КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 3 (23), 2024
ISSN 2663 - 4023

DOI 10.28925/2663-4023.2024.23.155170
UDC 004.72:004.75

Maksym Kotov
Master's degree in computer and information systems security, student at the department of cyber security and information protection
Taras Shevchenko National University of Kyiv,
Department of Cybersecurity and Information Protection, Kyiv, Ukraine
ORCID 0000-0003-1153-3198
maksym_kotov@ukr.net

Serhii Toliupa
Doct. Sci. (Engineering), Professor, professor of the department of cyber security and information protection
Taras Shevchenko National University of Kyiv, Department of Cybersecurity and Information Protection, Kyiv, Ukraine
ORCID 0000-0002-1919-9174
tolupa@i.ua

Volodymyr Nakonechnyi
Doct. Sci. (Engineering), Professor, professor of the department of cyber security and information protection
Taras Shevchenko National University of Kyiv, Department of Cybersecurity and Information Protection, Kyiv, Ukraine
ORCID 0000-0002-0247-5400
nvc2006@i.ua

REPLICA STATE DISCOVERY PROTOCOL BASED ON ADVANCED MESSAGE QUEUING PROTOCOL

Abstract. When it comes to the ever-changing landscape of distributed computing, having a solid understanding of how to maintain state information that is synchronized and consistent among replicas is extraordinarily critical. Within the scope of this investigation, the Replica State Discovery Protocol, which is a component of the Advanced Message Queuing Protocol (AMQP), is developed and examined in detail. The purpose of this investigation is to show how this protocol contributes to the maintenance of consistent state information across many replicas in distributed systems. We will start with the fundamentals of AMQP and the reasons why it is so important in the distributed systems of today. This lays the groundwork for our more in-depth exploration of the Replica State Discovery Protocol by providing the foundation. While going through each level of the protocol, we will pay special attention to the way messages are passed back and forth during the phases as well as the general handling of data. An important aspect examined in this study pertains to the difficulties associated with the development of said protocol. Mitigating challenges such as race conditions and executing seamless transitions between segments are not simple tasks. However, in this discussion, we shall examine several viable approaches and resolutions that illuminate the practical and theoretical dimensions of replica state management. This article is intended for individuals who are enthusiastic about or are already engaged in distributed computing. In addition to being educational, this work endeavors to inspire additional investigation and scrutiny concerning AMQP and state management in distributed systems.

Keywords: distributed computing; state synchronization; Replica State Discovery Protocol (RSDP); Advanced Message Queuing Protocol (AMQP); consistent state management; race conditions; replica management.

INTRODUCTION

AMQP is an open and standardized protocol designed for message-oriented middleware. Its goal is to ensure reliable and efficient message sharing across diverse and faraway networks. This is important for making conversation work in complicated system designs [1] – [3].

The three things that AMQP is based on are resilience, security, and interoperability [4]. It can handle a lot of different types of communication, from simple point-to-point lists to advanced topic-based publish-subscribe models [5]. Because it can do so many things, AMQP is the best choice for systems that need to communicate in a way that can grow, send messages reliably, and keep transactional integrity [5].

The capacity of AMQP to build a unified messaging system that goes across platform and language limitations is one of its notable features. In contrast to middleware systems that depend on implementations or proprietary standards, this one achieves universality through a protocol-centric approach. Through its role as a universal language for various systems, AMQP guarantees consistent message exchanges, regardless of the technology used [7].

Even more so when considering distributed systems, AMQP's importance becomes apparent. It solves problems like message sequencing, fault tolerance, and network delay that arise with distributed computing [8], [9]. To guarantee that messages are sent and received efficiently while keeping the system's integrity and coherence intact, AMQP offers a dependable and standardized messaging architecture.

Problem formulation. Keeping track of the status of each node (also known as a "replica") and ensuring that they all have the same information is of the utmost importance in the realm of distributed systems, which are essentially networks of nodes that are working together [10] – [17]. These systems consist of a few components that can function across a variety of network connection points. It is like having a team that is dispersed over multiple places, and for everything to run smoothly, every member of the team needs to have access to the same information that is up to date [10] – [15].

The difficulty arises when we attempt to maintain synchronization among all these components, particularly when we are confronted with disturbances in the network or when certain components of the system are momentarily disconnected [10] – [14]. What would it be like to try to keep a group of friends informed about a plan when the signal quality on each of their phones is different? It is possible that some messages will come late or not at all.

The ability of a distributed system to withstand problems, such as when a component fails or ceases to be accessible, is strongly dependent on the synchronization that occurs between its components [11] – [14]. It is possible that due to a component of the system not having access to the most recent information, the computations or conclusions that are drawn are inaccurate. In the case that one component of the system suffers issues, the synchronization method acts as a safety net to ensure that the remaining components of the system can continue to function correctly with the appropriate information that is communicated [13] – [15].

The correct integration of new or updated components is another essential aspect, as is the determination of the current condition of each component [16]. The act of simply sharing data throughout this procedure is not as crucial as making certain that every component is aware of the others and the condition in which they are currently located. Additionally, this makes it easier for the system to adjust to changes, such as the allocation of employment or the processing of new labor [17].

The purpose of the article. To summarize, the aim of this research is to develop a distributed system synchronization protocol that is durable, adaptive, and able to keep a record of each component and make certain that they are in sync with one another.

