

# Protocol-aware Cloud Gateway with Adaptive Rate Control

Ivana Kovacevic  
Faculty of Technical Sciences  
University of Novi Sad  
Novi Sad, Serbia  
email: kovacevic.ivana@uns.ac.rs

Vasilije Milic  
Faculty of Technical Sciences  
University of Novi Sad  
Novi Sad, Serbia  
email: milic.ra208.2019@uns.ac.rs

Isidora Knezevic  
Faculty of Technical Sciences  
University of Novi Sad  
Novi Sad, Serbia  
email: knezevic.ra47.2019@uns.ac.rs

Tamara Rankovic  
Faculty of Technical Sciences  
University of Novi Sad  
Novi Sad, Serbia  
email: tamara.rankovic@uns.ac.rs

Milos Simic  
Faculty of Technical Sciences  
University of Novi Sad  
Novi Sad, Serbia  
email: milos.simic@uns.ac.rs

**Abstract**— As cloud computing has emerged as the next-generation architecture for IT enterprises, it is challenging to envision a well-configured cloud environment that delivers services without adequate mechanisms for maintaining high availability, minimizing latency, and ensuring robustness. A notable feature of distributed cloud systems is their need to support a wide range of data formats and communication protocols. The pivotal role of communication protocols in facilitating seamless interactions among distributed components depends on their capability to perform real-time data and protocol conversion, ensuring interoperability without data loss while considering latency and reliability constraints. This paper proposes a prototype of open-source components designed to enhance distributed cloud infrastructure, including a protocol-aware gateway that performs configurable protocol transcoding. Additionally, the gateway component is connected to a rate-limiting service that ensures high availability and mitigates network congestion. These components are seamlessly integrable, preserving protocol features without performance trade-offs. Their effectiveness is demonstrated through integration into the open-source Constellations (C12S) platform, validating their flexibility and practical value in real-world cloud environments.

**Keywords**—Gateway; Service discovery; Rate-limiting; Protocol transcoding; Distributed cloud.

## I. INTRODUCTION

In the present era, cloud computing offers extensive computational capabilities and facilitates on-demand access to a shared pool of both hardware and software resources. It has been introduced as the next-generation architecture of IT enterprises and gives great capabilities that ensure improved productivity with minimal costs while offering a better level of scalability and flexibility in comparison to traditional IT systems [1]. High performance, high availability, and scalability present promising features guaranteed by the migration to cloud computing. To minimize complexity and ensure a stable environment conducive to future adaptations, both business and regular users choose to leverage the hardware or software resources offered by cloud providers, aiming to enhance cost-effectiveness and simplify maintenance.

It is not easy to envision a well-configured cloud environment delivering services without incorporating mechanisms for maintaining high availability, minimizing latency, and ensuring robustness. Moreover, addressing resource exhaustion and network congestion introduces a new set of rules that require careful consideration to ensure the overall health of cloud services and protect them from

common misuse. To mitigate such risks, implementing a rate-limiting service serves as a viable solution, as a rate-limiting mechanism helps prevent resource exhaustion by temporarily blocking requests or placing them in sleep mode once a maximum limit has been reached. On the other hand, a distributed cloud aims to accommodate a wide range of data formats and protocols, facilitating seamless integration among applications. While existing cloud solutions are typically optimized for inter-service communication through RPC in a binary format, the same approach is not always suitable for external web clients. The Constellations platform is no exception. It follows the pattern of loosely coupled Dockerized micro-services, but it does not support out-of-the-box request handling beyond RPC, limiting straightforward interaction with external clients. For such scenarios, an integration of the component responsible for data and protocol conversion becomes crucial. Such a component ensures proper routing to the destination service without data loss, considering the overall network response time.

This paper centers on the design, implementation, and evaluation of two integrated, open-source, platform-independent components to maintain performance features crucial for a distributed cloud environment. Specifically, the goal is to ensure high availability and elasticity of communication between users and services, while protecting the system from excessive misuse. We propose a prototype gateway as the primary entry point to the system, which exposes Remote Procedure Calls (gRPC) as Hypertext Transfer Protocol (HTTP) endpoints by transcoding one protocol to another in a configurable manner. This service demonstrates that protocol awareness can be centralized at the entry point of a distributed cloud environment. It features dynamic client discovery and utilizes flexible configuration files for managing Application Programming Interfaces (APIs), eliminating the need to modify source code when a new service is discovered. Furthermore, the gateway is connected to a rate-limiting service to ensure availability and mitigate potential attacks. This service enforces limitations based on both system and user levels, leveraging priority queues and algorithms, such as token bucket, leaky bucket, and sliding window, to enforce fair rate control. To assess the proposed solution, both components are integrated with an open-source Constellations platform [2], which operates as a module within the distributed cloud infrastructure.

