ISSN 2224-087X. Електроніка та інформаційні технології. 2024. Випуск 27. С. 84–101
Electronics and information technologies. 2024. Issue 27. P. 84–101

UDC 004.4`2:004.054 DOI: https://doi.org/10.30970/eli.27.7

DEVELOPMENT OF A SOFTWARE CONTROLLER FOR THE AUTOMATED CREATION AND MANAGEMENT OF VIRTUAL RESOURCES FOR RUNNERS

Yu. Korchak, B. Mikh, Yu. Furgala

Ivan Franko National University of Lviv,
107 Tarnavsky St., UA–79017 Lviv, Ukraine
yuriy.korchak@lnu.edu.ua

The work developed a functional model (prototype) of the controller for the automated creation and management of runners' virtual computing resources. The analysis of potential technologies for its implementation determined the choice of a combination of RPC, cRPC, HTTP/2 and Go technologies with the protobuff serializer, which meets all modern requirements for a distributed system's speed, efficiency and scalability. It has been experimentally proven that the RPC protocol can serve as a reliable interface for managing resources in virtual environments, providing convenience and efficient integration with CI/CD systems. Using test scenarios made it possible to reflect actual operating conditions and integration testing and, as a result, evaluate the interaction between the MVP and other system components.

Keywords: Continuous Integration, Continuous Delivery, Remote Procedure Call, gRPC, cRPC, HTTP/2, Golang, GitHub Actions, runner, controller, MVP.

Introduction

In today's world, where technologies are developing at an incredible speed, and the requirements for software efficiency are constantly growing, the automation of the development and delivery processes of software products is becoming critically important [1, 2]. As an example, the Continuous Integration (CI) and Continuous Delivery (CD) paradigms provide a framework for rapid and reliable integration and delivery of software products, which is critical for software development companies [3].

Flexible management of virtual resources becomes an integral part of this process, as it allows to quickly perform such tasks as training models, deploying programs, and automating testing [4]. The development of a software controller for automating the management of virtual resources is an important task, especially one that can use any virtualization technology, provided that a communication interface with the controller is implemented.

This paper presents a software solution based on the use of the Connect Remote Procedure Call (cRPC) framework, which allows for the integration of virtualization and containerization platforms. cRPC provides a high level of abstraction and independence from programming languages and virtualization technologies, which allows you to effectively adapt to various development environments and code execution on different platforms.

The choice of technologies and their justification

RPC, ConnectRPC, Golang, and Protocol Buffers technologies were chosen for the project's implementation. CI and CD were used as software development methodologies, and GitHub Actions (GHA) was used as a development environment. Let's consider in more detail the practicality of choosing such software technologies.

First of all, it should be noted that CI and CD allow you to automate and optimize the processes of development, testing and deployment of software products, or to automate routine or complex processes, such as training artificial intelligence models, building and delivering a software product to end users, automating testing, etc.

Fig. 1. General diagram of CI/CD paradigms.

These paradigms (see Fig. 1) provide a continuous cycle of integrating code changes and delivering them to end users, which significantly increases the speed and quality of software product development.

CI is a practice that involves regularly placing all changes in work activities (for example, raw code) in the main development branch or project [5]. The main components of the CI paradigm are: automated code assembly, automated testing, code analysis, and security.

CD is a software development practice that works closely with CI to automate deployment (Fig. 1) [6]. After the code has been tested and compiled as part of the CI process, the CD ensures that it is packaged with all the necessary components for deployment to any environment. CD covers the entire process from provisioning the infrastructure to deploying the application to a test or production environment. The main components of the CD paradigm: automatic deployment, monitoring and feedback, configuration management, readiness for deployment.

All these steps are performed on the smallest atomic units of CI/CD – runners that receive scripts to execute specific commands through instructions (pipelines). A pipeline is a set of automated steps executed sequentially and described by particular commands in the context of a YAML file. A pipeline consists of several stages, building code, running tests, analyzing code quality, and deploying to test or production environments [7]. Each step is performed on the runner. These steps are grouped into jobs. A Job is a single step or set of commands executed in a CI/CD process. A Job can include tasks such as building code, running tests, analyzing code, deploying and running software in its environment, etc. Each job is executed in an isolated environment.

Runner is a computing node or agent that performs tasks described in advance using specialized commands. It helps automate the building, testing and deploying software [8]. Runners function as CI/CD work units, running tasks defined in the project configuration. They are divided into common (for several projects simultaneously, do not require a specific environment) and unique (for a particular project, specific setting). Runners automatically perform tasks defined in scripts or project configuration files. One of the key features of runners is their ephemerality, which means creating a new, clean environment each time complete tasks.

This is important to ensure consistency and predictability of test and build results. For example, having the same environment during testing is necessary. In that case, the runner creates an isolated environment that disappears after the task is completed, thus guaranteeing the stability and repeatability of the processes. Advantages of using runners include process automation, scalability and flexibility, parallelism and isolation, automatic recovery, tracing and logging. Thanks to these features, runners act as an indispensable component in the automation of CI/CD processes, allowing developers to increase productivity, reduce development time and ensure high-quality end products.

As the development environment, GitHub Actions was chosen, an automation platform integrated directly into GitHub that allows you to automate all aspects of software development processes, including tests, builds, deployment, integration and delivery (CI/CD). The use of GHA significantly simplifies the development and maintenance of projects due to the automation of routine tasks and the implementation of continuous testing and deployment processes [9].

