SSMMP: a Simple Protocol for Service Mesh

Stanislaw Ambroszkiewicz
Institute of Computer Science
University of Siedlce
Poland
e-mail: sambrosz@gmail.com, ORCID 0000-0002-8478-6703

***Abstract*—The specification of SSMMP (a simple Service Mesh management protocol) is presented. It consists of the formats of messages and the actions taken by senders and recipients. The idea is that microservices of Cloud-Native Applications should also be involved in configurations of their communication sessions. It does not interfere with the business logic of the microservices and requires only minor and generic modifications of the microservices codebase, limited only to network connections. This idea was also applied in Netflix [1], however, at the software level. Thus, sidecars are not needed, which aligns with the current trends, e.g., Cilium Service Mesh. The formal specification of SSMMP is at GitHub [2], as well as a prototype implementation for a simple social media CNApp. It clearly proves that SSMMP should be viewed (by developers) as an integral part of CNApps.**

***Keywords*—Service Mesh, cloud-native applications, abstract architecture, management protocols.**

I. Introduction

Microservices (as a software architecture) were first developed from the service-oriented architecture (SOA) and the concept of Web services (HTTP and WSDL) by Amazon in the early 2000s. Hence the name AWS, which is short for Amazon Web Services. Perhaps Amazon didn't invent microservices alone. However, AWS became the most successful application of microservices for Cloud computing at that time.

Microservice architecture comprises fine-grained services and lightweight protocols. The architecture inherited the HTTP protocol (in the form of REST) from Web services as the basic means of data transport between microservices. Cloud Native Application (CNApp) is a distributed application composed of microservices and deployed in the Cloud.

Microservices constitute an architectural pattern where a complex and sophisticated application (CNApp) is made up of a collection of fine-grained, self-contained microservices that are developed and deployed independently of each other. They communicate over the network using protocols in accordance with the business logic of the application.

Once a collection of such microservices is composed and orchestrated into a dynamic workflow, it can be deployed on a cloud infrastructure.

Contemporary CNApps, developed and deployed by Big Tech, consist of thousands of microservices. For example, Uber [3]: *"Each and every week, Uber's 4,500 stateless microservices are deployed more than 100,000 times by 4,000 engineers and many autonomous systems. These services are developed, deployed, and operated by hundreds of individual teams working independently across the globe. The services vary in size, shape, and functionality; some are small and used for internal operations, and some are large and used for massive, real-time computation."*

The scale and complexity of CNApps force the transformation of microservices from an architectural style to an organizational style, see Ibryam and Losio 2024 [4]. It is called hyperspecialization of cloud services. *" A microservice will no longer be just a single deployment unit or process boundary but a composition of functions, containers, and cloud constructs, all implemented and glued together in a single language chosen by the developer. The future is shaping to be hyperspecialized and focused on the developer-first cloud."*

Hence, a contemporary challenge in IT is to automate the deployment and management of huge and complex CNApps. This very automation is supposed to be designed by developers.

A. The problem and related work

How to automate the execution, scaling, and reconfiguration of Cloud-Native Apps in a general way, but not at the software level? Following Mulligan 2023 [5], this automation can be accomplished by implementing a generic protocol that extends the networking stack on top of TCP/IP.

The solution we propose is the Simple Service Mesh Management Protocol (SSMMP) as a specification to be implemented in a Cloud cluster. The specification consists of the formats of messages exchanged between the parties (actors) to the conversation of the protocol, and the actions taken by the senders and receivers of the messages. The actors are: the Manager, agents (residing on the nodes that make up the cluster), and instances of microservices running on these nodes. All these actors are almost the same as in Kubernetes clusters. The main difference is the abstract architecture of CNApps (introduced in Section II), and simple general rules allowing for the automation of CNApps management.

Let's take a brief look at the current work on this topic. Service Mesh is an infrastructure for CNApps that allows transparently adding security, observability and management of network traffic between the microservices without interfering with the codebase of the microservices. Usually, Service Mesh is built on top of Kubernetes and Docker.

