International Information and Engineering Technology Association
Advancing the World of Information and Engineering

Ingénierie des Systèmes d'Information
Vol. 28, No. 3, June, 2023, pp. 655-662
Journal homepage: http://iieta.org/journals/isi

Handling HTTP Flood Attacks in High-Load Applications Using Akka Actors Model

Kairat Tokpayev*, Agyn Bedelbayev, Anar Iskendirova

Faculty of Information Technology, Al-Farabi Kazakh National University, Almaty 050040, Republic of Kazakhstan

Corresponding Author Email: tokpayevkairat@aol.com

https://doi.org/10.18280/isi.280314

Received: 20 February 2023
Accepted: 3 May 2023

Keywords:
DDoS attacks, microservice systems, Scala functional language, computing hardware, service-oriented asynchronous programming

ABSTRACT

The subject of this scientific research is the study of the principles of using asynchronous programming methods to process flood attacks and implement effective methods to handle HTTP Flood attacks on servers. The relevance of this research lies in the need to develop a practical example of using the Akka framework implementation for building microservices based on the Scala language. A practical combination of system analysis and observation of successive stages of the formation of the Akka actor to handle an HTTP Flood attack forms the basis of the methodological approach in this research. This provides ample opportunities to handle DDoS attacks in high-load applications through the use of the Akka Actors model, which aligns with the provisions of the Digital Kazakhstan state program. The practical significance of the results obtained in this research lies in the prospect of their implementation in creating microservice systems with strict API rules and a distributed system strategy, capable of handling millions of active users and ignoring potential fraud, including DDoS attacks.

1. INTRODUCTION

This scientific paper addresses the urgent need to counter server attacks, which have become more prevalent due to the emergence of tools that facilitate such attacks, including Distributed Denial of Service (DDoS) attacks such as the popular HTTP Flood. These attacks impose a heavy load on server applications and can potentially cause server failure if requests are delegated incorrectly. The resulting server failures have significant consequences and complicate server functioning [1]. DDoS attacks continue to be a significant threat, despite the availability of methods to detect and mitigate their consequences, such as monitoring network traffic and searching for anomalies, as well as technological solutions based on distributed query processing technology using actor models [2-4].

The Akka Actors technology, which is based on the Scala programming language, presents an efficient approach to asynchronous request processing in high-load systems, reducing potential damage to server infrastructure. Akka Actors is a high-level abstraction model for building parallel and distributed services, providing a strategy for writing parallel systems and simplifying the writing of services during thread construction. This approach aligns with the "Digital Kazakhstan" program's provisions [5, 6].

The increasing role of the internet in our lives has led to a rise in cybercrime, both financially and politically. Symantec statistics reveal a staggering 81% growth in the total number of attacks since 2011, reaching 5.5 billion in subsequent years. Agha [7] further explores the topic of parallel computing models in distribution systems, highlighting the mutual influence of actors in the system and their ability to send messages within the system. The Akka Actors model ensures high-quality interaction by maintaining a constant readiness to receive messages and allowing for dynamic changes in actor relationships, increasing the flexibility of managing this process [8, 9].

Weiser [10] draws attention to the latest developments in hardware systems for accelerating computing processes that have a clear focus on experimental platforms, resulting in improved operational efficiency in the future. Yoshioka et al. [11] discuss the general principles of stochastic optimization of the mixed moving average process, emphasizing the importance of considering the sequence of mathematical operations that describe the processing of random requests when creating a stochastic optimization model. In their joint scientific study on asynchronous global types in collaborative logic programming, Bianchini and Dagnino [12] draw attention to the significance of global types in communication programming, enabling high-level protocol specifications with a large number of participants involved and effective protection against external influences, including DDoS attacks.

The purpose of this research paper is to review the technology for building the Akka Actors model and provide a practical example of its implementation using the Akka framework. This study emphasizes the need to address HTTP Flood attacks in high-load applications and explores the potential of the Akka Actors model to enable efficient and reliable request processing in distributed systems. Cited references are included throughout the text.