UNDERSTANDING REPLICA STATE DISCOVERY PROTOCOL

Definition and role of AMQP in Replica State Discovery Protocol. The Replica State Discovery Protocol is an enhancement to the Advanced Message Queuing Protocol (AMQP), which was developed specifically for the difficult issue of managing states in distributed systems [3] – [7]. Its purpose is to ensure that every component of a network, referred to as a "replica", is aware of the others and is in sync with them. This protocol makes full use of the powerful messaging capabilities of AMQP to guarantee that every node in a distributed system can locate and align its state with the states of the other nodes while maintaining efficiency.

The following figure shows a sequential diagram of AMQP interactions:

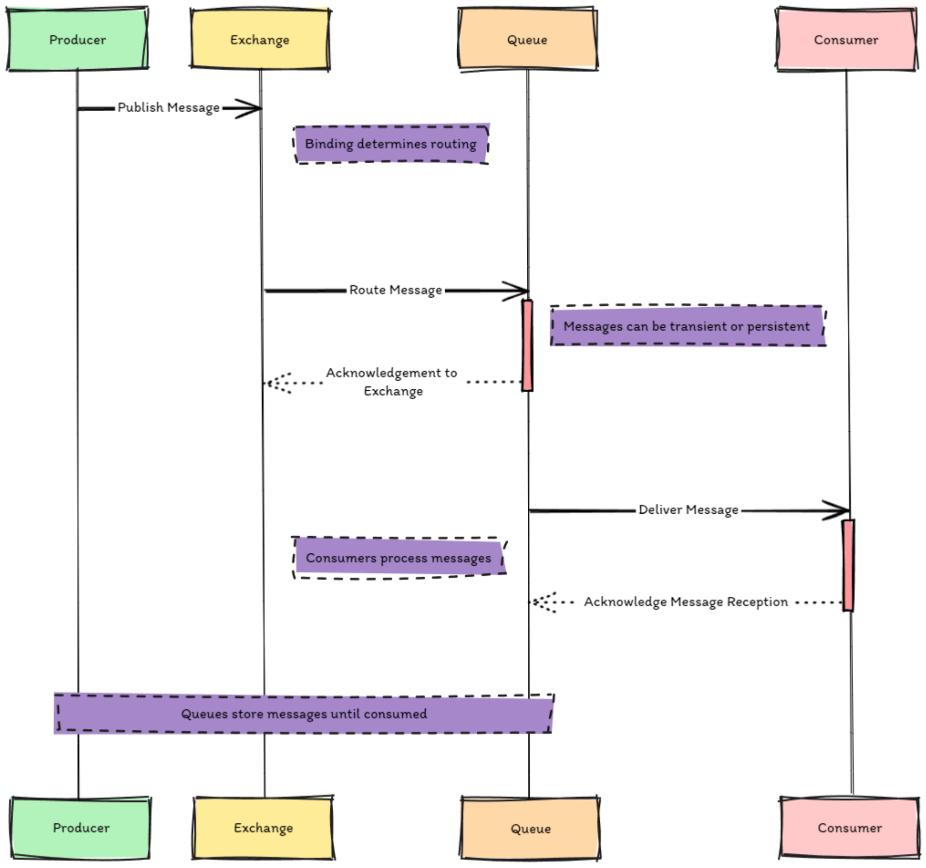
Producer
Exchange
Queue
Consumer
Publish Message
Binding determines routing
Route Message
Messages can be transient or persistent
Acknowledgement to Exchange
Deliver Message
Consumers process messages
Acknowledge Message Reception
Queues store messages until consumed
Producer
Exchange
Queue
Consumer

Fig. 1. AMQP interactions

The operation is as follows: Imagine a group of people working on different aspects of a project from where they are physically located. Using the Replica State Discovery Protocol, every member of the team (or replica) can not only learn about the new or existing members of the team, but also ensure that everyone is on the same page. In circumstances in which having information that is both up to date and consistent throughout all components of the system is essential to the system's reliability and efficacy, this is of the utmost importance.

The part that this protocol plays in systems that use AMQP is quite important. First, it makes these networks more reliable. It is important for each part of the network to know about the other parts and their state, just like it is important for a team to know who is working on what. It is important to have this kind of knowledge to keep the system's consistency and integrity, especially when the network is down or having issues.

The second point is that the protocol is a big part of how the system handles its tasks and how much it can cope with. According to the needs, the system can decide how to distribute resources and when to scale up or down by keeping an eye on the state of each copy. Making this change not only improves speed but also makes sure that resources are used in the best way possible.

Overview of the protocol's phases: DEBATES, SHARE, and CLOSE. There are clear steps that the Replica State Discovery Protocol follows: *"DEBATES"*, *"SHARE"*, and *"CLOSE"*. Each of these steps is a structured way to manage the state of a distributed environment, and they all work together to make sure the system is consistent and reliable.

The following figure shows a sequential diagram of RSDP phases:

