УДК 004.732:004.774 DOI: 10.31673/2412-4338.2024.039989

Maksym Kotov
Taras Shevchenko National University of Kyiv, Department of Cybersecurity and Information Protection, Kyiv, Ukraine
ORCID 0000-0003-1153-3198

Serhii Toliupa
Taras Shevchenko National University of Kyiv, Department of Cybersecurity and Information Protection, Kyiv, Ukraine
ORCID 0000-0002-1919-9174

Volodymyr Nakonechnyi
Taras Shevchenko National University of Kyiv, Department of Cybersecurity and Information Protection, Kyiv, Ukraine
ORCID 0000-0002-0247-5400

METHOD OF BUILDING LOCAL AREA NETWORK SIMULATION BASED ON AMQP AND ITS SUPPORT PROTOCOLS SUITE

Abstract. *Modern, scalable solutions require a simplified mode of communication management. To achieve scalability, replicated services or clustered solutions often utilize either a centralized management approach or a network of equally ranked peers. In the later case, a solution that manages the communication media should be implemented. Often, security in such networks is achieved through the use of VPNs (Virtual Private Networks) which allow for the abstraction of communication integrity and confidentiality components. To achieve addressing, network discovery, and direct communication, a complex set of network protocols is commonly utilized. Such setup puts an extreme burden on the engineers that have to design and implement complex network topology structures and integrate them with the application layer codebase. The purpose of this article is to propose a novel approach to building peer-to-peer networks based on the AMQP protocol. Through the course of the research, the internal workings of the AMQP protocol will be described through the implementation of the RabbitMQ broker. A description of the Local Area Network (LAN) simulation client is provided and examined. Additionally, performance metrics such as latency and throughput were considered and laid out for the use of a simulated LAN environment. This article provides security guidelines and considerations when building simulated LANs with a given method. On top of that, a set of replica support protocols was developed. Their purpose and description are provided as a clear illustration of potential LAN simulation method usage.*

Keywords: *Local Area Network (LAN); Advanced Message Queuing Protocol (AMQP); network simulation; message queuing; network architecture; network performance; communication protocols; data exchange; network topology; scalability; reliability; latency; throughput; distributed systems; real-time communication; interoperability; middleware.*

Котов Максим Сергійович
Київський національний університет імені Тараса Шевченка, Київ, Україна
ORCID 0000-0003-1153-3198

Толюпа Сергій Васильович
Київський національний університет імені Тараса Шевченка, Київ, Україна
ORCID 0000-0002-1919-9174

Наконечний Володимир Сергійович
Київський національний університет імені Тараса Шевченка, Київ, Україна
ORCID 0000-0002-0247-5400

МЕТОД ПОБУДОВИ СИМУЛЯЦІЇ ЛОКАЛЬНОЇ МЕРЕЖІ НА ОСНОВІ AMQP ТА НАБІР ПРОТОКОЛІВ ПІДТРИМКИ

Анотація. *Сучасні масштабовані рішення вимагають спрощеного режиму управління зв'язком. Щоб досягти масштабованості, репліковані служби або кластерні рішення часто використовують або централізований підхід до управління, або мережу однорангових вузлів. В останньому випадку повинно бути запроваджене рішення, яке керує засобами зв'язку. Часто безпека в таких мережах досягається за допомогою використання VPN (віртуальних приватних мереж), які дозволяють абстрагувати компоненти цілісності зв'язку та конфіденційності. Для досягнення адресації, виявлення мережі та прямого зв'язку зазвичай використовується складний набір мережевих протоколів. Таке налаштування створює надзвичайний тягар для інженерів, які повинні проектувати та впроваджувати складні топологічні структури мережі та інтегрувати їх із кодовою базою прикладного рівня. Мета цієї статті — запропонувати новий підхід до побудови однорангових мереж на основі протоколу AMQP. У ході дослідження буде описано внутрішню роботу протоколу AMQP через впровадження брокера RabbitMQ. Надано опис розробленого клієнта симуляції локальної мережі (LAN). Крім того, такі показники продуктивності, як затримка та пропускна здатність, були розглянуті для використання змодельованого середовища локальної мережі. У даній статті наведено міркування щодо безпеки під час побудови імітованих локальних мереж із заданим методом. Крім того, було розроблено набір протоколів підтримки реплік, їх призначення та опис надаються як чітка ілюстрація можливого використання методу моделювання локальної мережі.*

Ключові слова: *Local Area Network (LAN); Advanced Message Queuing Protocol (AMQP); імітація мережі; черга повідомлень; архітектура мережі; продуктивність мережі; комунікаційні протоколи; обмін даними; топологія мережі; масштабованість; надійність; затримка; пропускна здатність; розподілені системи; реальний час зв'язку; інтероперабельність; проміжне програмне забезпечення.*

Introduction

Building reliable distributed systems is an ever-challenging task that requires careful planning and comprehensive experience from researchers and engineers. An essential decision that must be made is choosing ranking relations between nodes inside the perimeter. Such a decision eventually comes down to either the vertical or horizontal control plane that manages the synchronization and coordination processes inside a replication group or a cluster.

In the case of the vertical control plane organization, there are designated leaders in a distributed system. Such architecture enforces a single source of authority and responsibility, which might be beneficial in the case of a complex state management system but is also much more perplexing when it comes to fault tolerance and robustness guarantees. It is of utmost importance for the control plane in such systems to provide recovery and an inner redundancy layer to achieve operational stability. Subsequently, said system organization requires far greater effort, experience and time from an engineering team responsible for the distributed system implementation.

On the other hand, peer-to-peer solutions offer no centralized authority and therefore must implement consensus algorithms to achieve state synchronization and coordination. Such solutions might come in a few variations, including ones that elect a single cluster leader that will serve as a designated temporal authority or those that rely completely on achieving horizontal consensus.

There are numerous leader election algorithms that allow for consensus, including Bully Algorithm, Ring Algorithm, Raft Consensus Algorithm, Paxos Algorithm, and others [1], [2]. In such systems, fault tolerance is achieved through the blurred lines between authority and followers due to the fundamental equality characterized by peer-to-peer networks. In case of the main node failure, a new leader will be promptly reelected, and a new network consensus will be achieved.