2. MATERIALS AND METHODS

The basis of the methodological approach in this research paper is a combination of a systematic analysis of the general principles and causes of DDoS attacks on high-load applications with the observation of the sequence of creating an Akka actor to handle an HTTP Flood attack. The theoretical basis of this scientific research is the analyzed results of scientific papers of a number of scientists who studied the problematic aspects of the practical application of service-oriented asynchronous programming as a method of legal actions in DDoS attacks. A systematic analysis of the principles of the occurrence of DDoS attacks on high-load applications made it possible to determine the scheme of an HTTP Flood attack on the web server architecture, which is necessary to understand the significance of the HTTP protocol as a targeted application layer attack protocol. This made it possible to create a graphical diagram of an attack of this kind, for a visual illustration of the process described in this scientific study. In addition, through the application of a system analysis of the key principles of the occurrence of DDoS attacks, an overview was performed of the HTTP Flood, as the most well-known server attack in high-load applications to date.

The observation of the sequence of formation of actors for handling the HTTP Flood attack made it possible to determine the model hierarchy of actors, from the parent actor to the creation of its child forms. Akka's actor hierarchy model on the example of some Internal Actor branch has been represented in the corresponding graphic. When conducting a scientific research, materials were used that make up its theoretical base and reflect the main aspects of creating the Akka Actors model for handling an HTTP Flood attack in high-load applications. This made it possible to create a graphical representation of the performance of web servers, which is necessary to form a comparative assessment of their throughput and visually illustrate the practical benefits of using the Akka Actors model [13].

In addition, this scientific research used the materials of the electronic resource [13, 14] (Scala Version 2.7.0) containing data on the construction of the hierarchy of Akka Actors, as well as the interaction between parent and child actors. This made it possible to obtain and present code fragments in the Scala language (Version 2.7.0), showing examples of data failure processing when receiving information from one of the actors included in the system. Scala code fragments were also obtained, displaying the actor model of the API (Application Programming Interface) microservice system and the main service. This is necessary to create a meaningful illustration of the actor hierarchy in the Akka Actors model, as well as to understand the sequence of procedures executed in order to perform additional checks before writing to the server database.

3. RESULTS

The “Digital Kazakhstan” development program adopted at the state level defines the key directions for the development of digital sectors of the economy. In this context, special attention is paid to the use of digital technologies in the medium term and the transition of the country's economy to a new development trajectory, which involves the formation of a digital economy in the long term [5]. This determines the need to find the best ways to counter server attacks that pose a significant threat to information stored on servers.

The most well-known attack to date is the HTTP Flood attack [15]. Due to high resource consumption, this type of DDoS attack can stop the service for clients and fill up the client-server socket channel, which can be detected by network monitoring. The main concepts behind this are an attack on the heaviest APIs, basically its “get” request with filter parameters and lots of response data. Due to network traffic monitoring, the website administrator or automatic system alerts can predict and block the source IP (Internet Protocol) address that is sending the HTTP Flood. However, in a situation where an attacker could try to imitate the behavior of a normal user using this endpoint, it became more difficult to quickly recognize what is happening in the process of servicing infrastructure resources. For these purposes, it is necessary to constantly monitor the use of the metric dashboard, analyze daily highload statistics and solve this problem through the synergy of the two approaches [16]. In DDoS attacks, detection and monitoring may be interrupted or disrupted, depending on the nature and severity of the attack.

DDoS attacks involve overwhelming a target system or network with a flood of traffic from multiple sources, rendering it inaccessible to legitimate users. These attacks can be highly disruptive, causing significant downtime, financial losses, and damage to the reputation of the targeted organization. Detecting and monitoring DDoS attacks is essential to mitigate their impact and prevent further damage. However, the effectiveness of detection and monitoring can be limited by the size and complexity of the attack, the sophistication of the attackers, and the capacity of the monitoring systems.

In some cases, DDoS attacks can be so intense that they overload and disrupt monitoring systems, making it difficult or impossible to detect and respond to the attack. Additionally, attackers may use tactics such as "low and slow" attacks, which are designed to evade detection by gradually increasing the volume of traffic over time. Therefore, while detection and monitoring are critical components of DDoS mitigation, they may not always be uninterrupted or foolproof, and other strategies such as prevention, mitigation, and response planning are also necessary to protect against DDoS attacks.