Each microservice is equipped with its own local proxy (called a sidecar). Sidecars can be automatically injected into

Kubernetes pods and can transparently capture all microservice traffic. The sidecars form the data plane of Service Mesh.

The control plane of Service Mesh is (logically) one manager responsible for configuring all proxies in the data plane to route traffic between microservices and load balancing, and to provide resiliency and security.

Linkerd [6] and Istio [7], both extending Kubernetes, are the best known and most popular open source software platforms for realizing Service Mesh. Istio uses Envoy's proxy [8], while Linkerd uses its own specialized micro-proxies.

Cilium [9] is also an open source software platform for cloud native environments such as Kubernetes clusters.

While all modern Service Meshes are developed as open-source software, the recent idea (see, e.g., Mulligan 2023 [5]) that the service mesh is now becoming a part of the networking stack is extremely interesting. It should be emphasized that the networking stack is primarily based on protocol specifications, not software.

Let's present the idea of our SSMMP. There are no sidecars, and no proxies. Each microservice instance communicates (according to SSMMP) directly with the agent running on the same host.

Execution of microservices, their replications and closing are controlled and monitored by Manager via its agents. A similar idea is also in Durán and Salaün 2016 [10]. Communication sessions between microservices (determined by the CNApp business logic) are controlled and monitored by the Manager through its agents.

Each communication session is (like in TCP) connection-oriented. A connection between client and server needs to be established before data can be sent during the session. The server is listening for clients. Dynamic management of such communication sessions is the essence of the proposed protocol.

A brief description of the protocol is provided in the next Sections II and III. Then, the generic functionality of the protocol is presented in Section IV. Our paper [11] arXiv: 4889471 provides the complete formal specification of protocol messages and the corresponding actions to be performed. The final Section V provides a short summary.

II. Microservices

CNApp is a network application where microservices communicate with each other by exchanging messages (following CNApp's business logic) using dedicated, specific protocols implemented on top of the network protocol stack. This is usually TCP/UDP/IP. Due to its ubiquity, HTTP, implemented on top of TCP/IP, can also be used as a transport protocol for these messages. Each of these protocols is based on the client-server model of communication. This means that the server (as part of a running microservice on a host with a network address) is listening on a fixed port for a client that is a part of another microservice, usually running on a different host. Since a client initiates a communication session with the server, this client must know the address and port number of the server. A single microservice can implement and participate in many different protocols, acting as a client and/or as a server. Thus, a microservice can be roughly defined as a collection of servers and clients of the protocols it participates in, and its own internal functionality (business logic). Usually, communication protocols (at the application layer) are defined as more or less formal specifications, independent of their implementations. API (in the context of Cloud computing) is a modern reincarnation of this very communication protocol in distributed systems; see, e.g., https://aws.amazon.com/what-is/api/ and https://thenewstack.io/what-is-api-management/.

Let *the protocol* be denoted as two closely related parties to the conversation: the server S and the client P, which are to be implemented in two microservices. Formally, let protocol be denoted as (P, S) with appropriate superscripts and/or subscripts if needed. After implementation, they are integral parts of microservices that communicate using this protocol.

Abstract inputs of a microservice can be defined as a collection of the servers (of the protocols) it implements:

$$IN := (S_1, S_2, \dots S_k)$$

Abstract outputs of a microservice are defined as a collection of the clients (of the protocols) it implements:

$$OUT := (P'_1, P'_2, \dots P'_n)$$

To avoid confusion, the server and client parts of a protocol will be renamed. Components of abstract input will be called *abstract sockets*, whereas components of abstract output will be called *abstract plugs*.

Let us formalize the concept described above. *A microservice* is defined as follows:

$$A := (IN, \mathcal{F}, OUT)$$

where IN is the abstract inputs of the microservice, OUT is the abstract outputs, and $\mathcal{F}$ denotes the business logic of the microservice. Incoming messages, via abstract sockets of IN or/and via abstract plugs of OUT, invoke (as events) functions that comprise the internal functionality $\mathcal{F}$ of the microservice. This results in outgoing messages sent via IN or/and OUT.

