Analyzing Inter-Service Communication Paradigms in Microservice Architectures

Md. Abdul Momin[1], Md. Musharaf Hussain[2], Quazi Abdur Rokib[3], Md. Ezharul Islam[4]
[1,2,4]*Department of Computer Science and Engineering, Jahangirnagar University, Dhaka*
[3]*Department of Computer Science and Engineering, University of Kalyani*

Abstract - ***Inter-service communication (ISC) is a key part of microservice architecture. It allows different services to work together and complete complex business tasks. This paper reviews the main ISC methods, such as synchronous types like REST and gRPC, and asynchronous types like message queues. It also studies how these methods affect system performance, scalability, and fault tolerance by analyzing existing research and comparing various communication protocols and frameworks. The paper provides strategic recommendations to guide architects in selecting the optimal ISC method to build robust, secure, and scalable microservice-based systems. The analysis concludes that the selection of an ISC strategy is highly context-dependent, with no universal solution, and that a hybrid approach is often necessary to meet the competing demands.***

Keywords- Inter-Service Communication (ISC), Messaging Protocol, Network Protocol, Publish/Subscribe, Remote Procedure Call (RPC).

1. INTRODUCTION

Inter-service communication (ISC) is a critical parts of microservice architectures. This parts serving as the backbone that assists discrete services to collaborate in executing complex business processes. The paper studies challenges like network delay and security issues. It studies the use of protocols, messaging, and remote calls for reliable communication. The paper concludes with strategic recommendations for selecting the optimal ISC method for better performance, scalability, and security. A microservice is an independently deployable component of bounded scope that supports interoperability through message-based communication. Microservice architecture is a style of engineering highly automated, evolvable software systems made up of capability-aligned microservices [1]. Microservices have many known benefits. However, designing fast and reliable communication between them is a big challenge. It requires a careful review of current methods and their pros and cons. Each service can scale up or down based on its workload, using resources better and improving performance. If one service fails, the problem stays within that service. The other services keep running normally. This enhances the overall application's robustness and reliability. They also promote agility, faster deployment, and innovation by giving teams flexibility in development and technology choices [2]. These features make microservices ideal for complex and evolving systems that demand speed, growth, and fault tolerance [3].

1.1 Importance of ISC

Inter-service communication (ISC) is key in microservices. It facilitates seamless service collaboration. Unlike monolithic apps, microservices handle specific tasks, and ISC helps them share data and requests. Without effective communication between services, they become isolated and the user experience breaks down. Business processes can also be disrupted. Good communication helps isolate faults so one service's failure doesn't affect the whole system. This helps the application stronger and more reliable.

1.2. Scope of the Paper

This paper analyses Inter-Service Communication (ISC) in microservices. It examines various service collaboration patterns. It analyzes the advantages, limitations, and optimal use cases for each approach. The paper also shows how ISC affects performance, scalability, fault tolerance, and maintenance. The paper aims to guide developers and researchers in selecting the most appropriate ISC approaches for their systems to optimize microservices design and

implementation. Through this review, the paper compares different ISC methods and their applicability in various contexts, providing insights to enhance decision-making in the development and use of microservices.

2. BACKGROUND

Microservices, though beneficial, pose communication challenges. Unlike monolithic systems, they depend on networks, causing delays and potential issues. Efficient communication needs robust service discovery, load balancing, retries, and fallbacks for resilience. Security requires strong authentication and encryption. Data consistency, compatibility during updates, and monitoring further add complexity. The main challenge is the increased complexity due to their distributed nature, as microservices communicate via APIs, requiring more design and management effort than monolithic systems [4]. APIs are key for detecting anomalies in microservices' traffic. Continuous monitoring helps identify intrusions and prevent attacks. However, rule-based methods struggle to keep up with frequent changes in dynamic development environments [5].

2.1 Challenge in ISC

A major challenge in interservice communication is the impact of network latency on system performance.In microservices, network delays can accumulate, slow the request processing and decrease user experience, especially when number of microservice and interaction grow[6]. Reliable communication faces issues like latency and message overhead. Strong service discovery, routing, and load balancing can be used to avoid slowdowns. Distributed transactions make systems more complex and need proper synchronization. Security also requires strong authentication and encryption. Real-time monitoring and debugging tools are important to keep the system clear and reliable. Though microservices excel in distributed interactions, effective ISC management is essential for success.