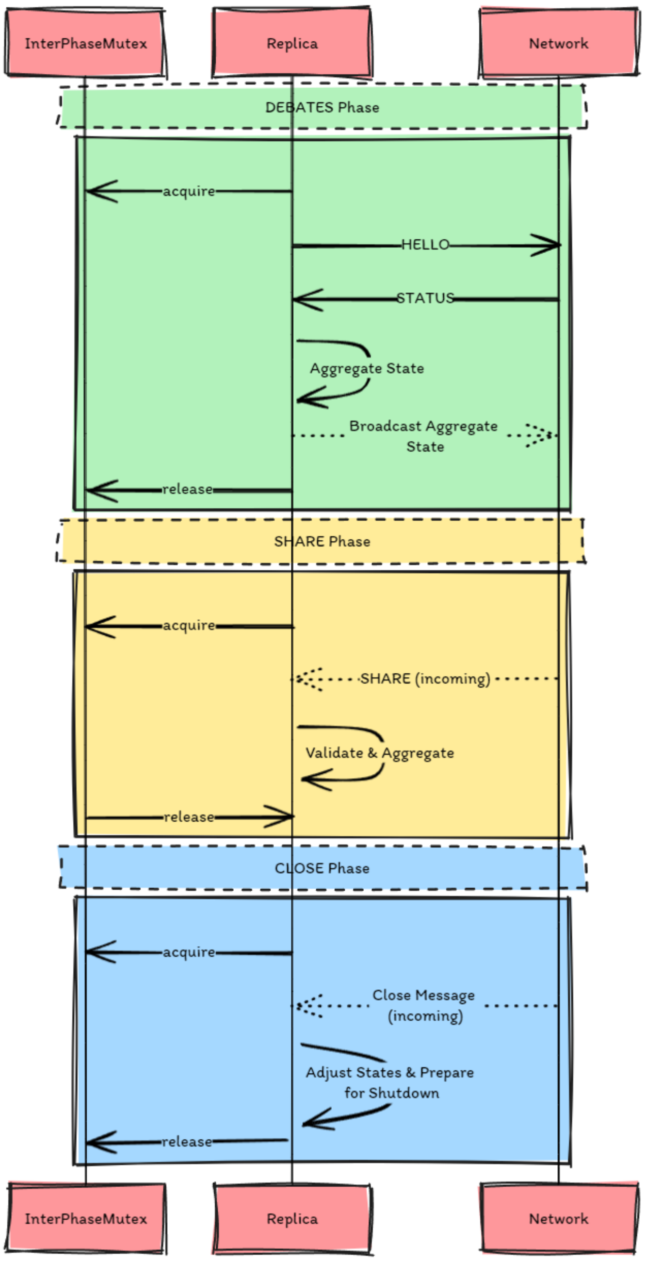

Fig. 2. RSDP Phases

The protocol starts the process of gathering states in the “DEBATES” step. At this stage, replicas send *“HELLO”* and *“STATUS”* messages to each other, which works as a preparation for sharing states. The *“HELLO”* messages are a way for replicas to introduce themselves to the network. After this, the *“STATUS”* messages are being sent in response to share data about what’s going on with each replica right now.

Right after the *“DEBATES”* phase is over, the protocol goes on to the *“SHARE”* phase. Now that the states have been received, the focus shifts to putting together and organizing the information about the states that was gathered in the *“DEBATES”* phase. An aggregated state will be shared with all the replicas during this stage. Every replica will broadcast its aggregation into the network thus achieving consensus after validation. This makes sure that every replica has the same copy of the shared state.

At some point when replica decides that its job is done, it could move on to the *“CLOSE”* step, which deals with the idea that replicas might be shut down or removed from the system when they are no longer needed. During this phase, a replica that is about to shut down will send a message to the other members. This message tells the replication members that are still alive that they need to change their states and get ready for the replica to leave. The *“CLOSE”* phase makes sure that the system can dynamically adapt to changes in its components, which preserves its working continuity and resilience.

The Replica State Discovery Protocol keeps a careful balance between speed and thoroughness during these stages to make sure that state synchronization is both complete and on time. Throughout the process, this balance is kept. The protocol has a methodical way of handling these steps, which makes it less likely that race conditions and errors will happen. This makes it an important tool in the field of distributed computing.

Overview of the protocol’s state management principles. Several key ideas form the basis of the Replica State Discovery Protocol, which was developed with the purpose of efficiently managing states in distributed systems while also addressing the inherent complexity and dynamic nature of these systems.

One of the most important aspects of the protocol is that it encourages state abstraction and encapsulation. In other words, the mechanism for preserving the state, which includes retrieving, updating, and synchronizing it, is housed within a separate module according to this declaration. By separating the components, the system becomes more organized, more manageable, and better equipped to respond to changes.

Implementing dependency injection for state management is another key factor that must be taken into consideration. Rather than being hardcoded into the system, the state management module is offered as a component that can be interchanged with other modules. The versatility of the system is increased utilizing this method, which also makes adjustments and extensions easier to implement. The system’s adaptability and scalability are both increased because of this feature, which also makes it possible to incorporate plugins, each of which is accountable for a certain state component.

Across all replicas, the protocol places a high priority on synchronization and consistency maintenance. For ensuring that the system continues to be in a consistent state, the state management module gathers state information from several replicas and resolves any inconsistencies that the information may contain. Because of this, advanced algorithms and methods are required to manage conflicts and achieve convergence to a unified state, even when the network conditions are challenging.

TECHNICAL DEEP-DIVE INTO THE PROTOCOL PHASES

Detailed explanation of the "DEBATES" phase. It is the broadcasting of *"HELLO"* messages that marks the beginning of the *"DEBATES"* phase of the Replica State Discovery Protocol, which concludes with the *"broadcastShare"* method. Establishing initial communication across replicas and laying the groundwork for state synchronization are both essential tasks that must be completed during this phase.

The *"broadcastHello"* method, which is responsible for declaring the presence of a replica in the network, is initiated at the beginning of the phase:

```
async broadcastHello() {
    await this.interPhaseMutex.acquire();

    await this.broadcast({
        type: ReplicaStateDiscoveryProtocol.MESSAGE_TYPES.HELLO,
        from: this.getSenderCredentials(),
    });
}
```

In this case, the *"interPhaseMutex.acquire()"* method guarantees exclusive access to the state management phase, which helps to prevent race conditions from occurring while greeting messages are being broadcast for the first time. The critical portion of the protocol, which is where concurrent activities have the potential to change the state of the protocol, is protected by this mutex [18] – [21].