The paper is organized as follows: Section 2 presents the related work for this research on performance in

distributed cloud, with a particular focus on gateways that ensure low latency and high availability. Section 3 provides an overview of rate-limiting algorithms, their advantages, and applications. In Section 4, the gateway is described. Section 5 explains the implementation of a protocol conversion service, a rate-limiting service, and their integration with the open-source platform for configuration dissemination in a distributed cloud. The usability, interoperability, and limitations of the proposed solution are discussed in Section 6. Finally, Section 7 presents the conclusion and future directions of the conducted research.

## II. RELATED WORK

In their study, El Kafhali et al. [1] presented a thorough overview of cloud computing mechanisms, offering a systematic literature review specifically focused on cloud computing security issues and frameworks through a comprehensive survey. Their paper provided an overview of the fundamentals of cloud infrastructure, reflecting on the mechanisms to achieve scalability and availability, while considering proper defense against attacks. Latha et al. [3] conducted research that addresses challenges in distributed applications, focusing on client satisfaction, confidence, and preventing revenue losses by ensuring service availability. Their study developed an overload protection technique that relies on a URI configuration file, in conjunction with the Zuul gateway, which can filter requests before obtaining tokens. The token bucket rate-limiting algorithm is implemented to ensure traffic limitation while improving the reliability and availability of the cloud platform service. Despite integrating the gateway with rate-limiting to enhance availability, this research remains protocol-dependent and lacks protocol transcoding, which would enable flexibility and broaden its usage. Distributed cloud control approaches are also demonstrated in papers by Raghavan et al. [15] and in “Load balancing vs. distributed rate limiting: a unifying framework for cloud control” written by Stanojevic Rade et al. [16]. However, they do not describe a holistic approach with an integrated API gateway for monitoring and filtering requests that could also be protocol-agnostic.

Ranawaka et al. [14] emphasized the need to provide a scalable microservice architecture that offers highly available and fault-tolerant operations. They implemented Custos, which exposes services through a language-independent Application Programming Interface that encapsulates science gateway usage scenarios. This work primarily focuses on science-specific gateways in a research domain, tailored for computational experiments while hiding the complexities of accessing and using cyberinfrastructure. Although the necessity for such a solution is evident, the paper lacks an explanation on how to ensure scalability as the number of requests increases while protecting the platform from malicious Denial-of-Service (DoS) attacks.

## III. RATE-LIMITING IN THE CLOUD ENVIRONMENT

To ensure service availability and achieve high scalability, cloud services must protect themselves against excessive usage, whether it is expected or not. Cloud services should be developed with rate limitations in mind to ensure the system operates properly and avoids cascading failure. For increasing throughput and decreasing end-to-end delay over large distribution systems, rate limiting on either the client or server side is critical [3]. Our approach in this research is to implement a prototype rate limiting at the OSI layer 7, to prevent resource exhaustion and maintain system resilience. We propose rate control at the entry point level, paired with the gateway. By integrating rate limiting within gateways, API usage can be centrally controlled across all deployed nodes, ensuring uniform policy enforcement and simplifying management.

Rate limiting helps prevent resource exhaustion by temporarily blocking requests or placing them in sleep mode once a maximum limit has been reached. After the sleep time, the request can be forwarded from the rate limiter to the handling server [4]. Rate limiting has found use in various cases, including improving overall system performance, protecting against brute force or Distributed Denial-of-Service (DDoS) attacks, preventing web scraping, and preventing resource starvation. Scalable rate limiting is achieved using various algorithmic approaches, including the leaky bucket algorithm, the token bucket algorithm, the fixed window, the sliding log, and the sliding window [3]. This paper focuses on the leaky bucket algorithm, the token bucket algorithm, and the sliding window, all of which are implemented within our rate-limiting service. The token bucket algorithm provides solutions for traffic shaping in packet-switched networks [5]. In this algorithm, when a new request arrives, the bucket grants one token to the requester, based on the availability [6]. If there are available tokens, the service accepts the request and removes one token from the bucket. If no tokens are available, the system rejects the request. This algorithm also requires a parameter for the refill rate, as it adds tokens to the bucket at a fixed rate defined by this parameter. It is a common choice in distributed systems, primarily due to its memory efficiency and ease of implementation.