In this case, ACL also plays a significant role – API access control levels at the stage of authorizing a user HTTP request. Knowing the role that has been assigned to users, such resource intensive requests could be encapsulated from the invention. Figure 1 is a schematic representation of an HTTP Flood attack on a web server architecture.

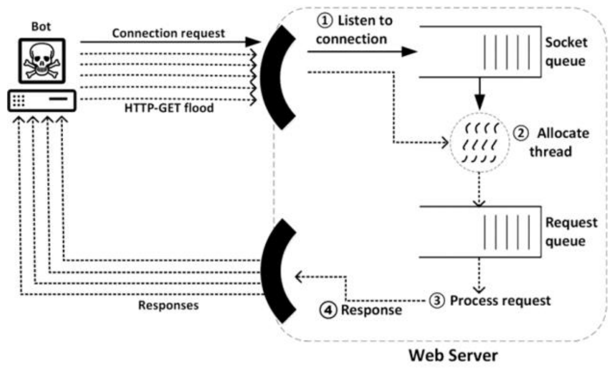

Figure 1. HTTP Flood attack on the web server architecture
Source: reference [17]

Considering all application layer protocols, HTTP is the most targeted application layer attack protocol due to its increasing role in the example of daily expanding online services. In addition, the attackers know that the default HTTP traffic protocol is not blocked by any company policies or infrastructure strategy policies. Web application protocols are not always enough to protect the provided methods, and the functionality of the Akkaactor-based microservice building model plays an important role in simultaneously processing, and thus, flooding requests from attackers and maintaining a high load on the service as a load balancer from the side of actors. Using parameters for distributed values and network traffic statistics, microservice systems can detect HTTP Flood behavior on a signal based on real-time metric parameters, and using the Akka actor model, a read flood request can be processed simultaneously by actors in the Scala programming language. By analyzing the final result of the metric data, it is possible to form and make a decision in a timely manner, which enabled taking care of HTTP Flood attacks and alerting subsystems to block these types of attacks at any access control level.

The actor's model hierarchy goes from top to bottom. The implementation of such a model helps to stop the development of the process of managing basic behavior on the low-level side of creating a service and allows getting all the features for creating and managing the life cycle of an actor, as well as handling errors in the system. The actor hierarchy is fundamentally strong for strategies, in which Akka Actors belong to a parent actor, so it is possible to create "child" actors and directly control the behavior of the entire service.

In Akka, a parent actor is an actor that creates and supervises one or more child actors. Child actors are actors that are created by a parent actor and are responsible for carrying out specific tasks or functions. The parent actor-child actor relationship is a fundamental concept in the Akka actor system, as it enables the creation of complex, hierarchical actor structures that can be used to model and manage the behavior of a service or application.

When a parent actor creates a child actor, it becomes the supervisor of that actor. This means that the parent actor is responsible for monitoring the behavior of the child actor and handling any errors or failures that may occur. If a child actor encounters an error or fails to perform its task, the parent actor can decide how to handle the situation, which may include restarting the child actor, terminating it, or taking other corrective actions. This parent-child actor hierarchy enables Akka to create fault-tolerant systems that can recover from errors and failures quickly and reliably. By creating a hierarchy of actors, developers can organize and control the behavior of an entire service or application, making it easier to implement complex business logic and manage the flow of data and messages between actors.

For example, in the event that it becomes necessary to process a "send" request from a client and add additional checks before writing to a database, the following procedures can be performed for this purpose:

1. Creation of a parent actor as well as an actor for the request processing moment.
2. Switching the actor to a validation method just before putting it into the database, for example, if the form of a telephone "regular expression" needs to be corrected.
3. While the method is being tested, an actor can be waiting for a response to be checked, and with another child actor created, a method can be simultaneously composed to check the name while waiting.
4. Depending on the result of the two methods carried by the child actor, the parent actor will decide whether to write the data to a database or issue an error due to an invalid validation case.

The hierarchy of actors, extending from child form to parent form, is shown in Figure 2.