Generally, we distinguish three kinds of such microservices.

1) The first one is for API Gateways. They serve as the entry points of CNApp for users. Usually, IN of API Gateway has only one element. Its functionality comprises forwarding users' requests to the appropriate microservices. Therefore, API Gateway is supposed to be stateless.
2) The second kind consists of regular microservices. Their IN and OUT are not empty. These microservices are also supposed to be stateless. Persistent data (states) of these microservices should be stored in backend storage services (BaaS).
3) The third kind is for backend storage services (BaaS) where all data and files of CNApp are stored. Their OUT is empty.

III. Abstract Architecture

An abstract plug (of one microservice) can be associated with an abstract socket (of another microservice) if they are two complementary parties of the same communication protocol.

CNApp may be abstracted to a directed acyclic graph representing a workflow composed of microservices of this CNApp. The edges of the graph are of the form (abstract plug → abstract socket). They are directed, which means that a client (of a protocol) can initiate a communication session with a server of the same protocol. These directions do not necessarily correspond to the data flow. This means that if a communication session is established, data (protocol messages) can also flow in the opposite direction, i.e., from an abstract input (abstract socket) to an abstract output (abstract plug).

Abstract graph of CNApp is defined as the following directed labeled multi-graph.

$$\mathcal{G} := (\mathcal{V}, \mathcal{E})$$

where $\mathcal{V}$ and $\mathcal{E}$ denote Vertices and Edges, respectively.

- Vertices $\mathcal{V}$ is a collection of names of services of CNApp.
- Edges $\mathcal{E}$ is a collection of labeled edges of the graph. Each edge is of the form:

  $$(C,\ (P, S),\ D)$$

  where C and D belong to $\mathcal{V}$, and (P, S) denotes a protocol. That is, P belongs to OUT of C, and S belongs to IN of D. Hence, the edges correspond to *abstract connections* between microservices. The direction of an edge represents the client-server order of establishing a concrete connection. There may be multiple edges (abstract connections) between two vertices.

The above graph is an abstract view of a CNApp. Vertex is a service name, whereas an edge is an abstract connection consisting of the names of two services and the name of a communication protocol between them.

Initial vertices of the abstract graph correspond to API Gateways (entry points for users), whereas the terminal vertices correspond to backend storage services (BaaS) where all data and files of the CNApp are stored.

The vertices representing regular microservices are between the API gateways and the backend storage services (BaaS).

Scaling through replication and reduction (closing replicas) of a service forces it to be stateless. The reason is that if the service is stateful, then closing (crashing) a replica causes it to lose its state. We assume that API Gateways and regular microservices are stateless and can be replicated, i.e., multiple instances of such a service can run simultaneously.

To run CNApp, instances of its services must first be executed, then abstract connections can be configured and established as real connections, and finally, protocol sessions (corresponding to these connections) can be started.

Some services and/or connections may not be used by some executions of CNApp. Temporary protocol sessions can be started for already established connections (and then closed along with their connections) dynamically at runtime. Multiple service instances may be running, and some are shutting down. This requires dynamic configurations of network addresses and port numbers for plugs and sockets of the instances. The novelty of SSMMP lies in the smart use of these configurations. A similar idea has been used by Netflix [1] at the software level.

IV. Simple Service Mesh Management Protocol - SSMMP

The complete formal specification of SSMMP and its prototype implementation are on GitHub [2]. Here we will present the protocol in an intuitive, somewhat informal way. The main actors of the protocol are: the Manager, the agents, and the running instances of services (API Gateways, regular microservices, and BaaS services), see Fig. 1.

There may be two (or more) running instances of the same service. Hence, the term service refers rather to its bytecode.

The Manager communicates only with the agents. An agent, on a node, communicates with all service instances running on that node. Within the framework of SSMMP, any service instance (running on a node) can only communicate with its agent on that node.