Another peer-to-peer variation is based on the complete abolishment of authority. These networks do not rely on any kind of centralized coordination and do not require such. A prominent example of such a system is described in the Replica State Discovery Protocol (RSDP) [3]. Synchronization in RSDP is achieved through the phases of state information exchange and

aggregation. In such a system, there is no downtime at all tied to the leader failure since each distinct cluster member knows exactly what it should be working on without any kind of centralized authority or coordination.

The mentioned RSDP protocol relies on a few distinct synchronization phases that are mirrored on each network peer and include debate rounds, sharing stages, and shutdown notifications. The protocol implements multiple mechanisms for race condition avoidance and eventual state completeness insurance [3]. For simplicity, throughout this article, networks built with the same principle in mind will be called Decentralized Coordination Networks (DCN).

Problem formulation. Building DCNs is a complex task that, at its core, requires a managed network environment that provides communication channels between peers. When building clustered or replicated solutions, it is usually implied that such communication channels must additionally implement security measures to guarantee the confidentiality and integrity of the message exchange.

An obvious and ubiquitous solution is the use of Virtual Private Network (VPN) solutions that imitate LAN over the Wide Area Network (WAN) and provide a security layer with native encryption and encapsulation mechanisms. An example of such a system is Wireguard protocol that allows for seamless and scalable management of VPNs [4].

The said solution is extremely popular but at the same time demands careful construction and consideration during architectural and infrastructural decisions. Managing a virtualized environment is a complex task that allows for network discovery and communication through a comprehensive suite of network protocols and services. A comprehensive set of subsystems that need to function properly to provide basic communication layer abstractions implies a comprehensive set of failure points that must be managed. Therefore, such a solution demands an immense amount of effort to maintain.

Additionally, communication through WAN with VPN comes with its own performance penalties and risks related to port mapping in NAT-based environments, such as sudden connection abruptions. These risks could be managed through the usage of static port mappings, methods that involve sending empty UDP datagrams, or a designated NAT Mapping Support Protocol (NMSP).

The purpose of the article. A new method of building LAN simulation systems is developed that resolves the issues mentioned and facilitates the rapid and seamless development of distributed systems. Throughout this paper, the implementation, performance, security, and reliability of simulated Local Area Networks over Advanced Message Queuing Protocol are described and evaluated. Additionally, a suite of state synchronization support protocols based on the said LAN simulation is developed and described to be eventually integrated into potential DCN systems of the future.

The goal of this research is to equip engineers of Decentralized Coordination Networks with a simple but fundamental communication abstraction method that allows simplified management of security, fault tolerance, and quality of services. As an addition, a practical showcase of said environment is done through a developed suite of state synchronization tools.

AMQP implementation overview through the rabbitmq broker

Advanced Message Queuing Protocol (AMQP) is an application layer protocol that, in its basis, relies on the complex routing infrastructure and asynchronous communication mode. AMQP is widely used in complex distributed systems that rely on the model of guaranteed and consistent task execution on a group of remote nodes. Its internal architecture allows for horizontal scaling, effectively allowing unrestricted processing and coordination infrastructure for the complex high-throughput systems in the contemporary digital landscape [5] – [9].

Incorporating AMQP alleviates complexities related to platform-specific implementations on both the hardware and operating system. There are numerous client-side implementations of the said protocol available for all the most frequently used platforms, thus eliminating any turbulence related to the development of network communication drivers and clients.

Mentioned AMQP clients additionally come into a broad spectrum of different programming languages such as JavaScript, Go, C++, Ruby, C#, PHP, Python, and the list goes on. That further facilitates the development process of contemporary distributed systems [10].

AMQP defines a set of communication rules between clients and its data format. The data transfer between communicating nodes is done completely by using binary data. Thus, achieving the overhead reduction that comes with some network protocols, which is related to inefficient data encoding schemas [9].

Said protocol additionally defines a conformant interface and procedures that the intermediary communication nodes, called “middlewares”, must implement in order to establish a well-defined communication environment. Such middleware nodes are commonly referred to as “brokers” and are responsible for implementing communication abstractions.

Communication abstractions include producers, consumers, exchanges, queues, and bindings [5], [7]. Out of the mentioned list, only producers and consumers are located on the client side and are implemented with the previously mentioned platform-specific libraries. Brokers are responsible for maintaining, synchronizing, scaling, and providing access for clients to the exchanges and queues. Queue bindings are not accessed directly by a client, as they represent a set of virtual connections between queues and exchanges.

The interaction process begins with a producer-initiated message being passed to the broker. First and foremost, the producer has to establish a connection that could be done in two recognized forms defined by the “publisher confirms” mechanism. This basically allows one to choose whether the communication between the producer and the broker should be done in a synchronous or asynchronous way. In the first case, for each message produced, the producer would then subsequently wait for the message acknowledgement from the broker. Upon timeout, the producer might resend the message or abort the process entirely. In the latter case, where communication is asynchronous, the producer would restore its original execution route without waiting for any kind of acknowledgement [11].

The producer is responsible for assigning metadata for each message that includes information about the exchange entity and routing parameters that should be used to deliver the message to a designated queue or a set of them. Upon receiving the message, the broker is responsible for the execution of the routing mechanism, which largely depends on the specific implementation of the exchange.

The process of AMQP message passing interaction is shown in Figure 1:

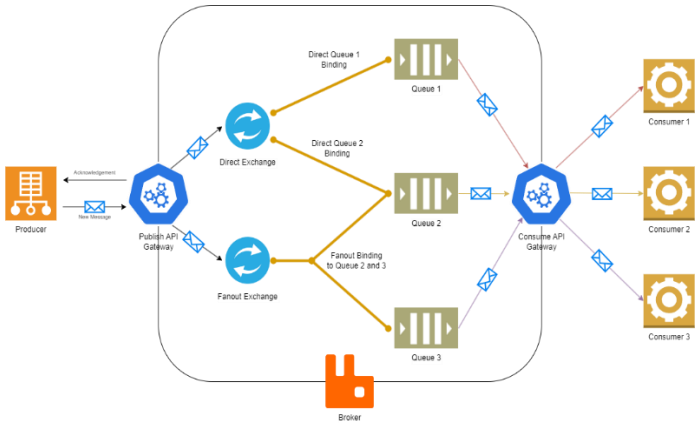

Fig. 1. AMQP interactions

AMQP defines four common types of exchanges that include direct, fanout, topic, and header variations. A direct exchange is used to send a message to a single distinct queue that has a binding rule tied to that particular exchange. The queue name serves as a routing address in such an exchange [5], [7].