3. COMMUNICATION PROTOCOL AND TECHNOLOGIES

Communication protocols can dictate the transmission order among network nodes, leading to specific scheduling behaviors that complicate the analysis and synthesis of network systems (NSs) [7]. Services in a microservice system need to talk to each other. Each service use different communication protocols and technologies to share data reliably and get work done. They ensure reliable data transmission and enable service interoperability. These tools support both worldwide internet use and local communication in many fields.

3.1 Network protocols

Network protocols are the base of communication between services. They help send and receive data reliably over the network. Network protocols works for the data is packaged, addressed, transmitted, and delivered. They provide the basic setup needed for communication. As shown in Figure 1, network protocols form the foundation. A comparison of key protocols is provided in Table 1

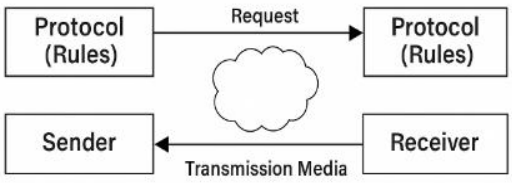

Figure 1. Network protocol (Client-Server communication)

Table 1. Comparison of various network protocol

Protocol	Performance	Scalability	Security	Use Cases
HTTP (Hypertext Transfer Protocol) is the web's communication language, governing how browsers and servers interact.	HTTP/1.1 has higher overhead compared to binary protocols, while HTTP/2 offers significant performance improvements.	High	Can be implemented (HTTPS)	Web browsing, RESTful APIs, Application data transfer
UDP (User Datagram Protocol) is a connectionless protocol that offers faster, but less reliable, data delivery than TCP.	High speed (lower overhead), Variable range (Depends on the underlying network)	High	Limited (no error checking)	Real-time data transfer (e.g., audio/video

				streaming), Simple data exchanges
TCP is a connection-oriented protocol that ensures reliable data delivery by establishing a connection between devices before data transmission.	Good (reliable)	High	Relies on TLS for encryption.	Reliable data transfer (web browsing, email), application communication

3.2 Messaging protocols

In message queues help applications communicate by organizing messages in a line. Every message representing a task to be processed. They carry information such as commands for tasks [8]. Messaging protocols help services communicate asynchronously. This is important for scalability and reducing dependency in microservices. They define how messages are structured and shared. These protocols often use message queues, topics, and subscriptions. This ensures reliable delivery, even if some services are down or busy. . A comparison of key messaging protocols is provided in Table 2.

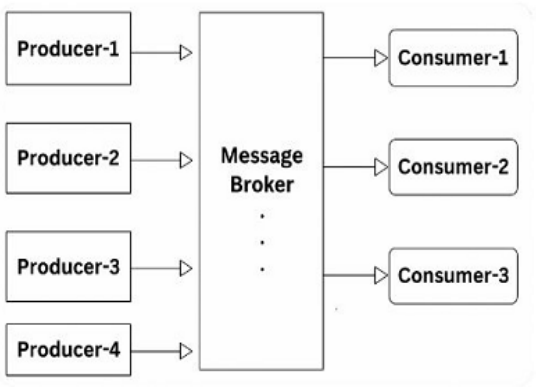

Figure 2. Messaging model (Producer to Consumer)

Table 2. Comparison of various messaging protocols

Protocol	Performance	Scalability	Security	Use Cases
MQTT (Message Queuing Telemetry Transport) is a messaging protocol for the Internet of Things (IoT) that uses a publish-subscribe model	Moderate speed, Variable range (Depends on the underlying network)	High (Pub/Sub, Lightweight)	Can be implemented (TLS)	IoT applications, Sensor data transmission, Machine-to-Machine communication
Kafka is a distributed streaming platform that manages real-time data using a messaging system	High speed, High throughput	High (Distributed architecture)	Can be implemented (Authentication, Authorization)	Real-time analytics, Log aggregation, Streaming data pipelines, High-volume data ingestion
Apache Pulsar is an open-source distributed messaging and streaming platform originally developed at Yahoo and later adopted by the Apache Software Foundation.	High speed, High throughput	High (Distributed, Multi-tenant)	Can be implemented (Authentication, Authorization)	Real-time analytics, Event streaming, Cloud-native messaging, Multi-tenant deployments
While MQTT is suited for resource-limited environments, STOMP is designed for real-time communication in web applications and messaging clients.	Moderate speed (uses underlying transport)	High (Flexible messaging)	Can be implemented (depends on transport)	Message broker communication, Enterprise messaging applications, Bridging between messaging systems
WebSocket is a protocol that enables full-duplex communication over a single, persistent connection between clients and servers.	Moderate speed (full-duplex connection)	High (Persistent connections)	Can be implemented (TLS/WSS)	Real-time bidirectional communication, Web applications (chat, collaboration), Push notifications
AMQP (Advanced Message Queuing Protocol) is a messaging protocol that facilitates communication between applications or services.	Good	High	Yes, via TLS/SSL and SASL.	Reliable message exchange (complex routing, workflows)