The following figure shows mutex operations sequential diagram:

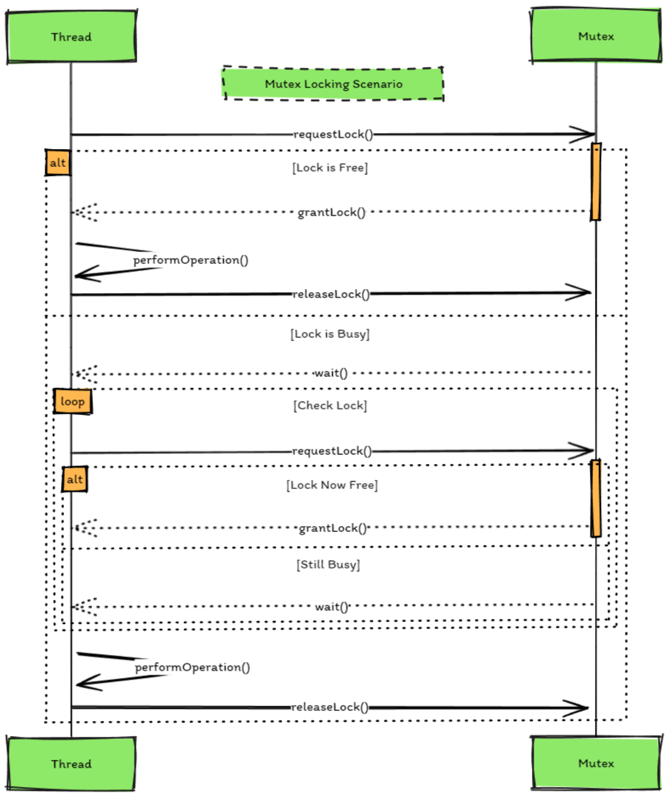

Fig. 3. Mutex Operations Sequence

Following the receipt of a *"HELLO"* message, a replica will send a *"STATUS"* message in response, which will indicate the present state of the replica. *"handleHelloMessage"* is the method that is responsible for managing this interaction:

```
async handleHelloMessage({ from }) {
    return this.amqpClient.publish(from.address, {
        type: ReplicaStateDiscoveryProtocol.MESSAGE_TYPES.STATUS,
        data: this.stateManager.getCurrentState(),
        from: this.getSenderCredentials(),
    });
}
```

The current state of the replica that is responding is included in the *"STATUS"* message that is published to the network by this approach. This is an essential component of state synchronization.

It is only when replicas start to receive *"STATUS"* signals that they begin to construct a perspective of the current state of the network. The following snippet shows implementation of this *"handleStatusMessage"* method:

```
async handleStatusMessage(message) {
    this.statusMessageBuffer.push(message);

    if (this.statusDebounceTimeout) {
        clearTimeout(this.statusDebounceTimeout);
    }

    this.statusDebounceTimeout = setTimeout(
        this.handleStatusUpdate.bind(this),
        this.statusDebounceDelay,
    );
}
```

The incoming *"STATUS"* messages are buffered by this method, which also makes use of a debounce timeout *"statusDebounceTimeout"*. By ensuring that the replica handles these messages collectively after a predetermined amount of time has passed, the timeout makes it possible to have a state representation that is more accurate. The *"broadcastShare"* method is the final step in the *"DEBATES"* phase, and it is responsible for broadcasting the aggregated data to all replicas.

```
async handleStatusUpdate() {
    const replicaStatus =
this.stateManager.aggregateState(this.statusMessageBuffer);

    await this.broadcastShare(replicaStatus);
}

async broadcastShare(status) {
    await this.broadcast({
        type: ReplicaStateDiscoveryProtocol.MESSAGE_TYPES.SHARE,
        data: status,
        from: this.getSenderCredentials(),
    });

    this.interPhaseMutex.release();
}
```

When the *"handleStatusUpdate"* function is called, it computes the combined state. This status is then sent to the network through the *"broadcastShare"* function. The most important thing about this process synchronization is the release of *"interPhaseMutex"* after transmission, which ends the protocol phase.

Beginning with the *"broadcastHello"* operation and concluding with the *"broadcastShare"* operation, the *"DEBATES"* phase consists of a series of controlled interactions that are repeated throughout the phase. It is essential to make careful use of the *"interPhaseMutex"* and to configure debouncing for the handling of status messages to ensure that the replicas can be brought into sync in a timely and accurate manner.

Exploration of the "SHARE" phase. The *"SHARE"* phase is a step that comes after the *"DEBATES"* phase. In the beginning of this phase, the replicas start the process of synchronizing the information that they have gathered about the state. In addition to this, they initiate the process of aggregating the data that they have stored from the previous phase.

The subsequent steps include broadcasting this information to all the replicas that are present on the network. This is accomplished with a broadcasting mechanism that is incorporated within the AMQP protocol.

At this stage, it is important to mention that the protocol is designed to deal with the possibility of racing conditions. It is common practice to make use of synchronization locks and mutexes to solve this problem [18] – [20]. The utilization of these synchronization techniques ensures that state changes are managed in a controlled and sequential manner, hence preventing any conflicts or data corruptions that may be generated by concurrent state alterations. This is accomplished by ensuring that the state changes are managed consecutively.