The fanout exchange type is used to broadcast messages to all connected queues [5], [7]. Usage of such exchanges excludes the possibility of restricted communication, though it does not restrict the amount of bindings with other exchanges that the connected queues might have. This type of exchange is widely used for system-wide event notification.

Topic exchanges allow for group segregation during the routing process [5], [7]. Each binding between a queue and the exchange would include a string separated by dots that represent its routing keys. Said routing keys are used during decision-making to determine the target recipients of a given message. In cases where the routing key specified in the message data field is exactly the same as in a route, it will be passed to the queues behind such bindings. There are also special characters such as "*" that allow for pattern expansion.

Finally, header-based exchange goes a step further and allows for routing based on matching object fields [5], [7]. The similarity degree could be controlled during binding setup for the queues, allowing for the most granular approach during the rerouting process. Eventually, it has to be said that the more complex the communication mode, the more processing power it requires to resolve the eventual message recipients.

Queues act as temporal message storage mechanisms used during asynchronous communication processes. Though the latency will increase in comparison with the direct message exchange schema, queues provide an additional layer of resilience, durability, security, and accessibility that are crucial for the synchronization processes inside DCNs. In particular, RabbitMQ allows for a granular configuration of whether the queue should be durable, that is, whether it should survive system shutdowns and other lifecycle options that are tied to the number of consumers and exclusivity [5], [7], [12].

From the point of view of a message consumer, it does not matter how complex the internal routing process was or which metadata was attached to the message to facilitate decision-making on the broker abstraction layer. The consumer is only concerned with message retrieval and processing, which could also be done in two distinct forms: synchronous and asynchronous. The schema resembles procedures mentioned during producer-client descriptions, but with an important ideological exception during the confirmation process. During such asynchronous communication, messages will be removed from the queue as soon as the consumer receives them. In synchronous communication, the message remains in the queue until the consumer sends an explicit acknowledgement signal, indicating its complete processing. This allows for confirmation of not only successful communication but also successful task completion [11].

Having said that and completed the overview section of the internal AMQP operations, the following sections will describe how the LAN simulation is achieved and implemented. It's worth mentioning and emphasizing that the previously mentioned RSDP was built entirely on the AMQP/RabbitMQ infrastructure and provides additional insights on how this infrastructure may be used. Additional considerations for latency and security will be provided to help potential DCN-building engineers during the architectural decision-making process.

Description of the lan simulation agent

The simulation heavily relies on the capabilities of the underlying AMQP protocol and is built as a layer on top of it. The reasoning behind such dependency and choice is due to a few crucial factors, namely the reliability of the message propagation and the simplicity of the communication media control in a private environment.

Message propagation reliability comes from the service and functionality that the queues provide. The message will remain stored and might be configured to be flushed on the disk to avoid the risks associated with system reloads [12]. Thus, it allows for a guarantee in the DCNs that every participant will eventually receive the designated message. In turn, the interface simplicity allows to streamline the development process by avoiding complexities tied to reliability and delivery guarantee mechanisms.

The network simulation client relies on direct and fanout exchange integrations to establish a controlled communication medium. Fanout exchange is used to broadcast messages inside the network, and direct exchange is used to establish direct message passing between the peers. Such communication modes are abstracted out for potential usage with “broadcastMessage” and “sendMessage” respectively. It’s crucial to mention that the peers take upon themselves roles of producers and consumers simultaneously.

Before communication may occur between the client and the network, it should establish all the required queues, exchanges, and bindings, which is done with an abstract “initialize” method. Its responsibility is to set up the connection to the broker, its channel, and the fundamental communication components. Respectively, when the client wants to leave the network, it should call the “terminate” method to release resources.

The simulated LAN logical topology is shown in Figure 2:

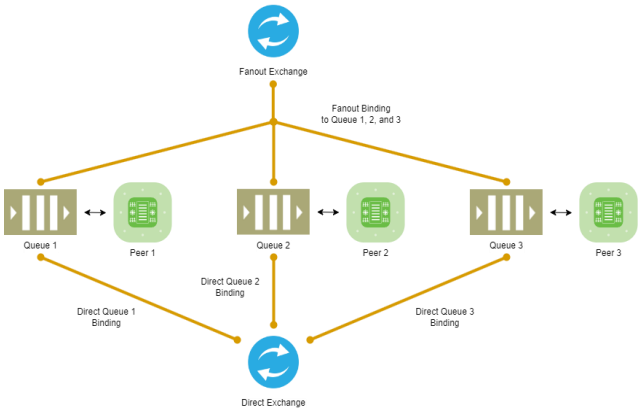

Fig. 2. Simulated network topology

The following code snippet shows a simplified implementation of the network simulation client:

```
class NetworkSimulationClient {
  static getGroupName(baseGroupName) {
    return `${baseGroupName}::${this.name}`;
  }

  constructor({
    amqpClient,
    rabbitGroupName,
  }) {
    this.amqpClient = amqpClient;
    this.rabbitGroupName = this.constructor.getGroupName(rabbitGroupName);

    this.consumerTag = null;
    this.queueId = null;
  }

  async initialize() {
    await this.amqpClient
      .getChannel()
      .addSetup(this.setupRabbitMQChannel.bind(this));
  }
```

```
 async terminate() {
  const channel = await this.amqpClient.getChannel();

  if (this.consumerTag) {
   await channel.cancel(this.consumerTag);
  }

  if (this.queueId) {
   await channel.unbindQueue(this.queueId, this.rabbitGroupName, '');
   await channel.deleteQueue(this.queueId);
  }
 }

 async setupRabbitMQChannel(channel) {
  await this.setupBroadcastExchange(channel);
  await this.setupQueue(channel);
  await this.startConsumer(channel);
 }

 async setupBroadcastExchange(channel) {
  await channel.assertExchange(this.rabbitGroupName, 'fanout', { durable: false });
 }

 async setupQueue(channel) {
  const queueAssertion = await channel.assertQueue('', { exclusive: true });

  this.queueId = queueAssertion.queue;

  await channel.bindQueue(this.queueId, this.rabbitGroupName, '');
 }

 async startConsumer(channel) {
  const consumerResponse = await channel.consume(this.queueId, this.handleMessage.bind(this),
{ noAck: true });

  this.consumerTag = consumerResponse.consumerTag;
 }

 async broadcastMessage(message) {
  const messageBuffer = Buffer.from(JSON.stringify(message));

  return this.amqpClient.getChannel().publish(this.rabbitGroupName, '', messageBuffer);
 }

 async sendMessage(address, message) {
  return this.amqpClient.sendToQueue(address, message);
 }

 async handleMessage() {
  throw new Error(`handleMessage method is not implemented for: ${this.constructor.name}`);
 }
```

```
}
```

The retry mechanism is not straightforward with the RabbitMQ broker and is not explicitly defined in the AMQP itself. If the consumer fails to acknowledge or reject the message due to a critical internal error or a shutdown, it will still reside inside the queue indefinitely.

To avoid such scenarios, AMQP supports a message time-to-live (TTL) mechanism. If the queue is configured with the TTL and the message resides longer than it is specified, it will be forwarded to the dead letters queue. Additionally, RabbitMQ allows defining a special dead letter queue for a specified functional queue. That mechanism is exactly what allows the creation of an additional retry mechanism layer within the simulated network [13].

During the creation of the functional queue, another custom queue designated for the invalid letters is created and specified as a dead letter queue for the main queue. The main queue has the TTL specified, so if it expires or the message gets rejected, it will be sent to the custom dead letter queue. In turn, the created custom invalid message queue will have the main queue specified as the dead letter queue for itself. Therefore, when the message in the invalid queue expires, it will be sent back to the main functional queue, and the consumption process will be re-executed.

It is of extreme importance to have a termination condition where the consumer, after a number of unsuccessful attempts, will eventually acknowledge the message to avoid an infinite execution of the same message that was ill-formatted or corrupted.

In the following code snippet, such logic is built with the usage of message headers and their subsequent incrementation on unsuccessful attempts:

```
async handleMessage(message) {
  if (!message) return;

  try {
    await this.process(message);
    this.amqpClient.getChannel().ack(message);
  } catch {
    await this.handleRetry(message);
  }
}

async handleRetry(message) {
  const headers = message.properties.headers || {};
  const retryCount = headers.retryCount || 1;

  if (retryCount <= this.retryLimit) {
    this.amqpClient
      .getChannel()
      .publish('', this.getRetryQueueAddress(), message.content, {
        headers: { ...headers, retryCount: retryCount + 1 },
      });
  }
  this.amqpClient.getChannel().ack(message);
}
```

The described client is the backbone for all the following DCN synchronization protocols that are described in this article. As a matter of fact, it is also a foundation upon which the RSDP implementation was built and relies heavily [3]. It makes the interaction process extremely simplified from a standpoint of development and architectural complexity.

The conditional messaging logic, therefore, could be defined inside the “handleMessage” method and might include but is not restricted to the different message types, verification and payload validation logic, identification, authentication, authorization, or virtually any required capability that might be needed for a distributed system. Additionally, the extensions in more complex cases might control the concurrency of the communication through the usage of mutexes and prefetch specifications, as will be shown in the following chapter [14].

Cluster management support protocols suite based on the lan simulation agent

Protocols described in this chapter are built as classes that extend the initial “NetworkSimulationClient” class described in the previous section. Internally, all the protocols go through the same set-up process, as described in the network simulation client, unless specified otherwise. Every protocol, thus, takes advantage of the simplified interface of communication that abstracts out every mechanism related to concurrency control, reliability guarantees, and infrastructure maintenance.

Each of the following protocols will describe high-level network communication logic and will manage an abstract entity called “managedTarget”. A managed target might be any service needed in the distributed system, and it just has to implement an interface that conforms to the expectations of the protocols.

Backoff Propagation Protocol overview. In time-sharing clusters, where the distributed workers refer to the same time-limited resource, the synchronization of rate limit policies is of utmost importance. The shared resource might change their security policy regarding frequency of API access at any time, especially if such resource is controlled by a third party. In such cases, the distributed system must promptly adjust itself to accommodate new access rules and avoid violations that might lead to temporary or permanent service denial.

For that purpose, the Backoff Propagation Protocol was developed. It introduces a basic interface for sharing the rate limit multiplier value with the network members. This way, when one node hits the rate limit, it will immediately notify the replica set to avoid any policy violations. Additionally, it defines the handling routines that handle incoming events from other peers to update their own state based on the information provided within the network.

The following code snippet shows a simplified implementation of the backoff propagation protocol:

```
class BackoffPropagationProtocol extends NetworkSimulationClient {
  static PREFETCH_COUNT = 1;

  async setupRabbitMQChannel(channel) {
    await super.setupRabbitMQChannel(channel);

    await channel.prefetch(this.constructor.PREFETCH_COUNT);
  }

  async broadcastBackoffEvent(multiplier) {
    return this.broadcast({ multiplier });
  }

  async handleMessage({ multiplier }) {
    if (!this.shouldReload(multiplier)) return;

    return this.managedTarget.reloadActive({
      multiplier,
    });
```

```
  }

  shouldReload(multiplier) {
    const currentMultiplier = this.managedTarget.getMultiplier();

    return multiplier && multiplier > currentMultiplier;
  }
}
```

Each message handling relies on the outsourced state checks for the management target. To avoid race conditions and inconsistencies that might happen in the concurrent setup, the prefetch count mechanism is used. Prefetch limits the number of concurrently processed messages to 1, thus effectively avoiding concurrency issues.

