCs involve the exchange of zero or more messages over a single connection. TCP enables these types of RPCs through full-duplex mode, where messages can be sent simultaneously in both directions. Whereas client and server streaming are half-duplex, i.e. only one entity can send messages at a time. Additionally, connections are only half-closed, which allows requests to be sent over the same connections multiple subsequent times [2]. This kind of reusing of connections impairs the load balancing mechanisms which dispatch the client request to the servers.

Once a client sends a streaming RPC request to a server, the load balancing mechanism checks the server's health status and dispatches the request to the server with less working load. After accepting a streaming RPC request, the client can keep the connection open and reuse it anytime. The server overloads and crashes if many clients resume their open-held connections and start streaming. This way, the load balancing mechanism will be bypassed and cannot handle the load on the server. This can happen if a sudden increase in a server's workload causes that server to fail. All clients connected to that server will receive an error and try to send a request to another available server. The servers that end up receiving incoming requests have an increased workload. In the worst case, these servers may also fail because of the sudden increase in workload. This server could also find itself overloading due to an unexpected increase in connections. This particular problem can be categorized as a chain of server failures [6]. Figure 1 illustrates this scenario.

3. Related Work

While gRPC is a well-used technology in the industry, to the best of our knowledge, existing research in this area is limited to a few research studies on Lee and Liu [7], Chamas et al. [8], and Shah et al. [9]. Indrasiri and Kuruppu [2] also published a book on gRPC that introduces building cloud-native applications with Go and Java and discusses gRPC as inter-service communication technology.

Lee and Liu [7] present a general approach for migrating APIs from REST to gRPC. Their motivation is to improve communication performance by leveraging the features of gRPC. They developed a manual refactoring workflow with six steps and identified two directions for improvement. The first direction is automation, which could streamline the migration process. The second direction is the inclusion of error handling, authentication, and load balancing, which could improve the robustness of the migrated API. While their approach provides a helpful starting point for organizations looking to migrate to gRPC, it does not address the challenge of load-balancing long-lived streaming RPCs that we identified in our work.

Chamas et al. [8] study the energy consumption of various sorting algorithms with varying input sizes and types in the context of computation offloading mobile applications. They compare four communication protocols for remote execution: SOAP, REST, sockets, and gRPC. They reported that the local execution is generally more economical for small inputs, with a few exceptions for object input types. Their study provides valuable insights into the performance of different communication protocols in computation offloading, including the poor performance of gRPC.

Shah et al. [9] provide an overview of load-balancing algorithms in cloud computing. They noted that the load on servers increases as more applications move to or run on the cloud. Increasing the load leads to the problem of over-utilized and under-utilized servers, which necessitates load balancing. They include several static and dynamic load-balancing algorithms to address the resource allocation problem in data centers.

4. Flow control Instead of Load Balancing

Our approach is based on the idea of flow control. We utilize the sliding window method to manage a server's workload dynamically. For this purpose, the server holds a simplified internal model of the workload where the sliding window counts ongoing requests over the last t_w seconds. We use this information in our experiments to discretize the workload into LOW LOAD, MEDIUM LOAD, HIGH LOAD, OVERLOAD and SYSTEM FAILURE.

We implemented our approach using bidirectional streaming RPCs exclusively. This mode of communication allows the server to request messages from the client. We impose a constraint on the client where it must wait before sending a message after the initial message. We achieve this by introducing a special field to each message from server to client. This field holds an integer value that specifies the number of messages the client is allowed to send. Whenever a client receives a response message with a non-zero value, it can send another message or complete the call. If the field is zero, the client does not send a message and continues to wait.

These semantics allow the server to suspend and resume individual connections to prevent itself from overloading. The server suspends connections in an overloading state and resumes suspended connections otherwise.

4.1. Experiments

In our experiments, a server shuts itself down if its internal workload model is SYSTEM FAILURE. This way, we can simulate real-world conditions via small-scale experiments. We performed our experiment on Ubuntu 22.04 (64 bit) running on a system equipped with Intel(R) Xeon(R) Gold 6254 CPU (3.10GHz), 8 Cores, and 16 GB of RAM. Furthermore, we used Docker Engine Version (v20.10.21) with *buildkit* enabled and Docker Compose Version (v2.12.2).