For instance, consider the following code snippet illustrating the synchronization process:

```
async handleShareMessage(message) {
    await this.interPhaseMutex.acquire();

    try {
        const replicaState =
this.stateManager.sanitizeShareMessage(message);

        if (this.stateManager.shouldReload(replicaState)) {
            this.stateManager.updateState(replicaState);

            await this.marketsManager.reloadActive(
                this.stateManager.normalizeState(replicaState),
            );
        }
    } finally {
        this.interPhaseMutex.release();
    }
}
```

In this code snippet, the *"handleShareMessage"* function operations are shown. The process begins with acquiring lock in the mutex called *"interPhaseMutex"*. This mutex ensures that no two *"SHARE"* messages are handled at the same time, thus avoiding race conditions. The *"sanitizeShareMessage"* method is responsible for filtering and validation of incoming state data. The *"shouldReload"* method contains logic to make a decision whether the state has changed and whether the internal state of replica should be updated. If this method decides that internal state should be updated — the data gets passed to the *"normalizeState"* method which is responsible for formatting in a form that could be used in this particular replica.

Description of the "CLOSE" phase. Each node in the network receives a *"CLOSE"* message from a replica that exits the network before the *"CLOSE"* phase begins. The network-wide shutdown notification procedure is initiated by this signal. This message's purpose is to inform the remainder of the system that the replica is going to depart. Because of this, the other replicas can adjust their parameters and reorganize their states according to the updated data. The remaining replicas will undergo a sequence of actions to adapt to the new network topology the moment they get a *"CLOSE"* message. They typically need to update their internal state to remove any references and dependencies to the replica that is exiting the network before they can accomplish this. This is also the time when the responsibilities of the node that is exiting the system may be transferred to another node. The system will continue to function normally if this is done.

The handling of a *"CLOSE"* message can be illustrated by the following code snippet:

```
async handleCloseMessage(message) {
    this.closeMessageBuffer.push(message);

    if (this.closeDebounceTimeout) {
        clearTimeout(this.closeDebounceTimeout);
    }

    this.closeDebounceTimeout = setTimeout(
        this.handlePolicyClose.bind(this),
        this.closeDebounceDelay,
    );
}

async handlePolicyClose() {
    await this.interPhaseMutex.acquire();

    try {
        const replicaStatus = this.stateManager.aggregateCloseState(
            this.closeMessageBuffer,
        );

        this.closeMessageBuffer = [];

        await this.marketsManager.reloadActive(
            this.stateManager.normalizeState(replicaStatus),
        );
    } finally {
        this.interPhaseMutex.release();
    }
}
```

In this implementation *"handleCloseMessage"* is responsible for collecting *"CLOSE"* messages that may happen at the same time. It uses debounce system to avoid premature status updates, thus optimizing resources utilization by avoiding unnecessary calls to the *"stateManager"* and stores incoming messages in the *"closeMessageBuffer"* buffer.

Once the configured debounce timeout is reached, through the event loop, the *"handlePolicyClose"* gets called. At this point the *"CLOSE"* state management phase begins and the *"interPhaseMutex"* gets acquired. Messages stored in a buffer will be aggregated through *"aggregateCloseState"* call on *"stateManager"*, thus providing a new stable state for replicas. After aggregation, the internal state gets updated and mutex will be released.

STATE MANAGEMENT AND DEPENDENCY INJECTION

Discussion on state management abstraction. Putting state management into its own class is a smart engineering choice that is needed to handle the complexity of distributed systems. The Replica State Discovery Protocol takes this method into account. There are several useful reasons for this abstraction, which are necessary to keep the system working correctly and quickly. To begin, flexibility is increased by separating the protocol's most important functions from its state management. The protocol can adapt to changes in how the state manages itself without affecting other parts of the process because it is built this way. It is possible to change how a state is managed without changing how the protocol works in its most basic form. This makes maintenance and updates much easier.

The following figure shows a *"ReplicaStateManager"* class diagram:

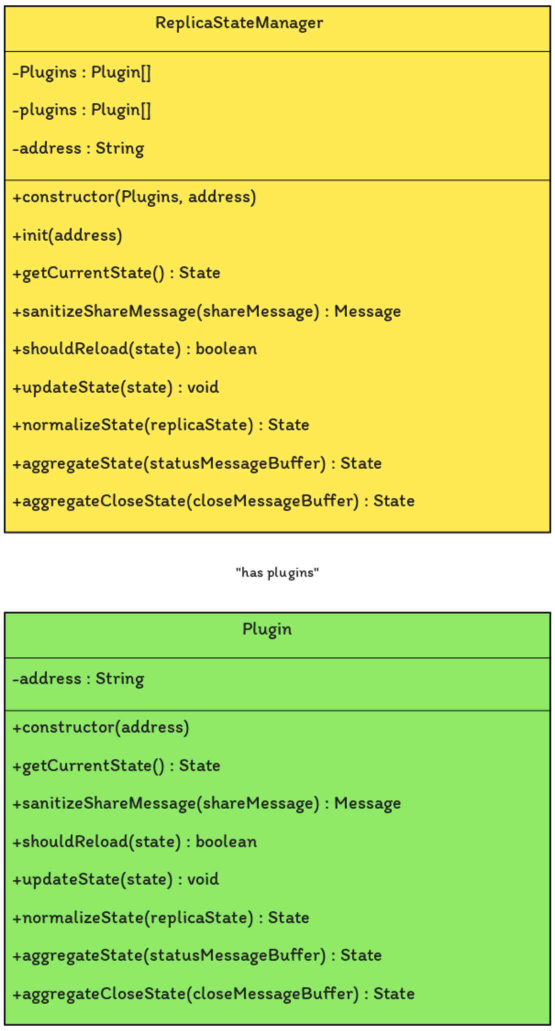
ReplicaStateManager
-Plugins : Plugin[]
-plugins : Plugin[]
-address : String
+constructor(Plugins, address)
+init(address)
+getCurrentState() : State
+sanitizeShareMessage(shareMessage) : Message
+shouldReload(state) : boolean
+updateState(state) : void
+normalizeState(replicaState) : State
+aggregateState(statusMessageBuffer) : State
+aggregateCloseState(closeMessageBuffer) : State
"has plugins"
Plugin
-address : String
+constructor(address)
+getCurrentState() : State
+sanitizeShareMessage(shareMessage) : Message
+shouldReload(state) : boolean
+updateState(state) : void
+normalizeState(replicaState) : State
+aggregateState(statusMessageBuffer) : State
+aggregateCloseState(closeMessageBuffer) : State