An agent has a service repository at its disposal. It consists of bytecodes of services that can be executed (as service instances) on this node by the agent. The agent (as an application) should have operating system privileges to execute applications and to kill application processes. An agent acts as an intermediary in performing the tasks assigned by the Manager. All service instance executions, as well as shutting down running instances, are controlled by the Manager through its agents.

Each agent must register with the Manager so that the network address of its node and its service repository are known to the Manager.

At the request of the Manager, the agent can execute instances of services whose bytecodes are available in its repository or shut down these instances.

Once a service instance is executed, it initiates the SSMMP communication session with its agent.

The agent can monitor the functioning of service instances running on its node (in particular, their communication sessions) and report their status to the Manager.

The Manager can also shut down (via its agent) a running instance that is not being used, is malfunctioning, or is being moved to another node.

Usually, in the existing service meshes, the Manager controls the execution of CNApps in accordance with a policy defined by the Cloud provider.

In SSMMP, the design and implementation of an instance of Manager is delegated to the developer of CNApp, who can take into account the cloud provider's policies. This makes this instance (dedicated to this CNApp) an integral part of the CNApp. The current state of the Manager, as well as its history, is stored in a dedicated database DB. The Manager knows the service repositories of all its agents.

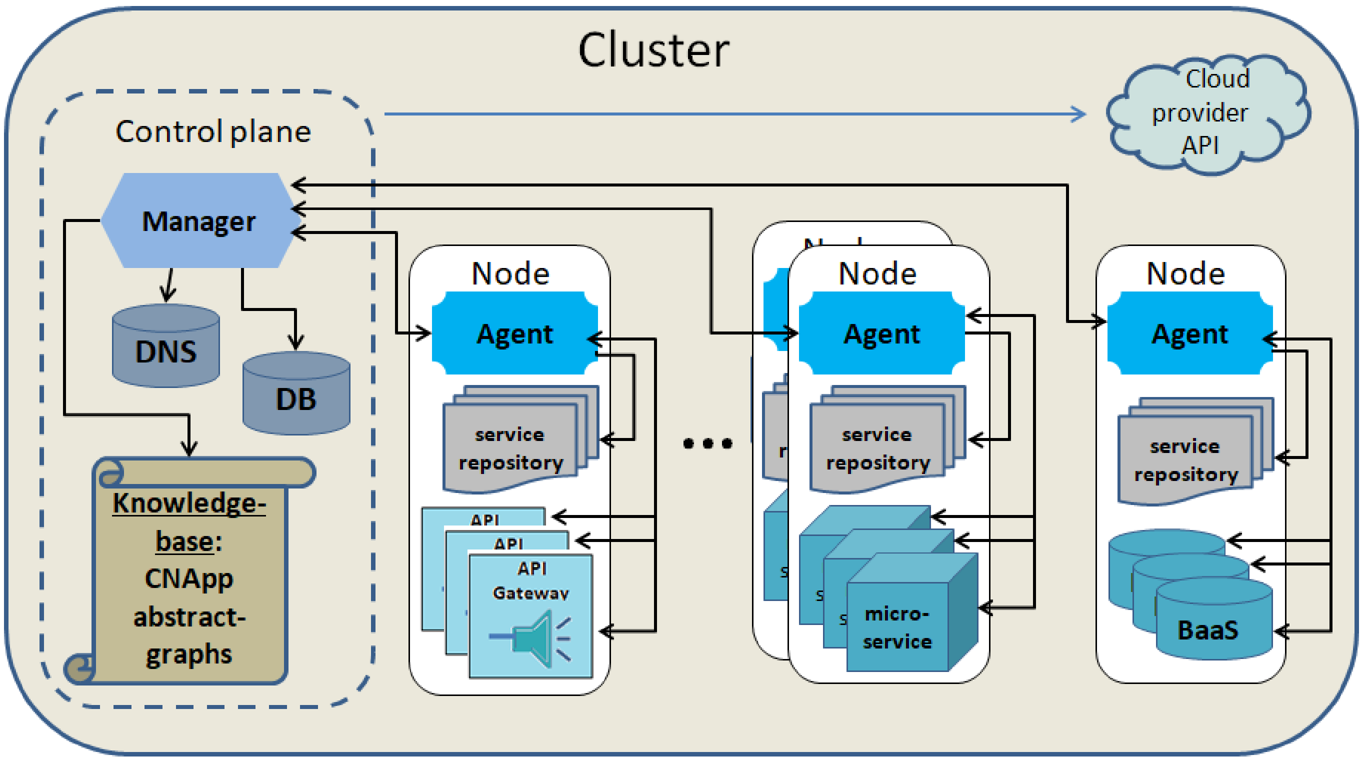

Fig. 1. Simple protocol to automate the executing, scaling, and reconfiguration of Cloud-Native Apps

The Knowledge base of the Manager consists of abstract graphs of CNApps, i.e., the CNApps that can be deployed on the cluster comprising all the nodes.

The current state of any running instance of the service is stored in the Manager's database, and consists of: open communication sessions and their load metrics; and observable (health, performance, and security) metrics, logs, and traces.

The key element of SSMMP is the concept of *a communication session*, understood jointly as establishing a connection and then starting a protocol session on this connection.The process of establishing and closing such sessions is controlled by the Manager through its agents.

V. Summary

SSMMP is simple if we consider its description presented above, and especially the complete formal specification at GitHub [2] with the complete Java API.