The Remote Procedure Call (RPC) protocol is used for client-server interaction - a fundamental technology for developing distributed, client-server applications. Historically, RPC was developed in the 1980s as part of the Apollo Network Computing System project [10] and further standardized in the 1990s as part of the Open Network Computing Remote Procedure Call. This system used Remote Procedure Call to provide interoperability between distributed systems, allowing developers to communicate between different computers on a network as if using local server procedures. Thanks to its efficiency and flexibility, RPC gradually became almost the leading solution for the implementation of distributed applications that require reliable mechanisms of interaction with each other under heavy service loads. Google subsequently extended the concept of RPC with the introduction of gRPC in 2015 [11], extending the concept of RPC as such. In addition, from that moment, gRPC began to successfully compete with the only REST API technology available at that time, which was already morally outdated at that time and had many shortcomings precisely in the context of large distributed loads.

gRPC is an open framework [12] that allows you to quickly and efficiently create interoperable services using HTTP/2 for transport, while REST uses an older version of the HTTP/1.1 data transfer protocol. gRPC enables the use of bi-directional streaming, i.e. a communication or communication channel that is constantly supported by both parties, and provides efficient and lossless data validation, serialization and deserialization using the Protocol Buffers tool.

Fig. 2 shows the client-server interaction diagram in the RPC protocol. RPC has the following sequence of procedures (see Fig. 2):

1. initialization of the call on the client machine: the client initiates the procedure call, which is sent to the client module, which serializes the call into a package;

2. sending a packet: the serialized call is forwarded through the network infrastructure using the RPC runtime, which handles the data transfer between the client and the server;

3. reception on the server machine: the server module receives the packet, unpacks it and interprets the procedure call;

4. execution of the procedure: the server performs the required procedure based on the information received from the client;

5. returning the result: the result of the procedure is packaged by the server, transmitted back to the client through the server module;

6. processing the result on the client machine: the client module unpacks the received result and passes it to the client program.

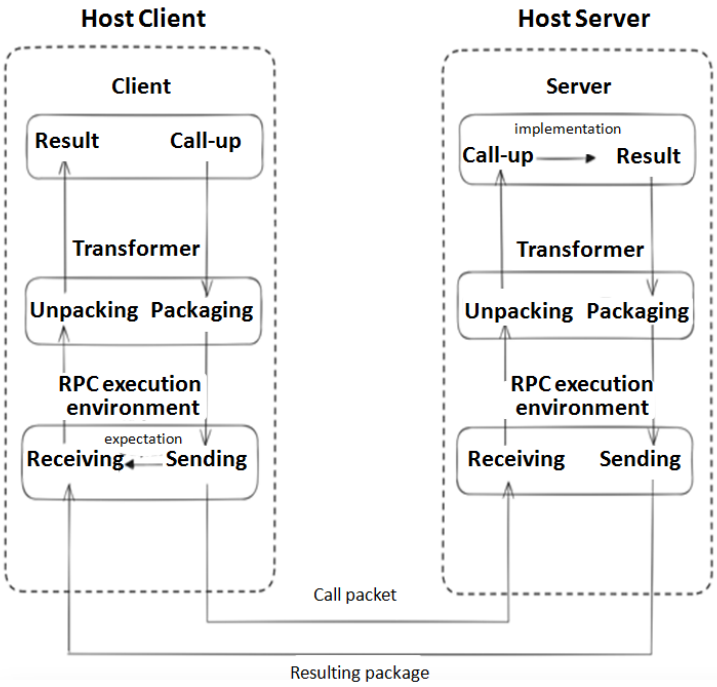

Fig. 2. Diagram of client-server interaction using the RPC protocol.

This process illustrates how RPC enables remote execution of functions between computers on a network, minimizing the complexity of communications for the user. It is important to note that all messages occur either in binary or hexadecimal code, which allows both to increase the speed of the application itself and to reduce the load on the server.

Today, another modification, or rather an addition, of this ConnectRPC (cRPC) protocol has already been created, which is a hybrid protocol designed to optimize the interaction between components of distributed systems that use modern and traditional web technologies [13]. This protocol integrates the capabilities of HTTP/1.1 and HTTP/2, providing efficient solutions for managing multiplexed requests and responses, and supporting streaming data. The main properties of cRPC: support for HTTP/2, compatibility with HTTP/1.1, ease of use (Fig. 3).

Fig. 3 shows a request processing scheme in the context of the cRPC protocol [13], which depicts the duality of supporting REST API and RPC calls.

The universal tool Protocol Buffers (protobuf), created by Google [14], is used for data serialization. This tool allows you to convert structured data into a compact binary format that is ideal for fast data exchange between different applications and systems. In contrast to XML or JSON formats, protobuf provides greater efficiency in storing and transferring large amounts of data, which is especially useful in high-load environments where speed and resource efficiency are critical factors. Advantages of protobuf include efficiency (due to serialization to binary or hexadecimal code), portability, and flexibility.

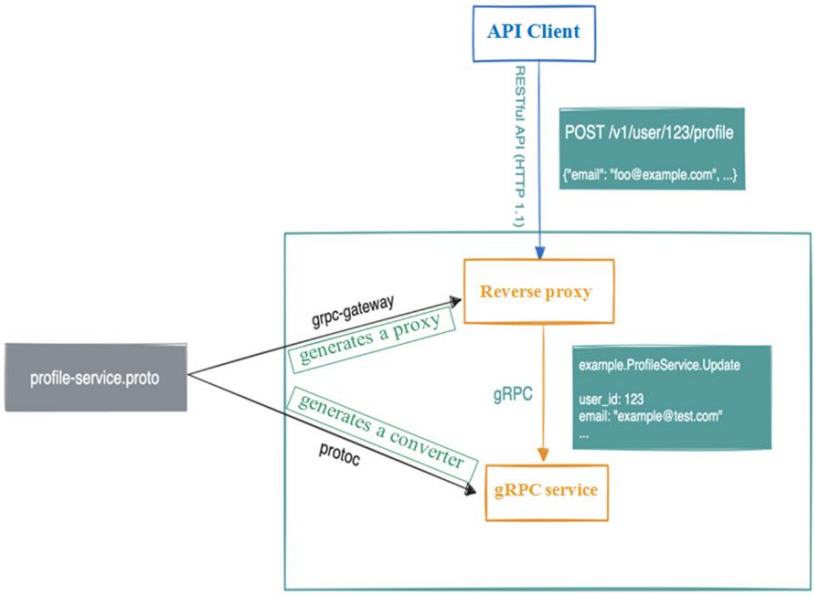

Fig. 3. Scheme of processing requests in the context of the ConnectRPC protocol.