3.3. RPC framework

Some middleware solutions, like web services, use two types of communication. They are asynchronous messaging and remote procedure call (RPC). RPC works like a normal function call but happens over a network. It lets one service call another and wait for a response. Tools such as gRPC and Thrift make this process easier. They hide the network details and let services talk to each other like they are local. These tools also handle turning data into the right format, support many programming languages, and use fast, compact ways to send data. Remote Procedure Call (RPC) frameworks (as illustrated in Figure 3) enable synchronous inter-service communication by abstracting network calls. A comparison of RPC frameworks is listed in Table 3

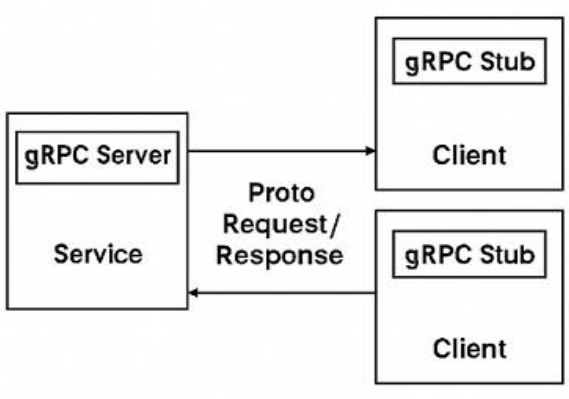

Figure 3. Remote procedure call

Table 3. Comparison of various RPC frameworks.

Protocol	Scalability	Performance	Security	Use Cases
gRPC is an open-source RPC framework developed by Google for communication between distributed systems	High speed, Low latency	High (Language-agnostic, distributed)	Can be implemented (TLS)	Microservices communication, Cloud-native deployments, Mobile backbends
Thrift is a software framework that enables communication between services in different languages	High performance, Cross-language compatibility	High (Language-neutral)	Can be implemented (SASL)	Service development across different programming languages, Enterprise applications
Apache Avro is a data serialization framework designed for efficient data exchange in distributed systems	High performance, Efficient data encoding	High (Schema-based)	Can be implemented (integrated with security frameworks)	Data serialization and exchange, High-throughput data pipelines, Polyglot data processing
Protocol Buffers (Protobuf) is a compact, efficient data format for fast transmission	High performance, Compact data representation	High (Language-neutral)	Can be implemented (TLS)	Efficient data serialization, Mobile app development, API development
Apache Dubbo, developed by Alibaba and now part of the Apache Software Foundation, is a high-performance, open-source RPC framework for distributed applications.	High performance (tuned for Java)	High (Service discovery, dynamic routing)	Can be implemented (integrated with security frameworks)	Microservices development (Java-centric), Distributed service invocation

These protocols and technologies are essential for interservice communication in microservices. They enabled efficient, reliable, and secure interactions across distributed systems. Right use ensures the scalability, flexibility, and resilience needed for modern cloud-native applications.