The concept of an abstract connection between services (in the abstract graph of CNApp) and its implementation as communication sessions is crucial. The abstract definition of the service of CNApp is also important here. Separation of these abstract notions from deployment is important.

The novelty of SSMMP consists in the dynamic establishment and the closing of communication sessions at runtime based on the configurations assigned to sockets and plugs by the Manager.

Although a similar approach has already been used in Netflix [1] (as a dedicated software), it can be fully exploited in Netflix by extending the network protocol stack with SSMMP.

Since executing, scaling, and reconfiguration of CNApp can be done by SSMMP, it seems reasonable to include SSMMP as an integral part of CNApp. Crash recovery for CNApp can also be performed using SSMMP.

References

[1] D. Vroom, J. Mulcahy, L. Yuan, and R. Gulewich, "Zero Configuration Service Mesh with On-Demand Cluster Discovery, https://netflixtechblog.com/zero-configuration-service-mesh-with-on-demand-cluster-discovery-ac6483b52a51," Aug 30, 2023.
[2] S. Ambroszkiewicz, "GitHub: SMMP," May 8, 2024. [Online]. Available: https://github.com/sambrosz/SSMMP-a-simple-protocol-for-Service-Mesh-management
[3] M. Schwarz and A. Neverov, "Up: Portable Microservices Ready for the Cloud," Sep 7, 2023. [Online]. Available: https://www.uber.com/en-PL/blog/up-portable-microservices-ready-for-the-cloud/
[4] B. Ibryam and R. Losio, "Cloud-Computing in the Post-Serverless Era: Current Trends and beyond," Jan 22, 2024. [Online]. Available: https://www.infoq.com/articles/cloud-computing-post-serverless-trends/
[5] B. Mulligan, "The Future of Service Mesh is Networking," February 24, 2023. [Online]. Available: https://www.infoq.com/articles/service-mesh-networking/?utm_source=email&utm_medium=cloud&utm_campaign=newsletter&utm_content=02282023
[6] Linkerd, "A different kind of service mesh," 2023. [Online]. Available: https://linkerd.io/
[7] "The Istio service mesh," 2023. [Online]. Available: https://istio.io/
[8] "Envoy," 2023. [Online]. Available: https://www.envoyproxy.io/
[9] "Cilium – eBPF-based Networking, Observability, Security," 2022. [Online]. Available: https://cilium.io/
[10] F. Durán and G. Salaün, "Robust and reliable reconfiguration of cloud applications," *Journal of Systems and Software*, vol. 122, pp. 524–537, 2016.
[11] S. Ambroszkiewicz and W. Bartyna, "A simple protocol to automate the executing, scaling, and reconfiguration of cloud-native apps," 11 May 2023. [Online]. Available: https://arxiv.org/abs/2305.16329