A typical .proto file defines data structures to be serialized and deserialized using Protocol Buffers. The Protocol Buffers Compiler (protoc) tool converts definitions in .proto files into source code of a specific programming language. To generate code in the case of the Go programming language, the following command must be executed:

```
protoc --go_out=. --go_opt=paths=source_relative user.proto
```

When the protoc command is executed, it reads the data structures defined in the .proto file. It generates the appropriate classes and methods for the programming language, including methods to serialize *serializeToString()* and deserialize *ParseFromString()* objects of those classes. In the case of generation, language segments of the selected programming language are formed depending on the libraries.

The serialization process in Protocol Buffers involves converting structured data into a binary format that allows for efficient storage or transfer of that data between different systems or components. Serialized data includes information about field types, their values, and data structure (Fig. 4).

To justify the choice of frameworks based on RPC, we will first compare gRPC and REST according to such criteria as architectural approaches, efficiency and breadth of application in distributed systems [16, 17]. Both methods have advantages and disadvantages, depending on the application's specific needs. Architecturally, REST is significantly overloaded with headers, and its queries can be used to interact with web services that accept data in JSON format for processing or storage. In contrast, gRPC demonstrates a two-way interaction, where the client initiates a request and the server responds. Both use specified methods in their mutual interface.

Using gRPC allows this interaction to be fast, secure, and efficient by compressing data and using HTTP/2 for transport. In this case, communication through procedure calls is more concise, less expensive, and abstracted from the meta-information that REST entails. The comparative characteristics of both methods of data transmission are presented in Table 1.

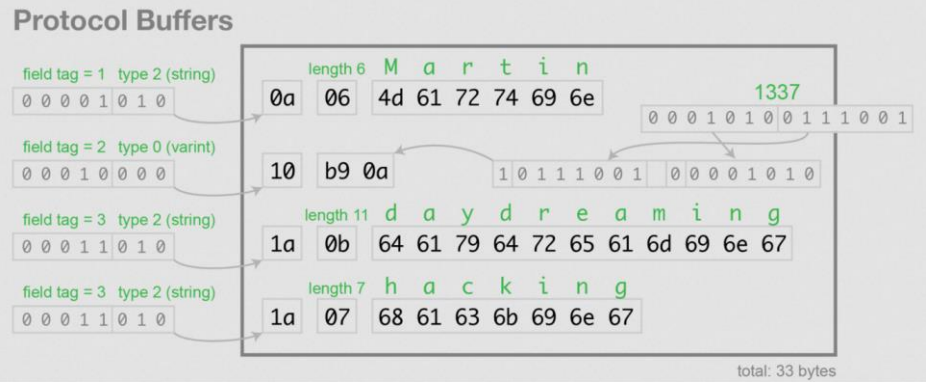

Fig. 4. An example of data serialization in RPC [15].

Because gRPC is implemented over the HTTP/2 transport, the protocol uses multiplexing and data streaming, which are key technologies for reducing latency in modern network protocols such as HTTP/2 used in virtual resource management systems and scalable web applications. These mechanisms enable more efficient use of available network bandwidth, reducing latency and server response time.

For the final selection of the framework for the development of the controller, a comparison was made between the gRPC framework and the REST architecture in terms of performance (response time and total throughput in the case of processing requests), scalability (the ability to adapt to the increase in the number of requests and users without loss of performance), reliability (the ability to withstand high loads and various failures, ensure stability of operation), resource management (efficiency in the use of computing and network resources), compatibility and flexibility (the ability to integrate with different programming languages and platforms easily). Each of these parameters allows a deeper understanding of the advantages and limitations of both protocols in various aspects of their application.

For each study, a specific scenario was chosen, which allows a deeper understanding of the effectiveness of a particular protocol for various tasks. The request processing time estimation model implements a separate program with the same architectural behavior, but under different gRPC and REST protocols. Experiments comparing the performance of REST and gRPC were conducted on a specially prepared virtual environment based on Docker. Each container was configured with the following resources: single-core CPU, 1 GB RAM, 1 GB/s bandwidth. For testing, a program in the Go programming language was used, which allows simulation of the processing of requests under both protocols on the same hardware configuration for a correct comparison. The number of requests for testing varied from 1 to 1000 in steps of 50 requests.

Table 1. Comparative characteristics of REST and gRPC data transfer methods

Parameter	REST	gRPC
Communication protocol	uses HTTP/1.1, which provides versatility and easy integration with web infrastructure, but is less efficient due to header overload and opening a new connection for each request	uses HTTP/2, which supports multiplexing of several requests through one connection, which reduces delays in communication channels
Data format	JSON or the older SOAP (XML) principle, which is a text-based format and is easily readable by humans, but may require more traffic and data to convey specific information	Protocol Buffers, a binary format that is more efficient in terms of size and serialization/deserialization speed
Call methods	operates with CRUD concepts using standard HTTP methods (GET, POST, PUT, DELETE)	allows calling remote procedures directly, which is implemented through defined services and methods in .proto files
Support for streaming	streaming support is possible but not built-in and may require additional implementation	native support for streaming requests and responses, which allows to work more efficiently with large data streams

A pseudo-random number generator was used to generate the load, which provided diversity in the choice of the number of requests, simulating real operating conditions:

```
> requests = np.arange(1, 1001, 50).
```

The model for estimating request processing time in REST and gRPC can be represented as follows:

$$T_{REST} = T_{conn} + T_{send} + T_{proc} + T_{resp};$$
$$T_{gRPC} = T_{send} + T_{proc} + T_{resp},$$

where T_{conn} – connection time; T_{send} – time to send data; T_{proc} – processing time on the server; T_{resp} – time to receive a response. Initially, the estimation of the request processing time and the influence of the number of requests on the response time to the request were considered (Fig. 5).