4. LITERATURE REVIEW

Research into inter-service communication (ISC) spans various domains, from specific industry applications to broader architectural challenges. In the context of critical infrastructure, Furda et al. [12] demonstrate the application of microservices to modernize rail operations, proposing an integration layer to overcome data silos and system rigidity. Their work acknowledges foundational concepts while addressing domain-specific challenges like orchestration and security. A significant portion of the literature focuses directly on optimizing ISC performance. The series of studies by Weerasinghe and Perera[6][15] provides a foundational evaluation

of communication methods and proposes optimized strategies to reduce latency and improve scalability. This theme is extended by Starck and Ghofrani [13], who target performance bottlenecks in gRPC, proposing a hybrid load-balancing technique for long-lived streams to improve distribution efficiency. Further optimization strategies are explored by Saxena et al. [19], who analyze the impact of low-level parameters like message size and network configuration on responsiveness and scalability. The complexity of microservice systems has also spurred research into architectural recovery and design guidance. Aksakalli et al. [14] offer a systematic review of deployment and communication patterns, linking them to performance and maintainability outcomes. Similarly, Wang et al. [18] describe a method for reconstructing microservice architectures by analyzing code, traffic, and runtime features to identify service boundaries and dependencies. This effort to provide practical plans is complemented by Weerasinghe and Perera's later work [17]. They proposes a reference architecture focused on scalability and resilience through optimized service discovery and load balancing. The challenge of migration is addressed by Kazanavičius and Mažeika [20], who evaluate communication patterns during the decomposition of monoliths, measuring key metrics like latency and throughput to guide a successful transition. As systems grow and interact, security and context-specific evaluations become paramount. Li, Chen, and Lin [16] tackle the security management overhead by proposing a framework for automated inter-service authorization, aiming to reduce manual effort and associated risks. The critical issue of securing communication across heterogeneous environments is explored in depth by Butt [21], whose research on inter-mesh communication emphasizes standards-based mutual authentication and encryption. Finally, the practical implications of ISC choices are illustrated by Gördesli and Varol [22], who use e-commerce case studies to compare the performance and reliability of synchronous HTTP/REST against asynchronous messaging, underscoring the direct link between communication patterns and business-level outcomes like user experience.

5. DIFFERENT COMMUNICATION TECHNIQUE

Synchronous and Asynchronous Communication: When choosing an IPC mechanism for a service, it's helpful first to consider how the services interact with each other. There are various styles of client-service interactions, which can be categorized along two dimensions. These are one-to-one and one-to-many [23]. In one-to-one communication between single to single service. actually In one-to-many communication among one to many service. Another one is synchronous or asynchronous communication. In synchronous, the client waits for a quick reply and may block until it comes. In asynchronous, the client does not wait. The reply may come later[23]. The table below shows how these styles connect in IPC. It categorizes IPC scenarios, clarifying how these dimensions shape communication between processes[23]. These communication styles can be combined, leading to distinct interaction patterns as summarized in Table 4.

Table 4. Communication Type and Relation with one-to-one/one-to-many interaction.

Communication Type	One-to-One	One-to-Many
Synchronous communication is a form of interaction where all participants engage simultaneously, enabling real-time exchanges of messages and immediate feedback.	Request/response	Synchronous communication requires a dedicated, immediate response, which is incompatible with a one-to-many broadcast.
Asynchronous communication is a form of interaction that allows participants to exchange messages and information at different times, without the need for simultaneous engagement.	Notification	Publish/subscribe
	Request/async response	Publish/async responses

Synchronous communication gives real-time interaction [24]. It works with quick feedback and faster decisions. It needs all participants to be available at the same time. This is hard for teams working in different time zones. Synchronous communication lets people reply when they can[24]. It allows more thoughtful answers. But replies can be delayed, and missing real-time context may cause miscommunication. The comparison between the two is shown in the table below[25]. A comparison of synchronous and asynchronous communication methods, based on parameters like latency, throughput, coupling and complexity, is presented in Table 5.Table 5. Comparison of Synchronous and Asynchronous

Parameter	Synchronous	Asynchronous
Latency	Typically lower latency since the caller waits for a response immediately	Higher latency because the caller does not wait and the response may come later.
Throughput	Usually lower throughput as each call blocks until completion	Higher throughput as multiple requests can be handled concurrently without waiting
Coupling	Tighter coupling; services depend on each other being available at the same time	Looser coupling; services can operate independently and handle messages when ready
Complexity	Simpler to implement and understand since it follows a straightforward request-response pattern.	More complex due to the need for handling message queues, callbacks, or event-driven logic.