Dynamic Configuration Protocol overview. Managing distributed systems is a complicated task that might and often will lead to inconsistencies if the node configuration has to be done manually each time the requirements of the environment change. For that purpose, the Dynamic Configuration Protocol was developed to facilitate rapid state configuration sharing between nodes in an automated way.

The following code snippet shows a simplified implementation of the dynamic configuration protocol:

```
class DynamicConfigurationProtocol extends NetworkSimulationClient {
  static MESSAGE_TYPES = {
    UPDATE: 'UPDATE',
    RELOAD: 'RELOAD',
  };

  async handleMessage({ type, data }) {
    switch (type) {
      case DynamicConfigurationProtocol.MESSAGE_TYPES.UPDATE:
        return this.handleUpdateMessage(data);
      case DynamicConfigurationProtocol.MESSAGE_TYPES.RELOAD:
        return this.handleReloadMessage(data);
    }
  }

  async handleUpdateMessage({ targetId, workerIds }) {
    if (targetId !== this.managedTarget.id) return;

    return this.managedTarget.updateWorkers(workerIds);
  }

  async handleReloadMessage() {
    return this.managedTarget.reload();
  }
}
```

Its implementation provides insights into the abstraction principles mentioned in the previous chapter. Message handling routing might establish its own high-level conditional processing logic, as shown in the implementation of this protocol. Dynamic Configuration Protocol expects messages of two types, “UPDATE” and “RELOAD”, that will be passed inside the message object. That

information will be used during the decision-making process to choose an appropriate handling routine.

Metric Exports Protocol overview. Automated monitoring is one of the fundamental principles in distributed environments. Having a comprehensive set of tools that allow to promptly discover malicious or bogus activity is a necessity to avoid huge resource losses in critical infrastructure.

One of the most popular solutions used in enterprise-scaling decentralized applications is Prometheus. Prometheus allows for metrics storage, aggregation, visualization, and various types of metrics collection methods. The basic data collection modes include pull and push based. In the first case, Prometheus provides an additional daemon that will have an open port and will listen for incoming push instructions from the clients. Such a setup might prove useful if the dynamic scaling of the infrastructure is not required or expected. Otherwise, the task of managing state transfers and their consistency is extremely complex [15], [16].

Another mode of metrics export involves configuring the main Prometheus service to continuously poll a given set of endpoints on its own [17]. In turn, a set of such endpoints must implement an HTTP REST API interface conformant to the Prometheus specifications. Such an interface is responsible for exposing a set of state buckets that describe the current operational status of the node.

Metrics Export Protocol is used to notify a set of such Prometheus-compliant nodes about their state change to allow monitoring for a continuously varying number of cluster workers. The network for such infrastructure consists of two types of nodes: worker nodes and metric exporters. Metric exporters listen for the incoming state events from the workers and implement the REST API interface. Implementation of such nodes is out of the scope of this article. Such a set-up further augments the possible landscape of the network topology based on the LAN simulated environment by giving an example network consisting of heterogenous members.

The following code snippet shows a simplified implementation of the metrics export protocol on the worker node’s side:

```
class MetricsExportProtocol extends NetworkSimulationClient {
  static MESSAGE_TYPES = {
    UPDATE: 'UPDATE',
  };

  static getGroupName() {
    return 'METRICS_GROUP';
  }

  async handleMessage({ type }) {
    switch (type) {
      case MetricsExportProtocol.MESSAGE_TYPES.UPDATE:
        return this.debounceNotification();
    }
  }

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

    this.statusDebounceTimeout = setTimeout(
      this.onDemandNotification.bind(this),
      this.policiesConfigs.heartbeat.statusDebounceDelay,
```

```
    );
  }

  async configureRabbitMQChannel(channel) {
    await super.configureRabbitMQChannel(channel);

    await this.assertExportersBroadcastExchange(channel);
  }

  async assertExportersBroadcastExchange(channel) {
    await channel.assertExchange(PROMETHEUS_EXPORTER_EXCHANGE, 'fanout', {
      durable: false,
    });
  }
  async onDemandNotification() {
    const marketStatuses = this.managedTarget.getWorkerStatuses();
    this.notifyExporters(marketStatuses);
  }

  async notifyExporters(marketStatuses) {
    const messageBuffer = Buffer.from(JSON.stringify({ marketStatuses }));

    return this.amqpClient.getChannel()
      .publish(PROMETHEUS_EXPORTER_EXCHANGE, '', messageBuffer);
  }
}
```

The shown protocols might be modified and extended in different directions that could allow for integration of the proprietary endpoint solutions. A collection of said protocols could be used simultaneously to achieve horizontal logic scaling and modularity. Complex intercommunication topologies could be built with inter-protocol communication systems that allow to relay information and events on particular conditions met in the received messages.

In more complex communication schemas, it might be needed to establish concurrency control through the means of mutexes and prefetch counts. Examples of how mutexes might help to control protocol stages were shown in the RSDP implementation [3]. In turn, prefetch is a mechanism described in the RabbitMQ specification that allows for control of the amount of simultaneous message processing, e.g., parallelism degree [18].

Evaluation of the latency and throughput in a simulated lan

Throughput considerations for the proposed method. The AMQP implementation provided by the RabbitMQ broker has well-known limitations regarding its scalability. RabbitMQ supports horizontal scaling, which, in theory, should allow for an unrestricted growth in throughput capabilities. In fact, it highly depends on the logical architecture of a given solution [19].

Entities defined in AMQP could be divided into two distinct categories: stateful and stateless. Stateless entities, such as exchanges and binds, can be replicated on other nodes and provide the same service without additional synchronization. Queues are stateful and cannot provide service from multiple nodes without synchronization. Queues are replicated asynchronously, and even if accessed from the replica, the request will be readdressed to the main node. Therefore, it is of utmost importance to design queues in such a way that they are responsible for as small a slice of the state as possible.