We utilize docker swarm networking with a round-robin load balancing policy for service discovery. We intentionally use a basic load balancing setup to illustrate how our approach performs. With this setup, we can imperfectly split the connections between our servers. This imperfection is a helpful approximation for real-world scenarios. The clients were configured to send a specific amount of messages per second. Finally, clients implement a simple retry policy which repeats failed requests up to 3 times before giving up.

$$\text{Messages per Second} = \frac{\text{Number of Clients}}{\text{Number of Servers}} \times \frac{\text{Number of Messages}}{\text{Duration of Requests}} \tag{1}$$

We run several experiments with an increasing number of messages. The experiments are designed to push the workload beyond the failure threshold. First, we test that our simulation functions correctly with fewer messages. Second, we increase the number of messages to the limit of what our servers should be able to handle. This tests the ability of our approach to handle periods of increased workloads. Finally, we increase the number of messages well above the intended limits to see how our approach scales. By controlling the number of messages, we approximate a real-world scenario of thousands of clients within our resource-constrained testing hardware. We published our experiment code in a public GitHub repository [10] and included steps to reproduce it.

Parameter Name	**Configuration**
Number of Servers	2
Number of Clients	50
Number of Requests	5
Number of Messages	20,40,60,80,100
Duration of Requests	20 seconds
Failure Threshold	150 Messages per second

Table 1
Configuration for the experiments.

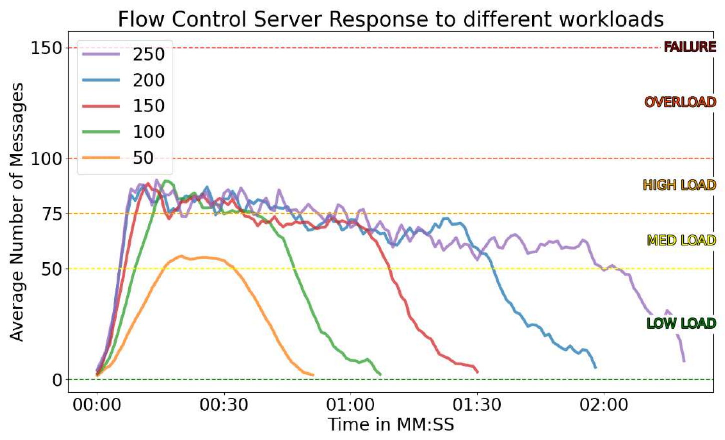


Figure 2: Flow Control server Response for various workloads. The lines represent different levels of expected workloads. Here, workload refers to incoming messages per second.

4.2. Results

Workloads above their threshold for system failure are handled robustly, far from an overloading state. The flow control mechanism can explain the server's response. When the server starts to overload, it momentarily suspends incoming requests. These requests still count towards the threshold, but subsequent requests are prevented until the workload subsides. Figure 2 shows resilient behavior to the increased workloads. The average number of messages per request decreases as the first clients complete their requests. This begins to happen when the global workload decreases. Experiments with higher initial workloads show a steeper decline in the average number of messages towards the end because each client sends a larger volume of messages.

No failed requests suggest that shifting the control from the load balancer to the application server is an effective way to handle sudden increases in incoming traffic.

We notice, however, that the requests take longer to be processed under high load, which is to be expected. In Table 2, we can see that the Round Trip Time (RTT) increases for higher workloads. The trade-off for a reliable but slower application may be worthwhile in environments where these use cases are essential.

Workload	Average RTT
50	9.52ms
100	11.98ms
150	18.03ms
200	24.77ms
250	32.88ms

Table 2
Average Round Trip Time (RTT) for different workloads. Each experiment was repeated at least 40 times.

4.3. Discussion

In Figure 2, we can see the normal response for the lowest workload in orange and the flow-controlled response for all other workloads. We find that the flow-controlled response behaves differently in preventing the sudden increase in workload. Additionally, we note that, on average, it stays below the overload threshold because the server suspends connections successfully, which further limits the workload.