The sliding window algorithm imposes limits within fixed time intervals, allowing for precise control over requests in smaller time windows. It admits a specified number of requests in a given timeframe  $L$ . As each request arrives, a request counter is incremented by one. This process continues as long as the request counter is less than a specified fixed number. At the end of a window interval, the request counter resets. Intervals are half open, i.e.,  $[t, t+L)$  [7]. The leaky bucket is a counter that increases by one up to a maximum capacity  $C$  for each arrival and decreases continuously at a given drain rate  $D$  to as low as zero; an arrival is admitted if the counter is less than or equal to  $C - 1$  (so that after the arrival it will be less than or equal to  $C$ ) [7]. The leaky bucket algorithm is designed to provide clients with smooth and steady

throughput by delaying requests rather than rejecting them outright. While this approach may increase latency due to its lack of drop behavior, it remains well-suited for use cases like background processing or metrics collection. That said, we also support two additional rate-limiting algorithms, giving clients the flexibility to choose the strategy that best fits their specific needs.

The proposed solution emphasizes implementing API rate-limiting as a centralized, independent component, which differs from traditional methods that integrate rate-limiting algorithms directly into individual services. By applying rate limiting on a system-wide basis, we gain finer control, allowing for multiple configurations for each request or service. Additionally, this approach can be developed and deployed separately, offering greater flexibility and ease of management. Having a single, global limit also avoids common problems related to communication and synchronization among multiple, distributed rate-limiting services [7].

#### IV. TRANSCODING HTTP TO GRPC

While HTTP is a very popular choice due to its simplicity and stateless nature, some studies have shown that RPC outperforms HTTP in terms of response time and data volume [8], [9]. Moreover, 80% of the public APIs available follow most Representational State Transfer (REST) conventions, and developers are accustomed to that pattern, implying the need for gRPC APIs also to follow REST convention [10]. Additionally, having multiple cloud providers joined in a distributed cloud, cross-platform compatibility issues, and inconsistent call standards arise. Placing separate components as an API gateway alleviates these problems to some extent. To enhance user experience and minimize development costs, we propose a configurable rate-limiting gateway that is designed to fully comply with the REST while retaining the advantages of remote procedure calls. With this, existing REST endpoints can be efficiently transcoded to use the RPC protocol, guaranteeing no data loss. Remote procedure calls heavily rely on Protobuf, an open-source technique for serializing structured data [10]. Unlike JavaScript Object Notation (JSON), Protobuf is optimized and runs in binary format, which is why it is often the preferred choice. Additionally, Protobuf offers a mechanism to segregate context and data, allowing data to be transmitted repeatedly without duplicating context, such as field or property names, as often occurs in JSON or eXtensible Markup Language (XML). In practice, both gRPC APIs and HTTP/JSON APIs serve distinct purposes, and an ideal API platform should offer robust support for both types.

For protocol transcoding, the proposed gateway component leverages gRPC client reflection to dynamically discover methods, ensuring interoperability across services and reducing the need for manual adjustments. Given a hostname and port provided in the configuration scheme, the gateway attempts to establish a connection to the specific gRPC server and dynamically discover available services and methods without prior knowledge. Discovered services are later used in the

process of protocol transcoding in order to forward data from the original HTTP request to the corresponding RPC service method. Moreover, by providing a transcoding feature, it is possible not only to determine what formats (i.e., which Protobuf messages) a server's method uses but also how to convert messages between a human-readable format, which is dominant in HTTP, and the binary wire format.

#### V. IMPLEMENTATION AND THE USE CASE

Configurable, highly available cloud services, namely a gateway and rate-limiting service, are integrated within the configuration dissemination tool in the distributed cloud. This tool is part of the Constellations, an open-source, distributed cloud platform [2]. The main objective of the tool is to enable cloud-like services for users who would benefit from highly elastic deployments, while also taking latency and privacy requirements into account. To achieve so, the tool offers streamlined processes for infrastructure provisioning, application life cycle, and behavior management [10]. As the platform has multiple services distributed across the cloud that communicate using gRPC protocol, adding a rate-limiting gateway only increased its heterogeneity and improved the response rate.