Benchmark method description. In this section, latencies are compared between communication in the networks built with the proposed method and HTTP-based data exchange. In both cases, the Node.js platform was used to build automated benchmark tests. The experiment aims to show relative rates of growth with the rising load and, by no means, should be treated as reference data for all communication instances due to the great influence of the underlying hardware and software technologies used.

In both benchmark instances, peers were running as processes on different machines to avoid interference with the inner process load related to sending and receiving the message. To calculate the latency, the sending peer received the timestamp with the "Date.now()" method and attached it as a message payload. The said method provides a number of milliseconds from a commonly known epoch described in ECMAScript [20]. It does not provide microsecond precision, but for evaluating relative growth, that will suffice.

It has to be mentioned that the latency is influenced not only by the time it takes to transfer the message but also by the conjunction created during the production process. Therefore, this benchmark effectively reflects how many resources are required to push the message into the network, transfer the data and receive it.

AMQP-based LAN simulation latencies. The following table shows the results for the proposed communication method benchmarks:

Table 1

Latency evaluation for LAN simulations based on AMQP

Number of simultaneous requests	Minimal latency in milliseconds	Average latency in milliseconds	95th percentile latency in milliseconds	Maximum latency in milliseconds
10	5.00	5.10	6.00	6.00
100	10.00	10.27	11.00	11.00
1000	45.00	49.16	52.00	53.00
5000	149.00	152.22	155.00	156.00
10000	271.00	275.42	280.00	283.00
20000	527.00	533.57	543.00	545.00
30000	722.00	740.89	752.00	755.00

Table 1 shows the results of benchmarks with the setup that includes two peers communicating through the RabbitMQ broker using a direct exchange. The latency shows the number of milliseconds it took from the point when one peer called "sendMessage" to the point until another peer received that message inside the "handleMessage" method.

HTTP communication latencies. The following table shows the results for the communication over HTTP benchmarks:

Table 2

Latency evaluation for direct network communication over HTTP

Number of simultaneous requests	Minimal latency in milliseconds	Average latency in milliseconds	95th percentile latency in milliseconds	Maximum latency in milliseconds
10	13.00	15.60	26.00	26.00
100	29.00	34.40	42.00	54.00
1000	222.00	252.59	299.00	320.00
5000	682.00	896.02	1122.00	1172.00
10000	1006.00	1457.71	1905.00	1988.00
20000	6644.00	7395.56	8142.00	8265.00
30000	14646.00	15790.65	16751.00	16918.00

Table 2 shows the latency results between two peers, where one of them had an open port waiting for requests and the other sent the requests. The results show that the communication channel based on AMQP had much lower latencies and much greater capability to adapt to the growing load. That suggests that the congestion during message processing, network traversal, and eventual processing by the receiving side is much more efficient with the use of the proposed method.

It might have been expected that the latency would be greater for the AMQP since it is based on asynchronous communication principles, but such a result comes from the fact that HTTP is a general-purpose stateless protocol that is not optimized for high-throughput communication. HTTP uses much more CPU time to parse an inefficient data format and might require using multiple handshake sessions for each subsequent request. Mechanisms such as "keep alive" might improve the performance but are still far behind the highly optimized model of binary data transmission described within AMQP.

Security considerations for building simulated networks

In the following chapter, security mechanisms included in the AMQP specification are described. In order to achieve access management along with data security and confidentiality, the said protocol natively incorporates Simple Authentication and Security Layer (SASL) for authentication, Transport Layer Security (TLS) for encryption, and Claims-Based Security (CBS) for authorization [21] – [24].

Simple Authentication and Security Layer. SASL provides authentication services within the scope of AMQP operations. It is a framework that is built to facilitate pluggable modes of authentication mechanisms by providing a core logic to manage and expose a unified API that allows to abstract out specific implementation details of the authentication process.

The core logic of SASL does not contain any definitions regarding the actual implementation of the authentication mechanism. When a client that supports SASL reaches the server, they first exchange information about supported mechanisms and, after achieving consensus, execute the authentication process [21], [22].

The authentication mechanisms mentioned in the AMQP specification include [24]:

– "PLAIN" is a mechanism that transfers a user's credentials in plain text with user identification and password specified. It is of utmost importance in such a scenario to use transport encryption; otherwise, the data might be pilfered and tempered with;

– "EXTERNAL" is used in environments that rely on external authentication mechanisms. Such mechanisms might include client certificates in environments where mutual TLS authentication is used;

– "ANONYMOUS" does not require authentication and grants access on every request. It's not used in production environments for obvious reasons and is mostly utilized for internal infrastructure testing environments and proof-of-concept.

Traffic encryption and authorization layers. To achieve confidentiality and integrity in communication, AMQP specifies the use of the Transport Layer Security protocol. TLS employs both symmetric and asymmetric encryption algorithms to distribute keys and establish a secure, confidential communication channel. It leverages the capabilities of the Public Key Infrastructure (PKI) to authenticate servers and verify identity authenticity. Additionally, it describes the use of HMAC algorithms that guarantee the integrity of the communication process [23].

In turn, the Claims-Based Security layer is responsible for request authorization based on tokens given by a trusted entity [24]. Tokens issued by a CBS subsystem include information about resources and permissions of the client to modify them or interact with them. The token can also contain an additional set of identity attributes tied to the user that might be used during the decision-making process.

Each token in CBS must contain a set of claims, which basically describe what resource is in the system and what action the current identity might take. To speed up the contextualization and authorization processes, AMQP defines a token cache component, which is responsible for holding authorization tokens in a fast-access memory. Tokens could be placed both during the SASL session

and through access to the CBS management entity and will be retrieved for verification each time a client sends a subsequent request.

Specific implementations of AMQP, such as RabbitMQ, often add their own extensions to facilitate high enterprise demands for security and versatility. For example, RabbitMQ defines a few additional SASL mechanisms, such as "AMQPLAIN", being a non-standard version of "PLAIN" enabled by default, and "RABBIT-CR-DEMO", which introduces a challenge-response cycle to establish authentication [25].