As seen from Fig. 5, the response time in the case of the REST API protocol increases linearly with the number of requests. This suggests that with each additional request, the overall response time of the system increases, which may be due to HTTP/1.1 limitations such as simultaneous opening of new TCP connections and header overload. In contrast, the curve for the gRPC platform, although it starts with similarly low response time values for a small number of requests, shows a much slower rise in response time as the number of requests increases. This fact indicates the higher performance and efficiency of gRPC, which is explained by the use of

HTTP/2, which supports multiplexing of streams and more efficient data management at the transport layer.

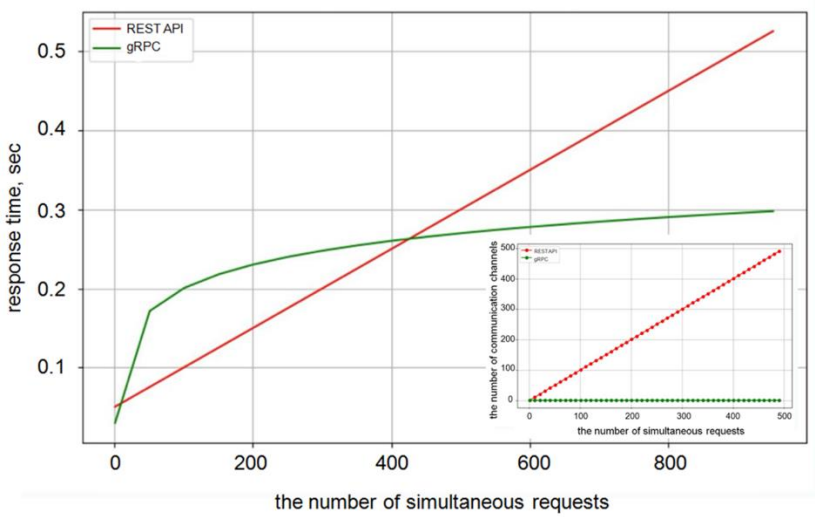

Fig. 5. Dependence of the response time on the number of requests. The inset shows the dependence of the number of used communication channels on the number of simultaneous requests. Both figures show the graphs for the REST API in red and for gRPC they are in green.

The traditional limitations of HTTP/1.1, where each request requires a separate connection, lead to delays with large numbers of concurrent requests. At the same time, gRPC allows multiple requests to be processed simultaneously on the same connection and reduces delays in request processing. As a result, the response time is significantly reduced compared to HTTP/1.1, especially with a large number of requests. This fact is well reflected in the inset of Fig. 7, which shows a linear growth of the number of used channels in sync with the increase in the number of requests for the REST API and the absence of such growth for gRPC.

Next, we will consider the results of research on the change in throughput (the number of processed requests per second) depending on the number of simultaneous requests for both protocols (Fig. 6). As can be seen from Fig. 6, for the REST API, there is a sharp decrease in bandwidth due to the increase in the number of requests. Initially, the curve has high throughput, but drops off quickly as the number of requests increases, indicating that REST becomes significantly less efficient under heavy loads. This may be due to limitations of HTTP/1.1, which do not allow large numbers of concurrent connections to be efficiently scaled over individual TCP connections for each request. In contrast, gRPC shows much better throughput even with increasing number of requests. This demonstrates the high efficiency of gRPC, which uses HTTP/2 to multiplex multiple requests in a single connection, as noted above. This approach allows gRPC to maintain high performance and reduce overall latency, providing more stability when processing a large number of requests.

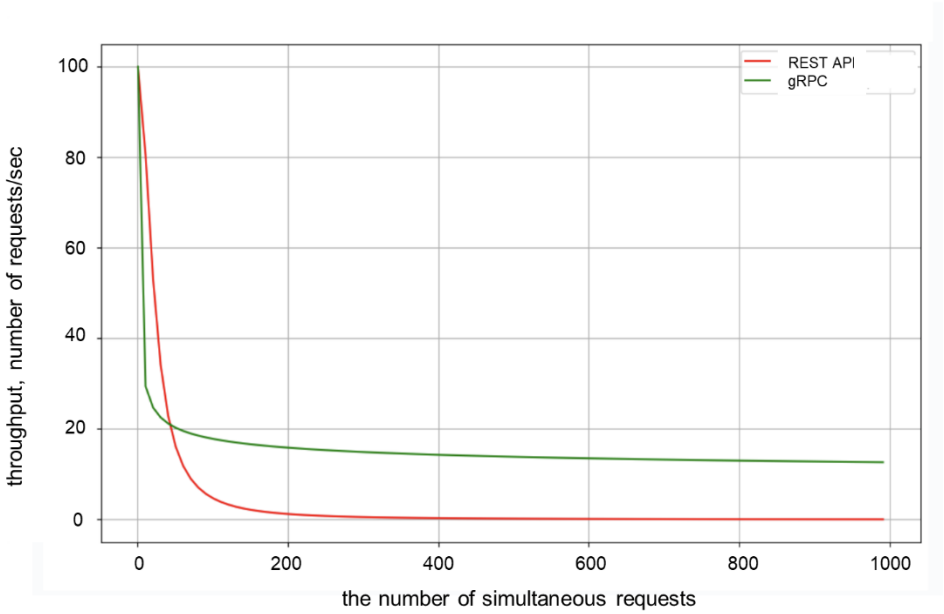

Fig. 6. Dependence of throughput (number of requests per second) on the number of simultaneous requests for REST API (red curve) and gRPC (green curve).