The number of messages is an incomplete model of a server's workload. It is inaccurate, but it does suffice in testing our load balancing mechanism on a small scale.

We showed that an incomplete model on a resource-constrained single host machine handles workloads beyond its configured limits. Our experiment setup uses a basic configuration of the popular orchestration framework docker compose.

The ideas of our proposed solution can be generally applied to any gRPC application because it only requires a change in protobuf service definitions. Furthermore, it does not presume any specific load balancing setups and even works between a server and a client without any load balancing entity.

5. Conclusion

In this paper, we raised the issues of load balancing in long-lived streaming RPCs and explained the inability of conventional load balancing mechanisms to handle it. To solve this issue, we proposed an approach based on shifting the control over incoming requests from load balancer to the application server. This method prevents the server from overloading, a desirable feature for inter-service communication in microservice-based architecture. The management of incoming requests proves to be an effective measure in preventing server overload and cascading server failure, a chain of events where servers failing lead to others failing due to a sudden increase in re-transmitted messages from clients.

The workload model could include more parameters to better model real-world scenarios. With an extended workload model, more sophisticated flow control mechanisms should be explored. A non-linear controller could lead to further improvements where the server would try to stay close to a desired state. The performance impact on the overhead the server incurs remains to be studied.

Applying L7 Flow Control to a real-world application could provide valuable information about its effects on availability and scalability. In real-world scenarios, there are often many services and many different types of RPCs in use. Modifying the protocol buffers for each service and RPC may not be a practical solution.

Another improvement is to develop a framework that is transparent to its users. However, this requires extensive development and expertise in multiple programming languages to work seamlessly across different frameworks in gRPC. There is the possibility of introducing flow control mechanisms on a lower layer. TCP natively supports flow control [11], and gRPC also supports it at the framework level.

Although our applied flow control mechanism shows promising results, future research is required. Future research could focus on finding other ways to implement the ideas of L7 Flow Control.

References

[1] R. T. Fielding, Architectural styles and the design of network-based software architectures, University of California, Irvine, 2000.

[2] K. Indrasiri, D. Kuruppu, gRPC: Up and Running - Building Cloud Native Applications with Go and Java for Docker and Kubernetes, ”O'Reilly Media, Inc.”, Sebastopol, 2020.

[3] D. Stenberg, Http2 explained, 2014.

[4] A. Bucchiarone, N. Dragoni, S. Dustdar, P. Lago, M. Mazzara, V. Rivera, A. Sadovykh, Microservices, Science and Engineering. Springer (2020).

[5] R. Kaur, P. Luthra, Load balancing in cloud computing, in: Proceedings of international conference on recent trends in information, telecommunication and computing, ITC, Citeseer, 2012.

[6] M. T. Nygard, Release It! Design and Deploy Production-Ready Software, Pragmatic Bookshelf, Raleigh, NC, 2007.

[7] Y. Lee, Y. Liu, Using refactoring to migrate rest applications to grpc, ACM SE '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 219–223. doi:10.1145/3476883.3520220.

[8] C. L. Chamas, D. Cordeiro, M. M. Eler, Comparing rest, soap, socket and grpc in computation offloading of mobile applications: An energy cost analysis, in: 2017 IEEE 9th Latin-American Conference on Communications (LATINCOM), 2017, pp. 1–6. doi:10.1109/LATINCOM.2017.8240185.

[9] J. Shah, K. Kotecha, S. Pandya, D. Choksi, N. Joshi, Load balancing in cloud computing: Methodological survey on different types of algorithm, in: 2017 International Conference on Trends in Electronics and Informatics (ICEI), 2017, pp. 100–107. doi:10.1109/ICOEI.2017.8300865.

[10] C. Starck, J. Ghofrani, Improving Load Balancing of Long-lived Streaming RPCs for gRPC-enabled Inter-service Communication, 2023. URL: https://github.com/BalticBytes/grpc-load-balancing-long-lived-streaming-rpcs. doi:10.5281/zenodo.7641639.

[11] J. F. Kurose, K. W. Ross, Computer networking: a top-down approach, 6th ed ed., Pearson, 2013.