Fig. 4. ReplicaStateManager Class Diagram

With this division, it's possible to make things better. Before anything else, this abstraction makes it easy to scale. When it comes to distributed systems, it can get harder to keep track of state as the number of nodes or copies grows. Because this part is abstract, the protocol can more easily include a number of different state management methods that can work with a wide range of scales and levels of complexity. This makes it possible to use a plug-and-play approach, which means that different state management strategies can be used depending on what the system needs.

State management plugins. When it comes to distributed systems, the *"stateManager"* is an important part of handling the complicated parts of state management. For this component, which was meant to be both adaptable and expandable, plugins are the main way that it can be changed and integrated. These plugins are specialized tools that handle different aspects of state management in order to meet the needs of a distributed system. For those needs, they offer a way that can be changed to fit them.

That's why the *"stateManager"* has plugins built in to handle specific jobs that are linked to the state. It is the job of each plugin to take care of a certain part of the state. Let's say that one plugin is in charge of keeping replica members up to date and another plugin is in charge of managing process queues:

```
class ReplicaStateManager {
    constructor({ Plugins, address }) {
        this.Plugins = Plugins;
        this.plugins = [];

        this.address = address;

        this.init({ address });
    }

    init({address }) {
        this.plugins = this.Plugins.map((Plugin) => {
            return new Plugin({
                address,
            });
        });
    }

    getCurrentState() {
        return this.plugins.reduce((state, stateManager) => {
            const partialState = stateManager.getCurrentState();

            return {
                ...state,
                ...partialState,
            };
        }, {});
    }

    sanitizeShareMessage(shareMessage) {
        return this.plugins.reduce((message, stateManager) => {
            const partialMessage =
                stateManager.sanitizeShareMessage(shareMessage);

            return {
                ...message,
                ...partialMessage,
            };
        }, {});
    }

    shouldReload(state) {
        const reloadVotes = this.plugins.map((stateManager) => {
            return stateManager.shouldReload(state);
        });

        return reloadVotes.some((shouldReload) => shouldReload);
    }

    updateState(state) {
        this.plugins.forEach((stateManager) => {
            return stateManager.updateState(state);
        });
    }

    normalizeState(replicaState) {
        return this.plugins.reduce((state, stateManager) => {
            const partialState =
stateManager.normalizeState(replicaState);

            return {
                ...state,
                ...partialState,
            };
        }, {});
    }

    aggregateState(statusMessageBuffer) {
        return this.plugins.reduce((state, stateManager) => {
            const partialState =
                stateManager.aggregateState(statusMessageBuffer);

            return {
                ...state,
                ...partialState,
            };
        }, {});
    }

    aggregateCloseState(closeMessageBuffer) {
        return this.plugins.reduce((state, stateManager) => {
            const partialState =
                stateManager.aggregateCloseState(closeMessageBuffer);

            return {
                ...state,
                ...partialState,
            };
        }, {});
    }
}
```

CONCLUSIONS

In real-life situations, like high-availability database clusters or global content delivery networks, it is important to handle protocol stages in the right way to keep all nodes up to date. This has to be done correctly for tasks like load balancing, fault tolerance, and data replication between nodes that are in different places.

The rules and steps used in the Replica State Discovery Protocol have a big effect on how distributed systems are designed and how reliable they are. These systems are more reliable as a whole because of the protocol. It does this by handling race conditions and controlled state synchronization. This means that services and companies that use distributed architectures can count on more uptime, consistent performance, and trustworthiness. In fields like finance, e-commerce, and cloud services, where data availability and integrity are very important, the ability to consistently maintain state across distributed systems is very useful.

The Replica State Discovery Protocol is brought to light in the field of distributed systems through an analysis that shows how important it is. The protocol shows the level of sophistication that can be used in current distributed computing by solving problems with state management, synchronization, and how well nodes can communicate to each other.

The designed procedure controls these steps to lower the chance of race conditions and keep the integrity of the system’s state. The fact that state management is abstracted, and dependency injection is built in is more proof that the protocol follows modern software design principles. You can see how flexible and adaptable it is by the fact that it can work with many different types of distributed systems and meet their needs by using tools for managing state.

In the real world, the protocol could be used for a large spectrum of domains, starting from edge computing to cloud computing and computing for the Internet of Things. This shows that it is both useful and flexible. Because it can react so well to different operational situations, it is very useful in situations where precise coordination and synchronization are needed.