The graphs in Fig. 6 indicate significant differences in scalability and performance between the REST API and gRPC. In particular, the REST API can face challenges in scaling, especially in large distributed systems, where a large number of simultaneous requests can cause significant delays. Instead, gRPC, using HTTP/2, provides much better connection management and lower overall latency, making it an ideal choice for systems where fast response and high throughput are important.

Next, we will consider such a characteristic as the probability of losing requests. Throughput according to the Erlang formula [18] for determining the load is described by the formula:

$$B(E) = \frac{\frac{E^c}{c!}}{\sum_{k=0}^{c} \frac{E^k}{k!}},$$

where E – request intensity, which estimates the average number of concurrently active requests, and c – the number of connection channels, where the numerator represents the probability that exactly c channels are engaged, and the denominator is the sum of the probabilities that any number of channels from 0 to c will be engaged.

Graphs in Fig. 7 show the probability of losing B requests according to Erlang's formula for REST API and gRPC systems, which are simulated with different number of channels: 10 for REST and 50 for gRPC (with an equal number of channels, the difference is even more significant). For REST APIs, the probability of losing requests increases rapidly with increasing

load. For a system with 10 channels (servers), this indicates significant limitations in its ability to handle high loads without losing requests. Almost all requests cannot be processed at high load levels (around 100) due to insufficient resources. In contrast, gRPC exhibits a significantly lower probability of losing requests at the same load level, due to the use of more channels. The likelihood of missing requests increases much more slowly, indicating higher throughput and more efficient resource management.

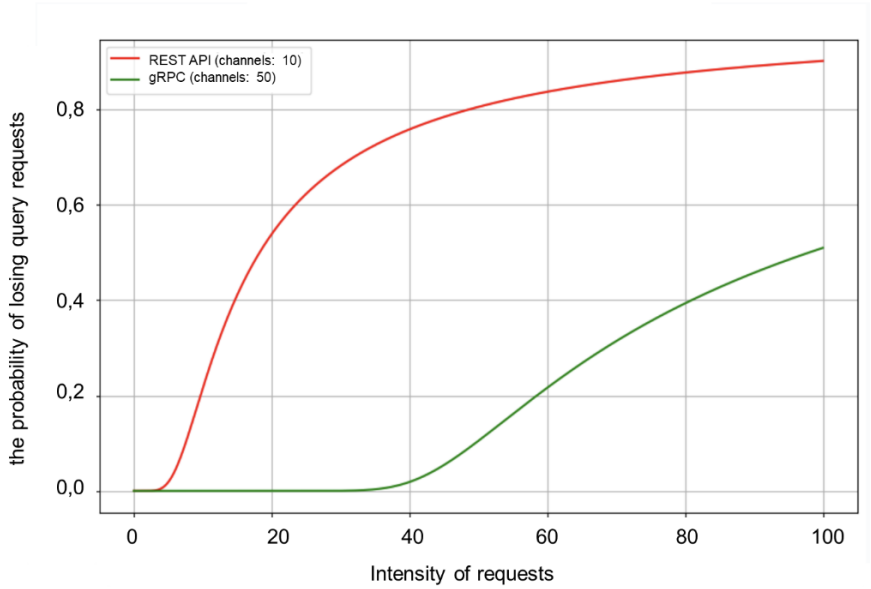

Fig. 7. Request loss probability by Erlang formula for REST API (red curve) and gRPC (green curve).

Fig. 7 demonstrates the importance of architecture and technology choices to ensure high availability and reliability in large-scale applications. gRPC, with its ability to use multiplexing over HTTP/2, provides better performance and a lower chance of losing requests compared to REST, which uses a more traditional single-channel approach with HTTP/1.1.

Therefore, the analysis of the results showed that gRPC provides more stable and significantly higher performance in the case of increased number of requests, due to the efficiency of HTTP/2, which supports multiplexing. At the same time, the REST API using HTTP/1.1 showed an increase in response time as the number of requests increased. In the case of scaling, REST may require additional resources to scale under a high number of requests, which may include extra costs to optimize server hardware and infrastructure. The choice between REST and gRPC depends on the specific application requirements. If performance with a large number of concurrent requests is critical, gRPC may be a better choice. REST may be more suitable for less dynamic or standardized web interfaces where simplicity of implementation and a wide range of client support are required.

Since the comparative analysis of the gRPC and REST protocols revealed significant advantages of the gRPC technology, it determined the choice of the RPC-based framework for developing the controller and the main Golang (Go) programming language. Go includes strong

typing and automatic memory management, which reduces the risk of common programming errors such as buffer overflows and memory leaks. Built-in tools like the race detector help detect race conditions, which is critical for multi-threaded applications typical of gRPC. The combination of simplicity, performance, built-in concurrency and the availability of a wide selection of typical standard library tools make Go an ideal language for developing RPC systems. These aspects allow Go to provide technical efficiency and reduce the overall complexity of projects, contributing to the rapid deployment of reliable and scalable solutions in modern distributed systems [19].

In the context of developing a modern distributed system, the importance of choosing the right technology stack cannot be overstated. Productivity, scalability, support of modern standards, and future flexibility are the main criteria for determining optimal technological solutions. Taking these aspects into account and conducting a comparative analysis of technologies made it possible to choose a combination of RPC, cRPC, HTTP/2 and Go technologies with a protobuff serializer for the implementation of the given task, which meets all modern requirements for speed, efficiency and scalability of a distributed system. This approach provides an optimal solution for ensuring stability, speed and reliability in managing large-scale real-time applications.

Selected technologies are integrated to create a single system:

1. *RPC* and *ConnectRPC* provide reliable communication between system components, simplifying integration and scaling;

2. *Golang* is used to develop the main components of the system, ensuring high performance;

3. *Protocol Buffers* provide fast and efficient data serialization for transmission between system components.

As part of this work, a minimum viable product (MVP) was developed for a software controller that would manage runner resources. The MVP's main purpose is to validate the controller's key functions and check its integration with the existing CI/CD system (Github Actions).