5.1 Messaging Protocol

Message-oriented middleware (MOM) allows to connect different applications. It is important in event-driven and distributed systems. It often uses publish/subscribe communication. It enables cross-platform communication, crucial for coordinating business processes in e-business. MOM systems are categorized into Point-to-Point and Publish/Subscribe models. This communication type includes two methods: point-to-point and publish/subscribe. The table below compares these techniques. There are two ways we can communicate in this type they are point-to-point communication and publish/subscribe communication. The below Table 6 compares this technique

Table 6. Comparison of Point-to-Point and Publish/Subscribe Communication Type

Feature	Point-to-Point	Publish/Subscribe
Message Target	Single receiver per message	Multiple subscribers per message
Communication Model	The sender and receiver are directly linked	Publisher sends messages to multiple subscribers
Use Case	Task distribution, load balancing	Broadcasting events, real-time updates
Reliability	Typically ensures message delivery to one receiver	Ensures delivery to all active subscribers at the time
Queue System	Uses a queue; message consumed by one receiver	Uses a topic; all subscribers receive the message
Example	Job dispatching in a processing system	Stock market updates to multiple clients

5.2 RPC

The concept of remote procedure calls (RPC) is straightforward. It builds on the understanding that procedure calls are a familiar and effective method for transferring control and data within a single computer program. Thus, the idea is to extend this mechanism to enable the transfer of control and data over a communication network. When a remote procedure is called, the calling environment is suspended, the parameters are sent across the network to the environment where the procedure will execute (referred to as the callee), and the desired procedure runs there [26]. Here is a table comparing Simple Control Transfer, Simple Data Transfer, Simple Stubs, and Design for Concurrency in RPC communication techniques.

Table 7. Comparison of various RPC communication techniques

Technique	Description	Purpose	Pros	Cons
Simple Control Transfer.	Allows control to be passed from a client to a server without significant data exchange.	Basic client-server interaction.	Minimal overhead, fast response.	Limited to control signals only.
Simple Data Transfer	Enables small amounts of data to be sent with control commands	Data exchange for lightweight tasks.	Quick data transfer.	Limited by small data capacity.
Simple Stubs	Uses auto generated code to simplify client-server communication in RPC calls.	Simplifies RPC implementation.	Reduces complexity for developers.	Limited flexibility for custom calls.
Design for Concurrency	Optimizes RPCs to handle concurrent client requests effectively.	Improves performance under load	Efficient handling of multiple requests	Higher complexity in implementation.

5.3 Message Broker and Queuing System

A message broker is a software tool that helps microservices communicate. It handles data, commands, and events between services. It improves real-time processing. It also separates services to make systems more reliable and scalable. Distributed message brokers use a publish-subscribe model for asynchronous communication. They are common in IoT and event-driven systems. Popular examples include Apache Kafka, AMQP, and ActiveMQ.[27]. A message queue helps services talk to each other. It works in serverless and microservices systems. It stores messages until one consumer takes them. It makes tasks independent from each other. It also holds workloads for a while. This helps handle sudden traffic spikes smoothly. Different terminals can publish or consume messages, with multiple brokers collaborating in distributed systems. The publisher is the producer, while the message retriever is the consumer. Messages are organized into queues or topics, with producers specifying the topic when sending [28]. Table 8 compares the message broker and message queuing system.

Table 8. Comparison of Message Broker and Message Queuing Systems.

Feature	Message Broker	Queuing System
Definition	An intermediary that routes messages between services using various protocols	A system that stores messages in a queue for asynchronous processing.
Protocol Support	Supports multiple protocols (e.g., MQTT, AMQP, HTTP).	Typically follows a standard protocol (e.g., AMQP).
Routing	Can intelligently route messages based on topics or rules.	Primarily follows FIFO (First-In-First-Out) processing.
Decoupling	Decouples producers and consumers, allowing independent evolution.	Also decouples, but focuses on queueing mechanics.
Message Transformation	Can transform messages as they are routed.	Generally, does not transform messages; focuses on delivery.
Load Balancing	Can support complex load balancing across multiple consumers.	Distributes messages among consumers, balancing the load.
Scalability	Highly scalable with support for many connections and messages.	Scalable but often limited to simple queue mechanics.
Complex Communication Patterns	Supports publish/subscribe, request/reply, etc.	Point-to-point communication.
Message Durability	Often includes durability and persistence features.	Often includes persistence to disk to prevent message loss.
Use Cases	Event-driven architectures, microservices, IoT systems.	Background job processing, asynchronous task execution.