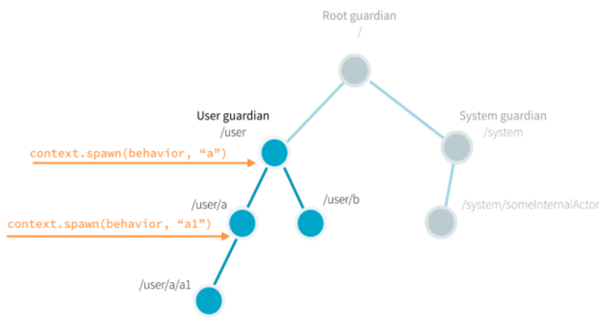
Root guardian
/
User guardian
/user
System guardian
/system
context.spawn(behavior, "a")
/user/a
/user/b
/system/someInternalActor
context.spawn(behavior, "a1")
/user/a/a1

Figure 2. The Akka Actor hierarchy model in the "some Internal actor" branch example
Source: reference [14]

The distinguishing feature of the process described above is that the original Akkaactors system already has two actors created in the Actor System block for inline management of actors within the system [14]. Interacting with many actors and creating a goal for the final state of the actor form a certain status received from all industry relations, after which the actors must be stopped. Their interaction, in which the parent actor stopped a particular child actor, also stopped, so the correct solution after the completion of the request processing is to stop the actor with the Actor Context using the special "stop" command. It is also possible to manage the process of the internal actor library, since the main strength of the actor model is the processing of the received results, especially in cases of fault tolerance:

```
def childActor(size: Long): ActorBehavior[String] =
ActorBehaviors.receiveMessage(message
=>childActor(size + message.length))
```

The above method shows an iterative function for handling a message received by the main actor class named Actor Behaviors. Below is an example of fail handling if an error occurs while receiving a message from one of the child actors:

```
def parentActor: ActorBehavior[String] = {
ActorBehaviors.supervise[String] {
ActorBehaviors.setup{result =>
val actorChildExample1 = result.spawn(child(0), "child1")
val actorChildExample2 = result.spawn(child(0), "child2")
ActorBehaviors.receiveMessage[String] {actorMessage =>
// message handling that might throw an exception
val parts = actorMessage.split(" ")
child1! parts (0)
child2! parts (1)
Behaviors.same
}}
}
onFailure(SupervisorStrategy.restart) // main handling of
error occur in case of onFailure strategy
}
```

Akka Actors are designed not only to handle errors and method results in this way, but can also perform a full set of tasks from a client-side request to receiving a response from the database server. This flexible model allows adjusting such a high load on enterprise applications depending on the stated business requirements regarding fraud prevention and control. For this case, it is assumed that messages sent from one actor to another must be immutable [18]. The flexibility of the actor model allows for the adjustment of the high load on enterprise applications, based on the specific business requirements related to fraud prevention and control. This is achieved by allowing messages to be sent from one actor to another in a flexible and customizable way.

In the actor model, actors communicate with each other by exchanging messages, which can contain data, instructions, or requests. This communication can be tailored to meet specific business needs, such as fraud prevention and control. For example, actors can be designed to monitor and detect suspicious activities, and to alert other actors or systems when necessary. The flexibility of the actor model also enables the implementation of complex workflows and business logic, which can help to prevent fraud and improve control over the system. For instance, actors can be used to enforce authorization and authentication rules, to validate data inputs, and to route messages between different parts of the system.

The functionality of an actor is assumed as an object of a computing system that can respond to an incoming message:

1. Send a fixed number of messages on demand to other actors.
2. Create, at the request of the task of counting new actors, in this case, of "child actors".
3. Develop rules for processing the next received message.

Because of this relation of actors to each other, they (actors) are created asynchronously, while the actor that sent the message does not wait for the second actor, but continues its operation through the to-do list. Only the second actor can exchange messages with exactly those actors that sent messages to it [7]. The actor model strategy written compared to the Apache web server, where the second Apache server can handle about 4000 sessions, while the Akka actor model strategy written by Yaws supports processing performance of 80000 sessions [13]. Statistics show that this strategy-based model, with real modification and a well-planned architecture, is 20 times more fault tolerant than Apache. This case shows the best aspects of actor models in the practice of performing operations with a huge number of requests and volumes of data.