Development of a functional MVP controller for managing runner resources

Let's first consider the architecture of the designed controller. The primary goal in designing the controller architecture is to create a framework that supports runners' core resource management functions while allowing for easy future scalability. The architecture should be modular, allowing new components to be added without rewriting existing code. In general, it should ensure high management efficiency and the ability to adapt to changes in requirements or volume of work quickly. It is also important to consider ensuring proper security and reliability when integrating with other CI/CD systems.

Fig. 8 shows the abstract architecture of the controller's interaction with the runners and describes the mechanisms of processing and managing runner resources.

The main functions of the controller include:

1. *processing requests from new runners* - the controller accepts initialization requests from runners for their registration in the system;

2. *registration of runners in groups* - effective distribution of runners in working groups based on their characteristics and tasks;

3. *authentication of runners* – checking the authenticity of runners before adding them to the pool to ensure security;

4. *protection of connections through SSL* - use of Secure Sockets Layer (SSL) to encrypt data transmitted between the controller and runners, ensuring confidentiality and protection against potential attacks;

5. *event logging* – recording of all runner connection operations for audit and monitoring of activity in the system.

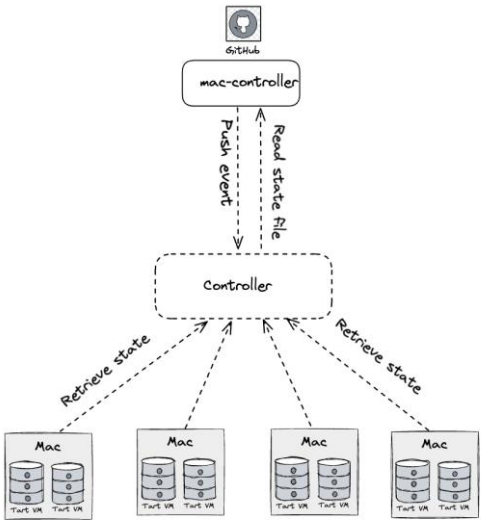

Fig. 8. Abstract architecture of controller interaction with runners.

Integrating the MVP with existing CI/CD systems, such as GHA in particular, is a critical development component. The main goal of integration is to ensure MVP compatibility with a wide range of operations and services already in use in the CI/CD system.

For testing, the MVP involves the use of a variety of methods to verify functionality (the application performs all declared functions), performance (system response time, request processing and performance under load) and product security (vulnerability and potential threats, penetration tests and security audits). An important aspect of development is integration testing, which allows you to make sure that MVP components interact correctly with both internal and external systems. Using automated tools and developing test scenarios will enable to identify problems at the early stages of integration.

Mock services can be used to simulate external interfaces and APIs, allowing integration testing without the need for a real environment. This provides more testing flexibility and helps avoid possible risks for actual operational processes.

The main steps of the process include the development of test scenarios that reflect real operational conditions and integration testing, which allows to evaluate the interaction between the MVP and other system components. It is also important to carry out load testing to determine the performance limits of the controller in the case of different load levels and to determine the points of possible failures.

Within the framework of this investigation, a .proto file was developed that defines the interfaces for the interaction between the controller and the runners through various RPC calls.

The following key functions are implemented: *RegisterRunnerRequest*, *StartJob*, *StopJob*, *CheckRunnerStatus.* In the future, their registration will be carried out through RPC.

Together, these services create a comprehensive interface for managing runners and their tasks, ensuring efficient allocation of resources and tracking the status of execution in the system.

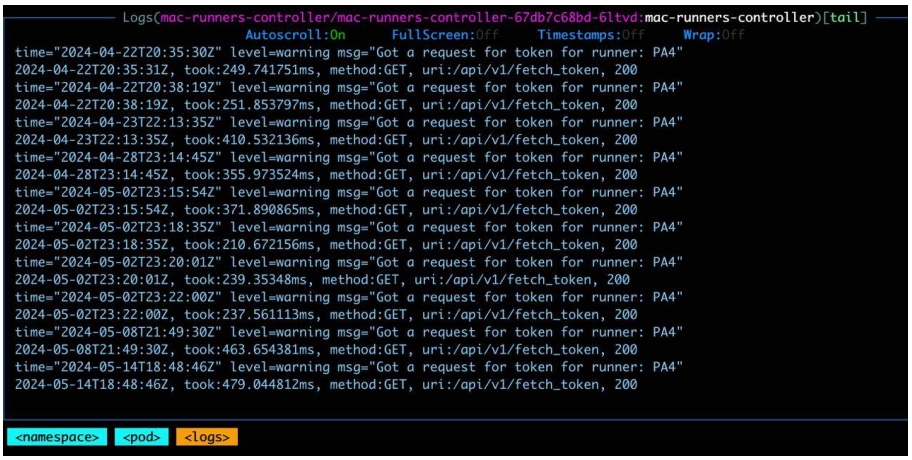

Fig. 9. Screenshot of the console with displayed logs.

Also, because the cRPC framework also implements message delivery through the standard REST interface, it becomes possible to use the controller in two modes by implementing the same procedures, but through REST. Call logging is presented in Fig. 9.

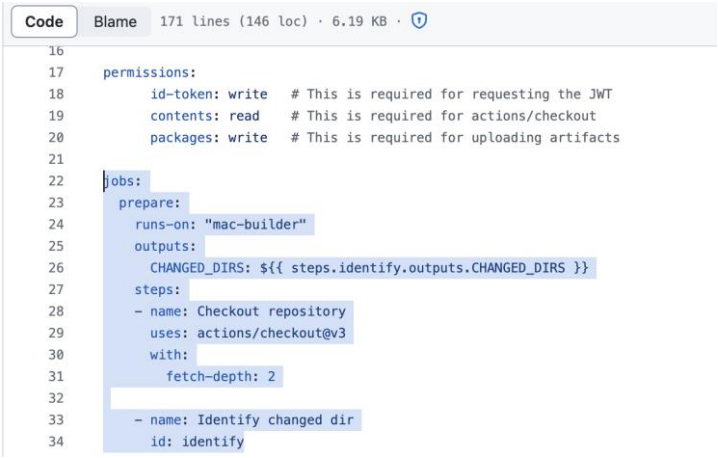

Fig. 10. Listing of *job workflow* instructions.