6. EVALUATION AND COMPARISON

Choosing a communication method depends on many things. Performance and scalability are important. Reliability and fault tolerance also matter. Security needs must be checked. Developer experience and method complexity play a role too. The use case and system goals guide the choice. Each method has pros and cons. Synchronous is simple but slow and less scalable. Asynchronous is fast and scalable but harder to manage. Service Mesh adds reliability with load balancing but may cause delay. Publish/Subscribe works well in event-driven systems but makes security harder. RPC is fast with low delay but needs careful fault handling. WebSocket gives real-time updates but struggles to scale. The best option depends on the system's needs and setup. Table 9 compares various communication approaches against key performance, scalability, reliability, security, and developer experience factors.

Table 9. Comparison of communication type concerning key factors to choose the best one.

Approach	Performance	Scalability	Reliability	Security	Developer Experience
Synchronous	Generally higher due to immediate response.	Limited scalability.	A single point of failure can impact the entire system.	Can be more secure as data is transmitted directly and can be encrypted.	Can be simpler to implement due to direct interaction.
Asynchronous	Can handle high throughput.	Highly scalable.	More resilient to failures due to decoupling and message queues.	Queued messages demand extra access controls and encryption	Can be more complex due to asynchronous programming models and message handling.
Service Mesh	Service mesh overhead is often negligible and justified by its operational benefits in complex systems.	Highly scalable	Provides built-in features for fault tolerance, retries, and circuit breaking.	Can provide strong security features like encryption, authentication, and authorization.	Simplifies distributed system development by abstracting away communication complexities.
Publish/ Subscribe	Can be efficient for many-to-many communication patterns	Highly scalable	Can be resilient to failures if message queues are used.	Requires careful consideration to ensure message confidentiality and integrity.	Can be more complex to implement due to asynchronous programming models and message handling.
Remote Procedure Calls	Can be efficient for simple, synchronous interactions.	Synchronous RPC promotes tighter coupling, potentially limiting scalability.	Can be less reliable due to direct dependencies.	Can be secured using encryption and authentication.	Can be simpler to use for synchronous communication patterns.
WebSocket	Can provide low-latency, bidirectional communication	Less scalable due to maintaining persistent connections.	Can be less reliable due to persistent connections.	Can be secured using encryption and authentication	Can be more complex to implement due to the persistent nature of the connection.

7. OPEN ISSUE AND FUTURE DIRECTION

Research on interservice communication (ISC) in microservices focus on areas like speed, reliability, network delays, and security. However, several studies show that important issues like developer experience, communication management overhead, scalability, and human factors still require more attention. While studies like [30, 31] provide valuable evaluations of specific communication methods, their scope highlights a broader research gap in understanding the impact of developer experience and human factors on ISC management in large-scale, real-world deployments. The paper [31] by Li, Chen, and Lin contributes through automated authorization, yet concerns remain regarding scale, metrics, bias, and the importance of human checks. The study [32] by Ramu looks at performance but only on a few metrics. It does not cover wider context or human factors. All these works add useful knowledge. But they have limits like narrow focus, missing context, and little qualitative study. More research is needed to fill these gaps and study new trends in ISC.

7.1. Security and Automated ManagementIn microservices, services must talk to each other safely. Strong security is needed to stop attacks and data leaks. Important steps are checking identity with authentication, giving permission with authorization, and keeping data safe with encryption. Securing API gateways and using service mesh security are also important. API keys must be managed well. Input should be checked, and logs should be monitored. Service discovery must be secured, and vulnerabilities managed. These practices reduce risks and keep microservices safe and reliable.

7.2. Scalability and Performance Optimization

In microservices, services must communicate efficiently. As more services are added, managing them gets harder. Load balancing helps spread traffic evenly. Service discovery helps services find each other. Message queues ensure messages are sent smoothly. Caching and API gateways improve speed and security. Using fast protocols like gRPC reduces delay. Service meshes give better control over traffic. Monitoring and logging track how the system performs. They help make the system more scalable and reliable.