Considering actor models with a messaging strategy, it is safe to say that the fault tolerance of actor-based models does not have problems with shared-memory computing systems, which cause additional responsibilities for managing threads and creating processes internally by hand, which also will impact debugging and troubleshooting, deadlocks, as well as low scalability potential. Below is a model of the API microservice system actor and the main service. Using library dependencies, it is necessary to create an Akka HTTP request maker by running the following code snippet where there are two API requests for post students and getting the URL (Uniform Resource Locator) of all objects [5]:

```
ddos-http-protection-api:
lazy valuserRoutes: Route = {
pathPrefix("entites") {
path("create-entity") {
//      authenticateOAuth2Async("user",
oauthAuthenticator) { user: UserContext =>
headersMap{ headers =>
post {
entity(as[SomeEntity]) { SomeEntity =>
log.debug(s"CREATE Rest for test further
$kaznuPhdStudent test REST API DDOS by HEAVY GET")
handleRequest(ddosPropsMaker,
DdosActor.CreateKaznuPhdStudent(Some(kaznuPhdStude
nt), headers, None))
}
}
//      }
}
} ~
path("user-all") {
//      authenticateOAuth2Async("user",
oauthAuthenticator) { user: UserContext =>
headersMap{ headers =>
log.debug(s"{| DDOS heavy scan endpoint | HTTP FLOOD
TYPED}")
get {
handleRequest(ddosPropsMaker,
DdosActor.GetAllEntities(None, headers, None))
}}}}
}
```

After the actor model of the message descriptor is prepared in the main part of the microservices, depending on the results, a decision is made that will allow adjusting the preservation of the HTTP Flood by the actors with each request:

```
ddos -http-protection-core:
in main method:
def receive = {
case message: GetAllEntities=> {
pipe {
ddosRepo.getAllEntitesMethod(Some(List(message.getAll
Entites.getOrElse(""))))
} to self
context.become(waitingResponse)
}
case message: CreateEntites=> {
pipe {
ddosRepo.createEntities(message.ddosHttpFloodEntity.get)
} to self
context.become(waitingResponse)
}}

consequently, next order operation:
def waitingResponse: Receive = {
case akka.actor.Status.Success(_) =>
log.debug(s"has been stored in repo. Finish request")
context.parent ! Accepted()

case result: ErrorInfo=>
log.debug(s"5 ERROR RESULT ${result.toString}")
context.parent ! result
}
```

The "waiting Response" method receives the response from the main request processing methods and handles the last step of the logical decision, including error handling and sending the response to the "ddos-http-protection-api" microservice in the parent actor hierarchy. Actors function for some time after they are created and are stopped according to the user's request. At the same time, at the moment when the parent actor stops, its descendants also stop synchronously. This situation actively contributes to the acceleration of resource cleanup, and also enables preventing resource leaks, in particular, this applies to leaks caused by open files and sockets [5]. In real conditions, when there is a need to work with low-level multi-threaded code, as a rule, the complexity associated with the need to manage the life cycles of various parallel resources that are closely related to each other is not taken into account.

To stop an actor, return Behaviors stopped inside it. In this case, as a rule, a response comes in the given form to a user message indicating the actor's stop when it has finally finished executing. It is technically possible to stop a child actor, which requires calling context.stop(childRef) from the parent actor. However, arbitrary actors (not parent ones) cannot be stopped in this way [5]. The Akka actor API provides a number of lifecycle signals. In particular, the PostStop dispatch takes place immediately after the actor has been stopped.

```
import akka.actor.{Actor, ActorSystem, Props}
class ResourceActor extends Actor {
  // Initialize the resource when the actor is created
  val resource: MyResource = initializeResource()
  override def receive: Receive = {
    case SomeMessage =>
      // Use the resource to process the message
      val result = processMessage(resource, SomeMessage)
      // Send the result back to the sender
      sender() ! result
  }
  override def postStop(): Unit = {
    // Release the resource when the actor is stopped
    releaseResource(resource)
  }
  private def initializeResource(): MyResource = {
    // Code to initialize the resource
    // ...
    new MyResource
  }
  private def releaseResource(resource: MyResource): Unit
= {
    // Code to release the resource
    // ...
  }

  private def processMessage(resource: MyResource,
message: SomeMessage): SomeResult = {
    // Code to use the resource to process the message
    // ...
    new SomeResult
  }
}
object ResourceActor {
  def props(): Props = Props(new ResourceActor)
}
object MyApp extends App {
  // Create the actor system
  val system = ActorSystem("MyApp")
  // Create the parent actor
  val parentActor = system.actorOf(Props[ParentActor],
"parentActor")
  // Send a message to the parent actor
  parentActor ! SomeMessage
  // Stop the actor system when done
  system.terminate()
}
```