REFERENCES (TRANSLATED AND TRANSLITERATED)

1. *AMQP 0-9-1 Model Explained | RabbitMQ.* (n.d.). RabbitMQ: One broker to queue them all | RabbitMQ. https://rabbitmq-website.pages.dev/tutorials/amqp-concepts
2. *AMQP 0-9-1 Model Explained.* (n.d.). VMware Docs Home. https://docs.vmware.com/en/VMware-RabbitMQ-for-Kubernetes/1/rmq/tutorials-amqp-concepts.html
3. *AMQP vs. MQTT: 9 Key Differences - Spiceworks.* (2024). Spiceworks. https://www.spiceworks.com/tech/networking/articles/amqp-vs-mqtt/
4. *Chapter 8. Advanced Message Queuing Protocol (AMQP) Red Hat AMQ 6.3 | Red Hat Customer Portal*. (2024). Red Hat Customer Portal. https://access.redhat.com/documentation/en-us/red_hat_amq/6.3/html/connection_reference/amqp
5. *FAQ: What is AMQP and why is it used in RabbitMQ? - CloudAMQP.* (2024). CloudAMQP. https://www.cloudamqp.com/blog/what-is-amqp-and-why-is-it-used-in-rabbitmq.html
6. *Understanding AMQP, the protocol used by RabbitMQ.* (2024). Understanding AMQP, the protocol used by RabbitMQ. https://spring.io/blog/2010/06/14/understanding-amqp-the-protocol-used-by-rabbitmq
7. Novikov, I. (2021). *What is AMQP Protocol ? All you need to know.* Medium. https://d0znpp.medium.com/what-is-amqp-protocol-all-you-need-to-know-c9eedb680c71
8. Selvam, M. (2023). *AMQP—Introduction and Story of the RabbitMQ.* Medium. https://medium.com/@manikandanselvam_89994/amqp-introduction-and-story-of-the-rabbitmq-6f905980369a
9. Tezer, O. (2013). *An Advanced Message Queuing Protocol (AMQP) Walkthrough.* DigitalOcean | Cloud Infrastructure for Developers. https://www.digitalocean.com/community/tutorials/an-advanced-message-queuing-protocol-amqp-walkthrough

10. Panwar, S. (2023). Synchronizing Distributed Applications: Harnessing the Power of Distributed Systems. Medium. https://medium.com/@shasviv2006/synchronizing-distributed-applications-harnessing-the-power-of-distributed-systems-33c6f61abb73#:~:text=Synchronization%20in%20distributed%20systems%20is,happens%20in%20the%20right%20order
11. *Synchronization in a Distributed System | 8th Light.* (n.d.). 8th Light. https://8thlight.com/insights/synchronization-in-a-distributed-system
12. GAME. (2018). *Synchronization between nodes in a distributed system forming a blockchain.* Medium. https://medium.com/game/synchronization-609369558ce7
13. *How to Synchronize Distributed systems?* (n.d.). Programmer Prodigy. https://programmerprodigy.code.blog/2021/07/07/how-to-synchronize-distributed-systems/
14. Lawal, S. (2023). *Distributed Systems: Synchronisation in Complex Systems.* Backend Engineering w/Sofwan. https://blog.sofwancoder.com/distributed-systems-synchronisation-in-complex-systems
15. *Synchronization In A Distributed Operating System – LEMP.* (n.d.). LEMP – App & Tech Guides. https://lemp.io/what-is-synchronization-in-distributed-operating-system/
16. *Synchronization in Distributed Systems - GeeksforGeeks.* (n.d.). GeeksforGeeks. https://www.geeksforgeeks.org/synchronization-in-distributed-systems/
17. Pan, L. (2018). *State Machine and Synchronization*. Lu's blog. https://blog.the-pans.com/state-machine-and-sync/
18. Babitski, Y. (2020). *What Is Mutex?* Medium. https://medium.com/swlh/what-is-mutex-6127af8ced4f
19. Ibrahim, D. (n.d.). *Semapores and mutexes.* ResearchGate. https://www.researchgate.net/publication/341708618_Semapores_and_mutexes
20. *Mutexes and Semaphores Demystified.* (n.d.). Software Expert Witness | Barr Group. https://barrgroup.com/blog/mutexes-and-semaphores-demystified
21. *Semaphores and mutexes [LWN.net].* (n.d.). Welcome to LWN.net [LWN.net]. https://lwn.net/Articles/165039/

Котов Максим Сергійович
Магістр кібербезпеки, студент кафедри кібербезпеки та захисту інформації
Київський національний університет імені Тараса Шевченка, Київ, Україна
ORCID 0000-0003-1153-3198
maksym_kotov@ukr.net

Толюпа Сергій Васильович
д.т.н., професор, професор кафедри кібербезпеки та захисту інформації
Київський національний університет імені Тараса Шевченка, Київ, Україна
ORCID 0000-0002-1919-9174
tolupa@i.ua

Наконечний Володимир Сергійович
д.т.н., професор, професор кафедри кібербезпеки та захисту інформації
Київський національний університет імені Тараса Шевченка, Київ, Україна
ORCID 0000-0002-0247-5400
nvc2006@i.ua