7.3 Developer Experience and Human Factors

Managing communication between microservices is tricky and can stress developers. Moving from a monolithic system to distributed services is not easy. Developers need to learn new ideas like eventual consistency, circuit breakers, and distributed tracing. This adds mental load and can slow down work. Debugging is especially tough. Without good tools, finding the cause of a failure across multiple services takes a lot of time and effort. The way services communicate synchronous or asynchronous also affects how teams coordinate and share responsibility. New technologies are improving inter-service communication in microservices. gRPC and GraphQL protocols help with faster data transfer and also decrease delays. Service meshes like Istio and Linkerd benefit routing, load balancing, and monitoring. Tools like Kafka and RabbitMQ support communication among services. This helps the system scale and stay strong. These tools help microservices connect smoothly and work reliably. To make things better, future work should focus on helping developers. This means simpler tools, better diagnostics, and clear learning resources so developers can confidently build and manage complex microservices.

8. RECOMMENDATION

Choosing the right interservice communication (ISC) strategy is important in microservices. It helps improve performance. The following recommendations are provided for different scenarios:

Real-time Applications : Synchronous methods like gRPC. Example chat, gaming

Event-Driven Architectures: Asynchronous Messaging Technique with tools like Apache Kafka or RabbitMQ. Example IoT, notifications

Batch Processing : Publish/Subscribe model with Message brokers. Example ETL, analytics

Complex Interactions : Hybrid approach. Synchronous calls for critical tasks like payments. Asynchronous messaging for less urgent tasks like order confirmations. Example e-commerce

High Scalability : Service mesh like Istio. Example cloud-native apps

Security-Sensitive Environments : RPC and messaging protocols with API gateways for extra security. Example finance

Legacy Integration: RESTful APIs with HTTP.

Choose a communication strategy that fits your app's needs. Consider traffic, scalability, and security. This helps make interactions efficient and reliable.

IX. CONCLUSION

Choosing the right interservice communication (ISC) approach is very important for microservices. It helps services share data smoothly, which improves speed and performance. A good strategy makes the system scalable and prevents bottlenecks. It also adds resilience so services can work and recover on their own. Strong communication methods keep data safe during transfer. In the end, the right ISC choice creates a flexible system. It can produce with business needs and manage more traffic. This creates the system stable, dependable, and easier to manage.

REFERENCES

[1] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, Microservice Architecture: Aligning Principles, Practices, and Culture. O'Reilly Media, 2016.

[2] L. De Lauretis, "From monolithic architecture to microservices architecture," in Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW), 2019, pp. 93-96.

[3] K. Gos and W. Zabierowski, "The comparison of microservice and monolithic architecture," in Proc. IEEE XVIth Int. Conf. Perspective Technol. Methods MEMS Design (MEMSTECH), 2020, pp. 150-153.

[4] S. Baškarada, V. Nguyen, and A. Koronios, "Architecting microservices: Practical opportunities and challenges," J. Comput. Inf. Syst., 2020.

[5] M. Sowmya et al., "API traffic anomaly detection in microservice architecture," in Proc. IEEE/ACM Int. Symp. Cluster, Cloud Internet Comput. Workshops (CCGridW), 2023, pp. 206-213.

[6] L. D. S. B. Weerasinghe and I. Perera, "Evaluating the inter-service communication on microservice architecture," in Proc. 7th Int. Conf. Inf. Technol. Res. (ICITR), 2022, pp. 1-6.

[7] L. Zou, Z. Wang, J. Hu, Y. Liu, and X. Liu, "Communication-protocol-based analysis and synthesis of networked systems: Progress, prospects and challenges," Int. J. Syst. Sci., vol. 52, no. 14, pp. 3013-3034, 2021.

[8] S. N. Raje, "Performance comparison of message queue methods," M.S. thesis, Univ. Nevada, Las Vegas, NV, USA, 2019.

[9] D. A. Menasce, "MOM vs. RPC: Communication models for distributed applications," IEEE Internet Comput., vol. 9, no. 2, pp. 90-93, 2005.

[10] Z. Houmani, D. Balouek-Thomert, E. Caron, and M. Parashar, "Enhancing microservices architectures using data-driven service discovery and QoS guarantees," in Proc. IEEE/ACM Int. Symp. Cluster, Cloud Internet Comput. (CCGRID), 2020, pp. 290-299.

[11] R. Chandramouli, Microservices-Based Application Systems (NIST Special Publication 800-204). Nat. Inst. Standards Technol., 2019.