The ResourceActor is responsible for managing a resource (represented here by the MyResource class). The initializeResource() method is called when the actor is created, and initializes the resource. The releaseResource() method is called in the postStop() hook, which is called when the actor is stopped, and releases the resource. The processMessage() method is called when the actor receives a message, and uses the resource to process the message. In this example, the message is of type SomeMessage, and the result is of type SomeResult. The ResourceActor is created by the ParentActor, which is not shown in this example. When the parent actor is stopped, it in turn stops its child actors, including the ResourceActor.

After that, the message is no longer processed. When a parent actor stops, it, in turn, stops its own child actor, after which it stops itself. This sequence of operations must be strictly adhered to, all PostStop signals of child elements should be processed up to the processing of the PostStop signal of the parent actor. Figure 3 provides a performance comparison of web servers that clearly illustrates the effectiveness of the practical application of the Akka Actors model in high-load applications.

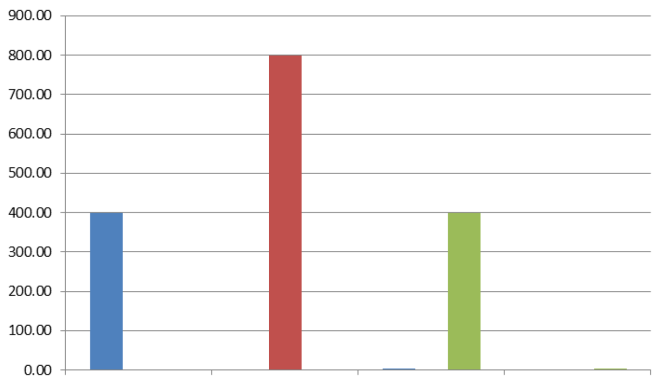

Figure 3. Comparative analysis of web server performance
Source: Compiled by the authors based on reference [13]

The information presented in Figure 3 is a clear illustration of the practical advantages of actor models in high-load applications. The colors indicate the amount of performance of web servers:

– red – Yaws (a web server created in the Erland language using the Akka Actors model in question);

– green – Apache (local disk);

– blue – Apache HTTP Server.

As follows from the data presented in Figure 3, the throughput of the Yaws server, in which the considered Akka Actors model was used, is 800 Kb/s, or up to 80000 insecure processes [19]. At the same time, the Apache server fails at about 4000 processes. Thus, the presented statistics clearly demonstrate that the fault tolerance of the model based on the Akka strategy is 20 times higher than that of Apache HTTP Server.

4. DISCUSSION

In a joint scientific paper aimed at studying the principles of building a new paradigm of social distributed computing, the team of researchers represented by Garcia-Valls et al. [20] touched upon the problems of seamlessly integrating computing into physical networks. The authors draw attention to the fact that the rational use of the Scala programming language can significantly improve the quality of asynchronous request processing in high-load systems. According to the researchers, this is essential when building social distributed computing systems, in which fog infrastructures play a dominant role, ensuring that the user of the systems maintains a high level of mobility. The conclusions of the researchers are fundamentally consistent with the results obtained in this scientific paper, demonstrating their practical significance for modeling processes in cyber-physical systems.

For their part, in a collaborative scientific study of the foundations of building a unified structure to improve interaction between languages and programming models of high-performance computing, Pineiro and Pichel [21] note that differences in software stacks are one of the most significant problems on the way to convergence of high-performance computing. According to scientists, any language or any programming model should tell the computer or computing system the actions that they should perform. For this reason, when building a specific programming model, it is necessary to clearly understand the behavior of individual actors and be able to determine the language that is used to display specific examples of these actors. This is extremely important for the effective construction of the Akka Actors model in order to counteract DDoS attacks on the server. The results obtained by the researchers are fundamentally consistent with the results of this scientific paper.