##### A. Gateway

The gateway solution offers a flexible approach for exposing gRPC calls as REST endpoints. Instead of burdening each service with boilerplate code to enable transcoding, this approach delegates the protocol conversion logic to a dedicated service acting as a proxy between the platform's end clients and the internal services. It uses the flexibility of the configuration file to avoid source code alterations that would otherwise be mandatory and are common in other prominent gateway implementations [12].

Within the configuration file, the highest level of API description is an API group, which encompasses versioned descriptions for the gRPC methods intended for exposure. These methods are grouped based on their purpose, allowing for the inclusion of gRPC methods from various services and applications. Bundling the APIs into API groups simplifies access to the methods needed for specific purposes, eliminating the need to search through an extensive list of APIs from each service to locate a particular method. A description of each gRPC method includes the REST route, HTTP method type (e.g., GET, POST), and the gRPC service that hosts it. The method's name serves as a key in a map during the dynamic generation of routes, and it must match the name in the source service. The configuration also includes the port of the gateway and the addresses of the gRPC services used. These addresses are kept internal to the gateway and are inaccessible from outside sources, meaning they cannot be directly reached via either gRPC or HTTP requests. The example of the configuration file is shown in Figure 1. The service registry, as a separate component within the gateway, allows services to register their endpoints, which are then stored in a configuration file. In the event of a failure, the gateway utilizes this configuration file to route

```

gateway:
  route: /apis
  port: 5555
services:
  Kuiper: kuiper:5000
  ExampleService: example:9001
  RateLimitService: rate_limiter_service:8080
groups:
  core:
    v1:
      CreateExample:
        method_route: /example-route
        type: POST
        service: ExampleService
      PutStandaloneConfig:
        method_route: /configs/standalone
        type: PUT
        service: Kuiper

```

Figure 1. Example of a YAML gateway configuration.

requests, eliminating the need to ping each service individually to collect routes.

Upon initializing the gateway, the configuration file is loaded, and for each gRPC service listed, a corresponding Client object is instantiated. Each client object includes an attribute called *DescriptorSource*, which is derived from the gRPC reflection mechanism and participates in obtaining a list of exposed gRPC methods from each client. However, this solution relies on gRPC services having reflection enabled, which allows clients to access detailed information about the Protobuf APIs they expose, including the specifications of each request and response, as well as their attributes. To invoke gRPC calls, this service depended on the Go library *grpcurl* [13]. This decision was beneficial because *grpcurl* efficiently converts HTTP data to gRPC data, simplifying the procedure. Moreover, *grpcurl* supports request headers during invocation, which was crucial for later authorization between services.

The process of HTTP route generation consists of several parts:

1. Generation of sub-routers for every group,
2. Sub-routing groups based on the version,
3. Assigning a path to each route based on the method name from the configuration file,
4. Creating a middleware that integrates a handler function and HTTP method type for each route.

The second step provides fine-grained configuration of routes, combining group and version, resulting in each method being mapped with its version and group, allowing for easier maintenance of clients in the future, based on the current API version. All of these parameters are required to create a complete path for each method. The full path is created in the third step, using the exact method name previously read from the configuration file. The final step is to prepare the router for gRPC method invocation. To achieve this, HTTP endpoints are wrapped into the middleware, which performs preprocessing and validation before the actual gRPC call. The transcoding process is

validated against GET, POST, PUT and DELETE HTTP methods, with and without custom HTTP headers. To extract parameters from routes, the gateway uses regular expressions and then performs implicit type conversion. To prevent unauthorized access, the middleware includes a check for authorization tokens, verifying each request before directing it to the destination service. This process helps eliminate redundant calls to services with restricted access, enhancing security and improving response time. Once the gRPC method is invoked and completed successfully, the response is returned as a byte buffer. Additionally, the gRPC status codes are mapped to their corresponding HTTP response codes. This mapping is particularly helpful in case of errors, as it enables the provision of informative messages that explain why the error occurred.