To test the functionality of tasks (*jobs*), a *job workflow* was created on GHA, the instructions of which are shown in Fig. 10. It allows you to deploy and test runners using a

controller. This *workflow* provides automatic configuration and launch of runners, checking their integration and performance in real conditions.

This *workflow* configures the testing environment. In particular, it uses the *mac-builder* runner, for which the controller automatically deploys the necessary resources and executes test tasks, checking them for correctness and efficiency.

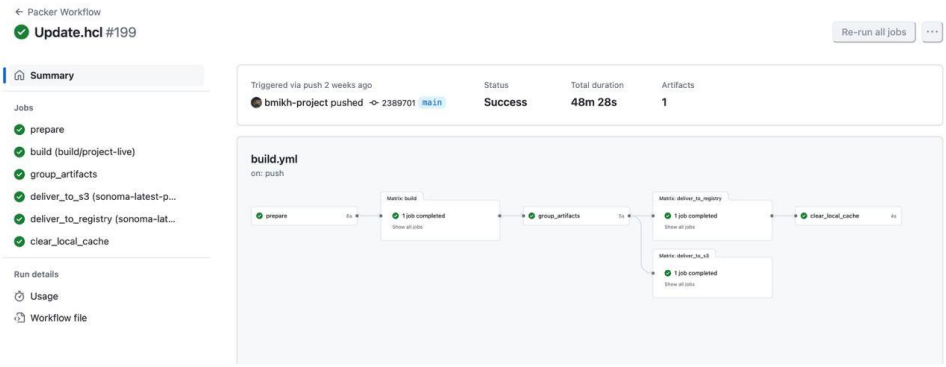

Fig. 11. Screenshot from GHA of the result of a successfully executed *job* task from the controller.

After testing, it can be seen (see Fig. 11) that the integration with GHA works and the tasks are performed successfully. This confirms that the controller interacts correctly with the runners in the GHA environment.

Fig. 12. Displaying the virtual machine window.

Fig. 12 shows a macOS virtual machine [20] where the runner has been successfully launched. This demonstrates that the runner, controlled via RPC commands from the controller, executes tasks in the macOS environment. The environment reflects the macOS user interface.

Thus, a prototype of the runner controller based on RPC technologies has been developed, which confirms the high efficiency of using RPC for interaction between the controller and virtual environments. This, in turn, demonstrates how RPC can serve as a reliable interface for managing resources in virtual environments, providing convenience and efficiency in integration with CI/CD systems. It is worth noting that the considered development provides several opportunities but has certain disadvantages.

Among the possibilities, the following can be distinguished:

- high performance and security thanks to the introduction of such advanced technologies as Connect RPC and OpenSSL;
- flexibility and scalability, which allows you to adapt the system to various production requirements and conditions effectively;
- expanding the potential of using the system in various environments thanks to integration with modern services and platforms;
- adaptability to any system, subject to additional development of the adapter or imitation of the RPC interface.

Disadvantages include:

- management and configuration of the system may require a high level of technical knowledge, which places specific requirements on the qualifications of administrators;
- the need to develop a specialized interface for interaction with virtual platforms, which can complicate integration with various execution environments;
- some virtual environments require additional development of an adapter for the possibility of working with the RPC protocol;
- clear continuous contact with the runner via the Internet is necessary.

The developed controller, created as a minimum viable product (MVP), has significant potential for development and improvement. Possible areas of improvement include optimizing and refactoring of the code, expanding functionality, scalability and integration, increasing the level of security (for example, using advanced encryption technologies and mutual Transport Layer Security (TLS) connection), etc.

Conclusions

Within the framework of the given investigation, a functional model of the controller was developed and an analysis of potential technologies for its implementation was carried out. In particular, a comparative analysis under the conditions of a typical load of RPC and REST API showed that the most effective solution for implementing the controller itself is the cRPC framework, which opens up opportunities to use different virtual environments through the implementation of interfaces or an adapter for RPC. This controller aims to improve resource efficiency and reduce response time, especially under high load conditions, by using modern RPC protocols.

The development opens wide prospects for further improvement and optimization, providing opportunities for integration with other platforms and services, such as GitHub Actions, which can significantly improve CI/CD processes. Scaling and expanding the controller's functionality can increase productivity and efficiency in processing tasks in real-time. These perspectives highlight the project's potential not only as a tool for automation and

resource management, but also as a platform capable of rapid adaptation and expansion in the future, ensuring sustainability and efficiency at a high level.

In general, the implemented development is relevant and promising. It has great potential to become an important tool for organizations that seek to optimize the management of their virtual resources and improve development processes. Thanks to the application of modern technologies and approaches, it provides power and flexibility, making the system a valuable asset for any IT structure.

References

[1] Value and benefits of DevOps in IT companies [Electronic resource]. – Mode of access: https://onecloudplanet.com/blog/article/value-and-benefits-of-devops-in-it-companies (in Ukrainian)

[2] Improving product efficiency through continuous integration and continuous delivery (CI/CD) [Electronic resource]. – Mode of access: https://www.londonproduct.academy/post/pidvishchennya-rivnya-efektivnosti-produktu-za-dopomogoyu-neperervnoyi-integraciyi-ta-neperervnoyi-dostavki-ci-cd (in Ukrainian)

[3] 10 reasons why CI/CD is important for DevOps [Electronic resource]. – Mode of access: https://cloudfresh.com/ua/cloud-blog/10-prichin-chomu-ci-cd-vazhlivi-dlya-devops/ (in Ukrainian)

[4] What is server virtualization and why is it useful? [Electronic resource]. – Mode of access: https://hyperhost.ua/info/uk/shho-take-virtualizaciya-serveriv-i-comu-vona-korisna (in Ukrainian)

[5] *Fowler M.* Continuous Integration [Electronic resource]. – Mode of access: https://martinfowler.com/articles/continuousIntegration.html

[6] *Wolff E.* A Practical Guide to Continuous Delivery / E. Wolff. – Addison-Wesley Professional, 2017. – 288 p.