ПРОТОКОЛ ВИЯВЛЕННЯ СТАНУ РЕПЛІКИ НА ОСНОВІ РОЗШИРЕНОГО ПРОТОКОЛУ ЧЕРГИ ПОВІДОМЛЕНЬ

Анотація. Коли справа доходить до ландшафту розподілених обчислень, який постійно змінюється, дуже важливо знати та розуміти, як підтримувати синхронізацію та узгодженість інформації про стан між репліками. Дане дослідження націлене на створення протоколу виявлення стану репліки, який побудований на основі розширеного протоколу черги повідомлень (AMQP). Метою цього дослідження є вивчення того, як створений протокол підтримує узгоджену інформацію про стан у різних репліках у розподілених системах. Почато дослідження з основ AMQP і того, чому він такий важливий для сучасних розподілених систем. Переглядаючи кожен рівень протоколу, було звернено увагу на загальну обробку даних і на те, як повідомлення передаються протягом кожного етапу. Проблеми, пов’язані з розробкою згаданого протоколу, є важливою темою цього дослідження. Непростим завданням є вирішення проблем, таких як стани гонитви, і забезпечення консистентних переходів між фазами. У даній роботі розглянуто теоретичні та практичні аспекти управління репліками стану. Дана стаття створена для тих, хто цікавиться або вже використовує розподілені обчислення.

Ключові слова: розподілені обчислення; синхронізація стану; Replica State Discovery Protocol (RSDP); Advanced Message Queuing Protocol (AMQP); послідовне управління станом; стан гонитви; керування репліками.

СПИСКИ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. *AMQP 0-9-1 Model Explained | RabbitMQ.* (n.d.). RabbitMQ: One broker to queue them all | RabbitMQ. https://rabbitmq-website.pages.dev/tutorials/amqp-concepts
2. *AMQP 0-9-1 Model Explained.* (n.d.). VMware Docs Home. https://docs.vmware.com/en/VMware-RabbitMQ-for-Kubernetes/1/rmq/tutorials-amqp-concepts.html
3. *AMQP vs. MQTT: 9 Key Differences - Spiceworks.* (2024). Spiceworks. https://www.spiceworks.com/tech/networking/articles/amqp-vs-mqtt/
4. *Chapter 8. Advanced Message Queuing Protocol (AMQP) Red Hat AMQ 6.3 | Red Hat Customer Portal*. (2024). Red Hat Customer Portal. https://access.redhat.com/documentation/en-us/red_hat_amq/6.3/html/connection_reference/amqp
5. *FAQ: What is AMQP and why is it used in RabbitMQ? - CloudAMQP*. (2024). CloudAMQP. https://www.cloudamqp.com/blog/what-is-amqp-and-why-is-it-used-in-rabbitmq.html
6. *Understanding AMQP, the protocol used by RabbitMQ.* (2024). Understanding AMQP, the protocol used by RabbitMQ. https://spring.io/blog/2010/06/14/understanding-amqp-the-protocol-used-by-rabbitmq

7. Novikov, I. (2021). *What is AMQP Protocol ? All you need to know.* Medium. https://d0znpp.medium.com/what-is-amqp-protocol-all-you-need-to-know-c9eedb680c71
8. Selvam, M. (2023). *AMQP—Introduction and Story of the RabbitMQ.* Medium. https://medium.com/@manikandanselvam_89994/amqp-introduction-and-story-of-the-rabbitmq-6f905980369a
9. Tezer, O. (2013). *An Advanced Message Queuing Protocol (AMQP) Walkthrough.* DigitalOcean | Cloud Infrastructure for Developers. https://www.digitalocean.com/community/tutorials/an-advanced-message-queuing-protocol-amqp-walkthrough
10. Panwar, S. (2023). Synchronizing Distributed Applications: Harnessing the Power of Distributed Systems. Medium. https://medium.com/@shasviv2006/synchronizing-distributed-applications-harnessing-the-power-of-distributed-systems-33c6f61abb73#:~:text=Synchronization%20in%20distributed%20systems%20is,happens%20in%20the%20right%20order
11. *Synchronization in a Distributed System | 8th Light.* (n.d.). 8th Light. https://8thlight.com/insights/synchronization-in-a-distributed-system
12. GAME. (2018). *Synchronization between nodes in a distributed system forming a blockchain.* Medium. https://medium.com/game/synchronization-609369558ce7
13. *How to Synchronize Distributed systems?* (n.d.). Programmer Prodigy. https://programmerprodigy.code.blog/2021/07/07/how-to-synchronize-distributed-systems/
14. Lawal, S. (2023). *Distributed Systems: Synchronisation in Complex Systems.* Backend Engineering w/Sofwan. https://blog.sofwancoder.com/distributed-systems-synchronisation-in-complex-systems
15. *Synchronization In A Distributed Operating System – LEMP.* (n.d.). LEMP – App & Tech Guides. https://lemp.io/what-is-synchronization-in-distributed-operating-system/
16. *Synchronization in Distributed Systems - GeeksforGeeks.* (n.d.). GeeksforGeeks. https://www.geeksforgeeks.org/synchronization-in-distributed-systems/
17. Pan, L. (2018). *State Machine and Synchronization*. Lu's blog. https://blog.the-pans.com/state-machine-and-sync/
18. Babitski, Y. (2020). *What Is Mutex?* Medium. https://medium.com/swlh/what-is-mutex-6127af8ced4f
19. Ibrahim, D. (n.d.). *Semapores and mutexes.* ResearchGate. https://www.researchgate.net/publication/341708618_Semapores_and_mutexes
20. *Mutexes and Semaphores Demystified.* (n.d.). Software Expert Witness | Barr Group. https://barrgroup.com/blog/mutexes-and-semaphores-demystified
21. *Semaphores and mutexes [LWN.net].* (n.d.). Welcome to LWN.net [LWN.net]. https://lwn.net/Articles/165039/