### B. Rate limiter

The rate-limiting service provides customizable rate-limiting mechanisms per request at both the application and system layers. Limitations are designed per client. In our scenario, clients refer to end users of the constellation platform. However, distinct rate limiters can be created based on the requirements of cloud services, irrespective of client types. Full support for managing rate limiters is also provided, enabling rate or type updates, safe deletion, and optional parameters. To support flexible request control, the prototype offers multiple rate-limiting algorithms that clients can choose from based on their specific needs. To apply safety measures against API overuse, we first define a rate-limiting strategy. Every rate limiter is assigned a unique ID in the format of *user\_id-method\_id*. For seamless integration, *method\_id* corresponds to a method name retrieved from gRPC clients in the gateway. As this data is already extracted and prepared for routing to the desired service, no further querying or communication with other services is necessary. Given that this ID is treated as a regular expression, any notation is allowed, making it usable not only for gRPC methods but also for users or organizations that require limited access to resources. For instance, it is possible to configure access limitations for authenticated users and request origins, forbidding usage from multiple devices simultaneously. Apart from ID, the rate limiter is also described with TYPE, REQ\_LIMIT, PRIORITY, PERIOD, and BURST. Attribute TYPE is directly related to supported rate-limiting algorithms, currently limited to

```

&pb.RateLimiter {
  id: "user1-PutStandaloneConfig",
  Name: "PutStandaloneConfig",
  UserName: "user1",
  Type: "tokenBucket",
  Priority: 1,
  ReqLimit: 1,
  Period: 60,
  Burst: 1,
  Idle: 2
}

```

Figure 2. Code snippet demonstrating a rate limiter object

token bucket, leaky bucket, and sliding window. The total number of allowed requests within a specified period is determined by the combination of the attributes REQ\_LIMIT and PERIOD. A period of time can be expressed in seconds. This additional parameter enables services to create rate limiters tailored to internal service metrics. These metrics might incorporate temporal factors, such as the number of served requests during specific periods of the day. Parameter BURST stands for the maximum number of concurrent requests that the API can handle, and it is used to regulate throttling in the token bucket algorithm. With larger bursts, the network may need to allocate more resources per connection [7].

The rate limiter also supports a priority queueing mechanism that can be utilized to favor users who are most frequently rejected due to system limitations. Priority can be handled at either the method or user level, where critical, time-sensitive methods have higher priority, while tracking or monitoring methods can be accessed with a delay. A lower PRIORITY value means a higher priority in the queue; thus, a value of 1 indicates the highest priority. If the userName is not provided, PRIORITY refers to the method. It is also possible to define an IDLE parameter for slow connections. This parameter and priority queueing are optional and can be deactivated based on system needs. The example of a rate limiter is shown in Figure 2. This rate limiter is set to allow only one request per minute, using the token bucket algorithm. It is defined for the user with the highest priority. Only one rate limiter can be active per client-request combination. This aligns with the notion of a fixed number of requests per user, as offered by cloud provider subscription plans. Any changes made to the rate limiter will override previous settings and reset the number of available tokens.

To minimize response time, a caching mechanism is implemented in the rate-limiting service. This mechanism stores the current state of the rate limiter in a cache memory for a specified period, reducing the number of calls to the database. The cache is updated each time a service modifies the rate limiter object. However, inconsistencies can arise due to network delays and concurrent updates, which may allow clients to exceed rate limits before the state is synchronized. Achieving strong state consistency can introduce significant overhead, resulting in longer processing times and reduced performance. To address this issue, we decided to integrate with Redis due to its ability to perform operations in memory, which reduces latency and makes it suitable for high-traffic environments where rate limits need frequent checking and updating.

### C. Integration with the Constellations platform

As an evaluation, developed components are integrated with an open-source platform within a distributed cloud infrastructure to facilitate two-way protocol conversion and manage resource availability by enforcing rate control. To illustrate the flow of the transcoding process, Figure 3 depicts a scenario in which two clients send identical requests within a predefined timeframe. Requests are sent to the service responsible for configuration management within the Constellations platform. The service responsible for this feature is represented as a constellation service in a diagram. For example, in Figure 3, we demonstrate two calls to an endpoint that has a system rate limit of one request per minute. After receiving a request, the gateway uses *DescriptorSource* to determine the actual method name bound to the received HTTP request. Based on the configuration file, it maps parameters (if any) and