[12] A. Furda, L. van den Berg, G. Reid, G. Camera, and M. Pinasco, "Developing a microservices integration layer for next-generation rail operations centers," IEEE Softw., vol. 39, no. 5, pp. 9-16, 2022.

[13] C. Starck and J. Ghofrani, "Improving load balancing of long-lived streaming RPCs for gRPC-enabled inter-service communication," in Proc. 14th Central Eur. Workshop Services Composition (ZEUS 2023), 2023, p. 45.

[14] I. K. Aksakalli, T. Çelik, A. B. Can, and B. Tekinerdoğan, "Deployment and communication patterns in microservice architectures: A systematic literature review," J. Syst. Softw., vol. 180, 111014, 2021.

[15] S. Weerasinghe and I. Perera, "Optimized strategy for inter-service communication in microservices," Int. J. Adv. Comput. Sci. Appl., vol. 14, no. 2, 2023.

[16] X. Li, Y. Chen, and Z. Lin, "Towards automated inter-service authorization for microservice applications," in Proc. ACM SIGCOMM Conf. Posters Demos, 2019, pp. 3-5.

[17] L. D. S. B. Weerasinghe and I. Perera, "Reference architecture for microservices with an optimized

inter-service communication strategy," in Proc. Int. Res. Conf. Smart Comput. Syst. Eng. (SCSE), vol. 7, 2024, pp. 1-6.

[18] L. Wang et al., "Microservice architecture recovery based on intra-service and inter-service features," J. Syst. Softw., vol. 204, 111754, 2023.

[19] D. Saxena, W. Zhang, M. Tummala, S. Goel, and A. Akella, "Towards efficient microservice communication," in Proc. 5th Workshop Adv. Tools, Program. Lang., Platforms Implementing Eval. Algorithms Distrib. Syst., 2023, pp. 1-5.

[20] J. Kazanavičius and D. Mažeika, "Evaluation of microservice communication while decomposing monoliths," Comput. Inform., vol. 42, no. 1, pp. 1-36, 2023.

[21] Z. Butt, "Secure microservice communication between heterogeneous service meshes," M.S. thesis, Univ. Oslo, Oslo, Norway, 2022.

[22] M. Gördesli and A. Varol, "Comparing interservice communications of microservices for E-commerce industry," in Proc. 10th Int. Symp. Digit. Forensics Security (ISDFS), 2022, pp. 1-4.

[23] C. Richardson and F. Smith, "Microservices: From design to deployment," Nginx Inc., vol. 1, pp. 24-31, 2016.

[24] Skedda, "Hybrid communication is broken: The pros and cons of synchronous and async," Skedda Blog, 2024. [Online]. Available: https://www.skedda.com/blog/hybrid-communication-is-broken-the-pros-and-cons-of-synchronous-and-async

[25] IEEE Computer Society, "Synchronous and asynchronous data transmission," 2023. [Online]. Available: https://www.computer.org/publications/tech-news/trends/synchronous-asynchronous-data-transmission

[26] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," ACM Trans. Comput. Syst., vol. 2, no. 1, pp. 39-59, 1984.

[27] V. John and X. Liu, "A survey of distributed message broker queues," arXiv Preprint, arXiv:1704.00411, 2017.

[28] G. Fu, Y. Zhang, and G. Yu, "A fair comparison of message queuing systems," IEEE Access, vol. 9, pp. 421-432, 2020.

[29] L. D. S. B. Weerasinghe and I. Perera, "Evaluating the inter-service communication on microservice architecture," in Proc. 7th Int. Conf. Inf. Technol. Res. (ICITR), 2022, pp. 1-6.

[30] S. Weerasinghe and I. Perera, "Optimized strategy for inter-service communication in microservices," Int. J. Adv. Comput. Sci. Appl., vol. 14, no. 2, 2023.

[31] X. Li, Y. Chen, and Z. Lin, "Towards automated inter-service authorization for microservice applications," in Proc. ACM SIGCOMM Conf. Posters Demos, 2019, pp. 3-5.

[32] V. B. Ramu, "Performance impact of microservices architecture," Rev. Contemp. Sci. Acad. Stud., vol. 3, no. 6, pp. 1-12, 2023.