[7] *Phillips A.* The Continuous Delivery Pipeline – What it is and Why it's so Important in Developing Software [Electronic resource]. – Mode of access: https://devops.com/continuous-delivery-pipeline/

[8] About GitHub Actions Runners [Electronic resource]. – Mode of access: https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners/about-github-hosted-runners

[9] GitHub Actions Documentation [Electronic resource]. – Mode of access: https://docs.github.com/en/actions

[10] *Nelson B. J.* Remote Procedure Call (PhD thesis) / B. J. Nelson. – Xerox Palo Alto Research Center, 1981. – 220 p.

[11] Introducing gRPC, a new open-source HTTP/2 RPC Framework [Electronic resource]. – Mode of access: https://developers.googleblog.com/en/introducing-grpc-a-new-open-source-http2-rpc-framework/

[12] gRPC Documentation [Electronic resource]. – Mode of access: https://grpc.io/docs/

[13] Connect RPC Documentation [Electronic resource]. – Mode of access: https://connectrpc.com/docs/go/interceptors

[14] protocolbuffers/protobuf [Electronic resource]. – Mode of access: https://github.com/protocolbuffers/protobuf

[15] Schema evolution in Avro, Protocol Buffers and Thrift [Electronic resource]. – Mode of access: https://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html

[16] *Bolanowski M.* Efficiency of REST and gRPC realizing communication tasks in microservice-based ecosystems / M. Bolanowski, K. Zak, A. Paszkiewicz, M. Ganzha, M. Paprzycki, P. Sowinski, I. Lacalle, and C. E. Palau. – arXiv preprint arXiv:2208.00682, 2022. – 12 p.

[17] GRPC Vs REST API Communication [Electronic resource]. – Mode of access: https://www.wallarm.com/cloud-native-products-101/grpc-vs-rest-api-communication

[18] *Allen A. O.* Probability, Statistics, and Queueing Theory with Computer Science Applications / Arnold O. Allen. – Academic Press, 1978. – 406 p.

[19] The Go programming language: perspectives, pros and cons [Electronic resource]. – Mode of access: https://senior.ua/articles/mova-programuvannya-go-perspektivi-plyusi-ta-mnusi (in Ukrainian)

[20] MacOS Virtualization Framework [Electronic resource]. – Mode of access: https://developer.apple.com/documentation/virtualization

РОЗРОБКА ПРОГРАМНОГО КОНТРОЛЕРА ДЛЯ АВТОМАТИЗОВАНОГО СТВОРЕННЯ ТА КЕРУВАННЯ ВІРТУАЛЬНИМИ РЕСУРСАМИ РАННЕРІВ

Ю. Корчак, Б. Міх, Ю. Фургала

Львівський національний університет імені Івана Франка,
вул. Ген. Тарнавського, 107, 79017 Львів, Україна
yuriy.korchak@lnu.edu.ua

У роботі розроблено функціональну модель (прототип) контролера як мінімально життєздатний продукт (MVP) для автоматизованого створення та керування віртуальними обчислювальними ресурсами раннерів.

Порівняльний аналіз за умови типового навантаження протоколів RPC та REST API щодо продуктивності, масштабованості, надійності, керуванням ресурсами, сумісності та гнучкості показав, що найбільш ефективним рішенням задля імплементації самого контролера є фреймворк на базі RPC, а саме cRPC. Експериментально доведено, що протокол RPC може слугувати надійним інтерфейсом для керування ресурсами у віртуальних середовищах, забезпечуючи зручність та ефективність в інтеграції з системами CI/CD. Проведений аналіз інших потенційних технологій для досягнення поставленої мети зумовив обрати комбінацію мови програмування Golang та серіалізатора Protocol Buffers, яка відповідає всім сучасним вимогам до швидкості, ефективності та масштабованості розподіленої системи.

У рамках цієї розробки створено файл .proto, який визначає інтерфейси для взаємодії між контролером та раннерами через різні виклики RPC. Використання тестових сценаріїв дозволило відображати реальні операційні умови та інтеграційне тестування і, як результат, оцінити взаємодію між MVP та іншими компонентами системи. Проведене тестування

засвідчило, що інтеграція з GitHub Actions працює і завдання щодо підвищення ефективності використання ресурсів і зниження часу відгуку, особливо в умовах високих навантажень, виконуються успішно.

Розробка відкриває широкі перспективи для подальшого вдосконалення та оптимізації, надаючи можливості для інтеграції з іншими платформами та сервісами, такими як GitHub Actions, що може суттєво покращити процеси CI/CD. Масштабування і розширення функціональності контролера можуть забезпечити збільшення продуктивності та ефективності обробки завдань у реальному часі. Ці перспективи підкреслюють потенціал розробки не тільки як інструменту для автоматизації та керування ресурсами, але і як платформи, здатної до швидкої адаптації та розширення в майбутньому, забезпечуючи стійкість і ефективність на високому рівні.

Ключові слова: парадигма неперервної інтеграції, парадигма неперервної доставки, протокол віддаленого виклику процедури, gRPC, cRPC, протокол обміну даними HTTP/2, мова програмування Golang, середовище GitHub Actions, раннер, контролер, мінімально життєздатний продукт.

The article was received by the editorial office on 30.07.2024.
Accepted for publication on 27.08.2024.