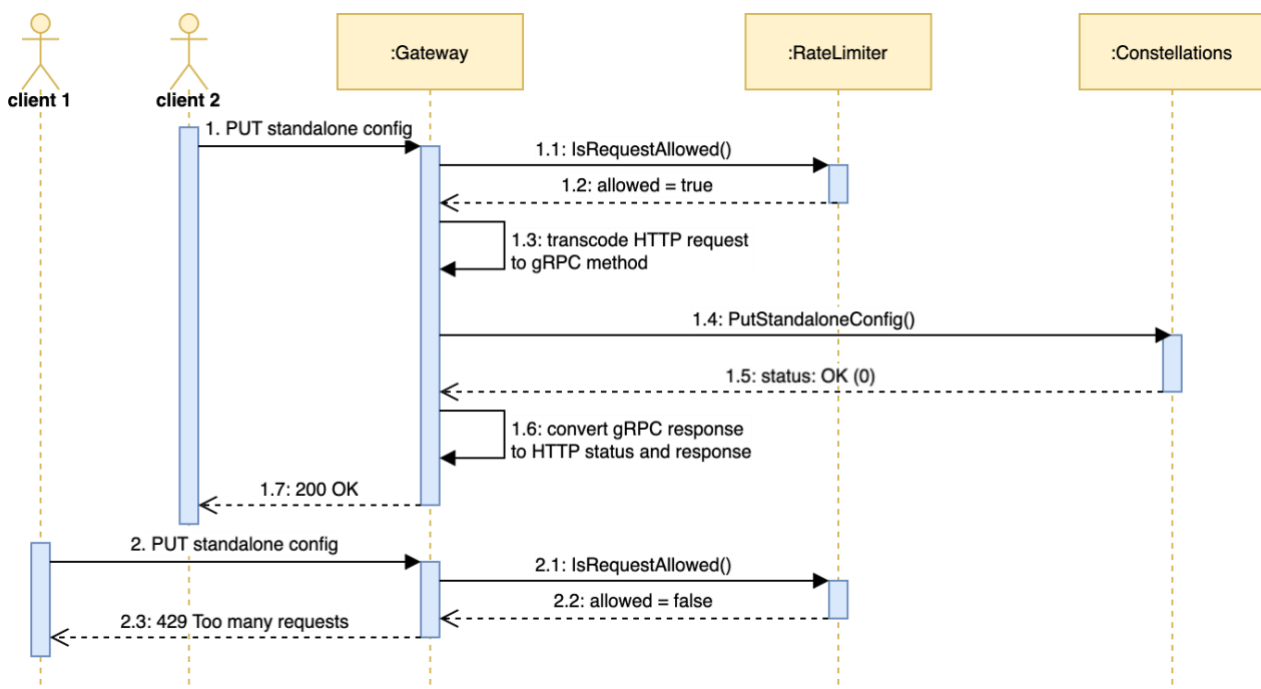


Figure 3. Sequence flow demonstrating the integration of gateway and rate-limiting services with the Constellations platform

converts the request payload to a byte stream suitable for the Protobuf format. The method name from the configuration is then used in a direct gRPC call to the rate-limiting service. The *IsRequestAllowed* method in the rate-limiting service searches for a rate limiter object based on its ID and then examines the rate limiter type to determine whether a request can be executed at the moment.

If an optional parameter is provided, the rate limiter service is also responsible for checking the user priority. Based on the examination of parameters, if the limit is reached, the method returns a false flag. Given the flag value, the gateway decides whether to invoke the actual gRPC method and transfer the request. As shown in Figure 3, for the second client, the rate limit is reached, and the request is blocked immediately. The same sequence is followed in case where a thousand users concurrently send identical requests, and the control rate is shown in Table 1, for each rate-limiting algorithm.

In Table 1, we compared the average latency introduced by different rate-limiting algorithms implemented in our service. We sent 1000 requests to the same route, configured to use the token bucket, leaky bucket, and sliding window algorithm, with the same *reqLimit* parameter set to 10. This seemed reasonable, considering the configuration is per IP address, and the average response time without a rate limit for the route was approximately 100-200ms. Both client IP address and Constellations' server were connected to the same internal network. Control rate is measured for the system rate limiter, representing the ratio between the number of successful and the number of rejected requests (those with 429 status code). Average latency represents the ratio between regular response time and response time when the rate limiter is applied.