Conclusions

Having a reliable, secure, and simplified interface for managing and interacting with communication media is of utmost importance for building Decentralized Coordination Networks. In this research, the existing methods of communication and security management in distributed systems were discussed, and their shortcomings were shown.

Throughout this research, a new method of building Local Area Network simulation based on Advanced Message Queuing Protocol was developed. An analysis of the internal workings of AMQP through the implementation provided by RabbitMQ was shown and explained in detail. Said analysis served as a theoretical foundation for the description of the LAN simulation agent, which implements the developed method.

The simplified implementation of LAN simulation agent described in the "NetworkSimulationClient" class showed the fundamental concepts of the proposed method. On top of that base implementation, an additional layer of consistency guarantees could be developed for production-ready applications, as shown with the retry mechanism extension. Additionally, the internal limitations of AMQP and RabbitMQ were discussed.

As a practical showcase of potential use cases, a set of simple state management protocols for DCN systems was developed and described in detail. Each subsequent protocol introduced new potential coordination and extension concepts that could be used in more complex state management protocols.

On top of that, throughput considerations were provided for the usage of said method and AMQP-compliant solutions in general, providing crucial information during architectural decision-making processes. An evaluation of latency related to the usage of AMQP and HTTP protocols and their comparison was provided. The reasons behind the performance results obtained were described and discussed.

Therefore, in this article, a new reliable and simple method of building network simulations was shown with concrete examples of its potential use-cases and performance estimations. The said method has great potential for building network protocols for future DCN-based distributed systems.

References

1. Prateek S. Leader election in distributed systems – A deep dive!. LinkedIn: Log In or Sign Up. URL: https://www.linkedin.com/pulse/leader-election-distributed-systems-deep-dive-saurav-prateek/ (date of access: 11.04.2024).

2. Leader Election Algorithms: History and Novel Schemes. IEEE Xplore. URL: https://ieeexplore.ieee.org/abstract/document/4682163 (date of access: 12.04.2024).

3. Kotov M., Toliupa S., Nakonechnyi V. REPLICA STATE DISCOVERY PROTOCOL BASED ON ADVANCED MESSAGE QUEUING PROTOCOL. Cybersecurity: Education, Science, Technique. 2024. Vol. 3, no. 23. URL: https://doi.org/10.28925/2663-4023.2024.23.156171 (date of access: 13.04.2024).

4. WireGuard: Next Generation Kernel Network Tunnel – NDSS Symposium. NDSS Symposium. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/wireguard-next-generation-kernel-network-tunnel/ (date of access: 18.04.2024).

5. AMQP 0-9-1 model explained | rabbitmq. RabbitMQ: easy to use, flexible messaging and streaming | RabbitMQ. URL: https://rabbitmq-website.pages.dev/tutorials/amqp-concepts (date of access: 23.04.2024).

6. OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0, Part 3: Messaging. OASIS. URL: https://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-messaging-v1.0-os.html (date of access: 23.04.2024).

7. Selvam M. AMQP – introduction and story of the rabbitmq. Medium. URL: https://medium.com/@manikandanselvam_89994/amqp-introduction-and-story-of-the-rabbitmq-6f905980369a (date of access: 23.04.2024).

8. Tezer O. S. An advanced message queuing protocol (AMQP) walkthrough. DigitalOcean | Cloud Infrastructure for Developers. URL: https://www.digitalocean.com/community/tutorials/an-advanced-message-queuing-protocol-amqp-walkthrough (date of access: 23.04.2024).

9. Ashtari H. AMQP vs. MQTT: 9 key differences – spiceworks. Spiceworks. URL: https://www.spiceworks.com/tech/networking/articles/amqp-vs-mqtt/ (date of access: 03.02.2024).

10. Clients Libraries and Developer Tools | RabbitMQ. RabbitMQ: One broker to queue them all | RabbitMQ. URL: https://www.rabbitmq.com/client-libraries/devtools (date of access: 25.04.2024).

11. Consumer Acknowledgements and Publisher Confirms | RabbitMQ. RabbitMQ: One broker to queue them all | RabbitMQ. URL: https://www.rabbitmq.com/docs/confirms (date of access: 26.04.2024).

12. Reliability Guide | RabbitMQ. RabbitMQ: One broker to queue them all | RabbitMQ. URL: https://www.rabbitmq.com/docs/reliability (date of access: 28.04.2024).

13. Dead Letter Exchanges | RabbitMQ. RabbitMQ: One broker to queue them all | RabbitMQ. URL: https://www.rabbitmq.com/docs/dlx (date of access: 03.05.2024).

14. Ibrahim D. Semapores and mutexes. ResearchGate. URL: https://www.researchgate.net/publication/341708618_Semapores_and_mutexes (date of access: 04.05.2024).

15. Overview | Prometheus. Prometheus – Monitoring system & time series database. URL: https://prometheus.io/docs/introduction/overview/ (date of access: 06.05.2024).

16. When to use the Pushgateway | Prometheus. Prometheus – Monitoring system & time series database. URL: https://prometheus.io/docs/practices/pushing/ (date of access: 08.05.2024).

17. Pull doesn’t scale – or does it? | Prometheus. Prometheus – Monitoring system & time series database. URL: https://prometheus.io/blog/2016/07/23/pull-does-not-scale-or-does-it/ (date of access: 08.05.2024).

18. Consumer Prefetch | RabbitMQ. RabbitMQ: One broker to queue them all | RabbitMQ. URL: https://www.rabbitmq.com/docs/consumer-prefetch (date of access: 10.05.2024).

19. Quorum Queues | RabbitMQ. RabbitMQ: One broker to queue them all | RabbitMQ. URL: https://www.rabbitmq.com/docs/quorum-queues (date of access: 14.05.2024).

20. Date – JavaScript | MDN. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date#the_epoch_timestamps_and_invalid_date (date of access: 16.05.2024).

21. OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0, Part 5: Security. OASIS. URL: https://docs.oasis-open.org/amqp/core/v1.0/amqp-core-security-v1.0.html (date of access: 18.05.2024).

22. SASL Overview (GNU Simple Authentication and Security Layer 2.2.1). The GNU Operating System and the Free Software Movement. URL: https://www.gnu.org/software/gsasl/manual/html_node/SASL-Overview.html (date of access: 18.05.2024).

23. RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2. IETF Datatracker. URL: https://datatracker.ietf.org/doc/html/rfc5246 (date of access: 22.05.2024).

24. AMQP Claims-based Security Version 1.0. OASIS. URL: https://docs.oasis-open.org/amqp/amqp-cbs/v1.0/amqp-cbs-v1.0.html (date of access: 24.05.2024).

25. Authentication, Authorisation, Access Control | RabbitMQ. RabbitMQ: One broker to queue them all | RabbitMQ. URL: https://www.rabbitmq.com/docs/access-control (date of access: 27.05.2024).

Список використаної літератури

1. Prateek S. Leader election in distributed systems - A deep dive!. LinkedIn: Log In or Sign Up. URL: https://www.linkedin.com/pulse/leader-election-distributed-systems-deep-dive-saurav-prateek/ (date of access: 11.04.2024).

2. Leader Election Algorithms: History and Novel Schemes. IEEE Xplore. URL: https://ieeexplore.ieee.org/abstract/document/4682163 (date of access: 12.04.2024).

3. Kotov M., Toliupa S., Nakonechnyi V. REPLICA STATE DISCOVERY PROTOCOL BASED ON ADVANCED MESSAGE QUEUING PROTOCOL. Cybersecurity: Education, Science, Technique. 2024. Vol. 3, no. 23. URL: https://doi.org/10.28925/2663-4023.2024.23.156171 (date of access: 13.04.2024).

4. WireGuard: Next Generation Kernel Network Tunnel - NDSS Symposium. NDSS Symposium. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/wireguard-next-generation-kernel-network-tunnel/ (date of access: 18.04.2024).

5. AMQP 0-9-1 model explained | rabbitmq. RabbitMQ: easy to use, flexible messaging and streaming | RabbitMQ. URL: https://rabbitmq-website.pages.dev/tutorials/amqp-concepts (date of access: 23.04.2024).

6. OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0, Part 3: Messaging. OASIS. URL: https://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-messaging-v1.0-os.html (date of access: 23.04.2024).

7. Selvam M. AMQP – introduction and story of the rabbitmq. Medium. URL: https://medium.com/@manikandanselvam_89994/amqp-introduction-and-story-of-the-rabbitmq-6f905980369a (date of access: 23.04.2024).

8. Tezer O. S. An advanced message queuing protocol (AMQP) walkthrough. DigitalOcean | Cloud Infrastructure for Developers. URL: https://www.digitalocean.com/community/tutorials/an-advanced-message-queuing-protocol-amqp-walkthrough (date of access: 23.04.2024).

9. Ashtari H. AMQP vs. MQTT: 9 key differences - spiceworks. Spiceworks. URL: https://www.spiceworks.com/tech/networking/articles/amqp-vs-mqtt/ (date of access: 03.02.2024).

10. Clients Libraries and Developer Tools | RabbitMQ. RabbitMQ: One broker to queue them all | RabbitMQ. URL: https://www.rabbitmq.com/client-libraries/devtools (date of access: 25.04.2024).

11. Consumer Acknowledgements and Publisher Confirms | RabbitMQ. RabbitMQ: One broker to queue them all | RabbitMQ. URL: https://www.rabbitmq.com/docs/confirms (date of access: 26.04.2024).

12. Reliability Guide | RabbitMQ. RabbitMQ: One broker to queue them all | RabbitMQ. URL: https://www.rabbitmq.com/docs/reliability (date of access: 28.04.2024).

13. Dead Letter Exchanges | RabbitMQ. RabbitMQ: One broker to queue them all | RabbitMQ. URL: https://www.rabbitmq.com/docs/dlx (date of access: 03.05.2024).

14. Ibrahim D. Semapores and mutexes. ResearchGate. URL: https://www.researchgate.net/publication/341708618_Semapores_and_mutexes (date of access: 04.05.2024).

15. Overview | Prometheus. Prometheus - Monitoring system & time series database. URL: https://prometheus.io/docs/introduction/overview/ (date of access: 06.05.2024).

16. When to use the Pushgateway | Prometheus. Prometheus - Monitoring system & time series database. URL: https://prometheus.io/docs/practices/pushing/ (date of access: 08.05.2024).

17. Pull doesn't scale - or does it? | Prometheus. Prometheus - Monitoring system & time series database. URL: https://prometheus.io/blog/2016/07/23/pull-does-not-scale-or-does-it/ (date of access: 08.05.2024).

18. Consumer Prefetch | RabbitMQ. RabbitMQ: One broker to queue them all | RabbitMQ. URL: https://www.rabbitmq.com/docs/consumer-prefetch (date of access: 10.05.2024).

19. Quorum Queues | RabbitMQ. RabbitMQ: One broker to queue them all | RabbitMQ. URL: https://www.rabbitmq.com/docs/quorum-queues (date of access: 14.05.2024).

20. Date - JavaScript | MDN. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date#the_epoch_timestamps_and_invalid_date (date of access: 16.05.2024).

21. OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0, Part 5: Security. OASIS. URL: https://docs.oasis-open.org/amqp/core/v1.0/amqp-core-security-v1.0.html (date of access: 18.05.2024).

22. SASL Overview (GNU Simple Authentication and Security Layer 2.2.1). The GNU Operating System and the Free Software Movement. URL: https://www.gnu.org/software/gsasl/manual/html_node/SASL-Overview.html (date of access: 18.05.2024).

23. RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2. IETF Datatracker. URL: https://datatracker.ietf.org/doc/html/rfc5246 (date of access: 22.05.2024).

24. AMQP Claims-based Security Version 1.0. OASIS. URL: https://docs.oasis-open.org/amqp/amqp-cbs/v1.0/amqp-cbs-v1.0.html (date of access: 24.05.2024).

25. Authentication, Authorisation, Access Control | RabbitMQ. RabbitMQ: One broker to queue them all | RabbitMQ. URL: https://www.rabbitmq.com/docs/access-control (date of access: 27.05.2024).