TABLE I. A COMPARISON OF RATE-LIMITING ALGORITHMS

	Token bucket	Leaky Bucket	Sliding Window
avg. response time	0.097s	1.001s	0.095s
avg. latency	0.074s	0.043s	0.022s
control rate	0.1273	/	0.1235
total time (~1000 req)	10.502s	13.253s	10.084s

This approach enhances performance by minimizing unnecessary calls to the destination service while also providing the possibility to enforce a global rate limit that a user can achieve, regardless of the cloud service being accessed. As shown in Table 1, the control rate for the leaky bucket is not calculated, since all requests pass with a slight delay. Therefore, it is a client's responsibility to define the rate limit in advance, choosing the most appropriate algorithm depending on the use case. The transcoding process occurs at the beginning of the request call. It takes less than 5 ms, which turned out to be negligible performance-wise, especially considering that mapping is performed in the beginning, and no additional handling of routes is needed.

## VI. DISCUSSION

In this research, we propose a solution to address issues in multiprotocol environments, emphasizing the need for cloud services to communicate in a predefined manner. Most cloud platforms support RPC internally and require additional time and resources to expose RPC methods as REST endpoints. Instead of the time-consuming process of refactoring existing services, we propose integration with a component that already offers protocol conversion and enables straightforward migration with API versioning. Therefore, we developed a protocol-aware gateway responsible for transcoding HTTP to RPC, following reconfigurable mapping of routes. This approach proved helpful in different settings as it supports both client reflection and the set of configuration rules described in YAML files, enabling proper connection between service methods and REST endpoints. Having this configuration separated from the internal logic of the connected services in the cloud reduces development time while making management easier. Its scheme is tested against routes with query parameters, path parameters, authorization, and custom headers, as well as with a request body, and it performs transcoding without data or header information loss. It can differentiate between unauthorized and authorized methods, preventing misuse, and could leverage access control measures if they are implemented further in the cloud environment. Moreover, the solution only requires following the schema pattern and can be easily integrated into existing cloud infrastructures, which we have demonstrated by incorporating it with the Constellations platform. However, since the proposed transcoding process heavily relies on the configuration file to extract routes, it is important to note that a strong automated YAML scheme validation is needed in order to minimize ambiguity and reduce the risk of errors.

Furthermore, this prototype relies on I/O operations to read the configuration and to track changes as new routes are added. This did not come as a bottleneck for the current setup, but it should be monitored as the number of services grows in the cloud. One possible approach would be to partition the configuration by services or their deployment location and scale horizontally. We integrated a protocol-aware gateway with the rate limiter and demonstrated its strong properties in precision rate control and manageability. With its priority queueing and system-agnostic features, it effectively enhances system safety and ensures alignment with platform requirements. Such granularity can be a trade-off between rate-limiting accuracy and performance; therefore, it is up to end users to decide whether to include fine-tuning of requests or not.


## VII. CONCLUSION

The paper presents mechanisms for achieving desired performance features in a distributed cloud environment, with a focus on high availability, scalability, and robustness. To achieve these goals, we demonstrated the integration of two prototype components, namely the gateway and rate-limiting service, with an existing



configuration dissemination solution. The prototype emphasizes its ease of adoption, platform-agnostic design, and the ability to enforce consistent communication patterns without sacrificing flexibility or performance. Key contributions include enabling protocol transcoding from HTTP to gRPC calls with reflection for method discovery, facilitating the transcoding of HTTP headers and body to Protobuf messages, and, in the opposite direction, packaging byte streams into readable HTTP responses in JSON format, while also ensuring proper status code mapping. This addressed the necessity for each service to expose both HTTP and gRPC endpoints. Additionally, it enhanced the availability of each service by maintaining communication within the platform on gRPC, thus boosting efficiency. Furthermore, the gateway, paired with an independent rate-limiting service, eliminates the need for each service to alter its internal logic or modify request implementation to manage and regulate network congestion. System rate limiting manages and controls overall network flow within the platform, while also allowing for the creation of specific limitations on a per-request or priority basis, which emerges as a suitable solution for subscription plans structured around request rates from cloud providers. As part of our future work, we aim to enhance rate-limiting capabilities by making them adjustable based on service telemetry and monitoring, and to extend the current solution to support distributed rate-limiting. Additionally, the goal is to further research prototype performance and general applicability by integrating it with more real-world solutions.

#### ACKNOWLEDGMENT

 Funded by the European Union (TaRDIS, 101093006). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

This research has been supported by the Ministry of Science, Technological Development and Innovation (Contract No. 451-03-65/2024-03/200156) and the Faculty of Technical Sciences, University of Novi Sad through project “Scientific and Artistic Research Work of Researchers in Teaching and Associate Positions at the Faculty of Technical Sciences, University of Novi Sad” (No. 01-3394/1).

#### REFERENCES

- [1] S. El Kafhali, I. El Mir, and M. Hanini, “Security Threats, Defense Mechanisms, Challenges, and Future Directions in Cloud Computing,” *Archives of Computational Methods in Engineering*, vol. 29, no. 1, Apr. 2021, doi: <https://doi.org/10.1007/s11831-021-09573-y>.
- [2] “constellations” *GitHub*. [Online] Available from: <https://github.com/c12s> [retrieved: 06, 2025]
- [3] V. L. Padma Latha, N. Sudhakar Reddy, and A. Suresh Babu, “Optimizing Scalability and Availability of Cloud Based Software Services Using Modified Scale Rate Limiting Algorithm,” *Theoretical Computer Science*, Jul. 2022, doi: <https://doi.org/10.1016/j.tcs.2022.07.019>.
- [4] D. Goetz, M. Barton, and G. Lange, “Distributed rate limiting of handling requests,” United States Patent 8930489, Jan. 6, 2015.
- [5] L. Sarakis, N. Moshopoulos, D. Loukatos, K. Marinis, P. Stathopoulos, and N. Mitrou, “A versatile timing unit for traffic shaping, policing and charging in packet-switched networks,” *Journal of Systems Architecture*, vol. 54, no. 5, pp. 491–506, Sep. 2007, doi: <https://doi.org/10.1016/j.sysarc.2007.08.004>.
- [6] J. Rexford, F. Bonomi, A. Greenberg, and A. Wong, “Scalable architectures for integrated traffic shaping and link scheduling in high-speed ATM switches,” *IEEE Journal on Selected Areas in Communications*, vol. 15, no. 5, pp. 938–950, Jun. 1997, doi: <https://doi.org/10.1109/49.594854>.
- [7] A. W. Berger and W. Whitt, “A comparison of the sliding window and the leaky bucket,” *Queueing Systems*, vol. 20, no. 1–2, pp. 117–138, Mar. 1995, doi: <https://doi.org/10.1007/bf01158434>.
- [8] M. Niswar, R. A. Safruddin, A. Bustamin, and I. Aswad, “Performance Evaluation of Microservices Communication with REST, GraphQL, and gRPC,” *International Journal of Electronics and Telecommunication*, vol. 70, no. 2, pp. 429–436, 2024, [Online] Available from: <https://ijet.ise.pw.edu.pl/index.php/ijet/article/view/10.2442-5-ijet.2024.149562>
- [9] M. Śliwa and B. Pańczyk, “Performance comparison of programming interfaces on the example of REST API, GraphQL and gRPC,” *Journal of Computer Sciences Institute*, vol. 21, pp. 356–361, Dec. 2021, doi: <https://doi.org/10.35784/jcsi.2744>.
- [10] “Protocol Buffers,” *protobuf.dev*. [Online] Available from: <https://protobuf.dev> [retrieved: 06, 2025].
- [11] T. Ranković, I. Kovačević, V. Maksimović, G. Sladić, and M. Simić, “Configuration Management in the Distributed Cloud,” *Lecture notes in networks and systems*, pp. 224–235, Jan. 2024, doi: [https://doi.org/10.1007/978-3-031-71419-1\\_20](https://doi.org/10.1007/978-3-031-71419-1_20).
- [12] “Gateway architecture | NGINX Documentation,” *Nginx.com*, 2025. [Online] Available from: <https://docs.nginx.com/nginx-gateway-fabric/overview/gateway-architecture/> [retrieved: 06, 2025].
- [13] “grpcurl package - Go Packages,” *Go.dev*, 2025. [Online] Available from: <https://pkg.go.dev/github.com/fullstorydev/grpcurl> [retrieved: 06, 2025].
- [14] I. Ranawaka *et al.*, “Custos: Security Middleware for Science Gateways,” *Practice and Experience in Advanced Research Computing*, pp. 278–284, Jul. 2020, doi: <https://doi.org/10.1145/3311790.3396635>.
- [15] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren, “Cloud control with distributed rate limiting,” *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4, pp. 337–348, Oct. 2007, doi: <https://doi.org/10.1145/1282427.1282419>.
- [16] R. Stanojevic and R. Shorten, “Load Balancing vs. Distributed Rate Limiting: An Unifying Framework for Cloud Control,” *IEEE Xplore*, Jun. 01, 2009. <https://ieeexplore.ieee.org/abstract/document/5199141>.