At the same time, the team of researchers represented by Pineiro et al. [22] jointly considered the key principles of the Ignis platform that is designed to work with large amounts of data and have the ability to run applications based on various programming languages. The study of scientists notes that the use of various massively parallel architectures determines the presence in them of several types of key elements, among which sequential processes, data parameter transformation functions, and actors should be distinguished. At the same time, the creation of applications involves the use of programming languages that are supported by the existing data structure and most of all meet the tasks set for developers. The Scala programming language in this case should be considered a priority when building the Akka Actors model. The conclusions of the researchers contribute to the expansion of ideas about the creation and practical use of massively parallel architectures, without contradicting the results obtained in this scientific paper.

In a collaborative study of split integration and coordination using a self-organizing coordinate area pattern, the research team represented by Pianini et al. [23] note that the so-called design patterns are often applied directly in software development, preserving knowledge of the most common problems. According to scientists, one of these problems is HTTP Flood attacks on servers (databases). The problem can be effectively solved by developing microservice systems with strict API rules, followed by the implementation of a distributed system strategy, which will allow interaction with a large number of external users. The findings are fundamentally consistent with the results of this scientific paper. In a scientific paper aimed at studying the sequence of applying factoring to improve the quality of the source code as one of the most important stages in the evolution of software, Tesone et al. [24] note that "live" programming provides the ability to create quality software at a faster pace than it takes place during the processes of its editing, compiling and debugging. At the same time, the concept of a sequential construction of a software process implies a clear execution of operations, with the ability to process service requests with a high load on the server and scaling the needs for services. This greatly contributes to the prevention of flood attacks on the server. The results obtained by scientists expand the understanding of the possibilities of service-oriented asynchronous programming without conflicting with the results of this research paper.

In a joint scientific study of attack detection mechanisms in networks with low power consumption, Ankam and Reddy [25] draw attention to the fact that flooding should be considered not only as one of the options for transmitting significant amounts of information, but also as a variant of a DDoS attack, capable of completely paralyzing the functioning of the server. A server failure results in the inability to process subsequent user requests, which negatively affects the operation of an application. This necessitates the development and implementation of special measures to prevent DDoS attacks on the server and counter them. The conclusions of the scientists are fundamentally consistent with the results obtained in this research paper. In a joint scientific paper aimed at studying the prospects for the development of distributed network algorithms, Castafieda et al. [26] note that combinatorial topology is of great importance for improving the efficiency of the analysis of distributed algorithms of high fault tolerance, which are used in applications with large memory and high load. According to the authors, flood attacks on applications with a high load can destabilize their work for a long time, while the introduction of microservice systems with built-in API rules and a strategy for using a distributed system with a large number of external users is the best solution in terms of the effectiveness of countering attacks of this kind. The conclusions of the researchers fully coincide with the results obtained in this scientific paper.

Niknejad et al. [27] conducted a joint study of the general principles of building a service-oriented architecture and came to the conclusion that this kind of architecture, being a separate architectural approach, improves the performance of typical systems while maintaining their key functions. At the same time, service-oriented programming using software models to counter flood attacks on servers in high-load applications is a method of legal action in such situations. The conclusions of the researchers are fully consistent with the results obtained in this scientific paper. In a scientific study of aspect-oriented programming based on semantics for the composition of context-sensitive web servers, the research team represented by Li et al. [28] concluded that a change in context forms a certain composition, in which the context itself is organically woven into the server architecture. According to the authors, any change in the context allows changing the architecture of servers, which is essential when the risk of flood attacks on the server increases. The conclusions of the authors expand the results of this research paper within the context of assessing the value of various changes in the service architecture when it becomes necessary to prevent attacks on the server.

Thus, discussing the results obtained in this research paper within the context of their comparison with the results obtained by other scientists who have investigated various aspects of the use of server-side DDoS flood processing methods, as well as the development and implementation of models for handling attacks in applications with a high load demonstrated their fundamental coincidence in the main aspects. This indicates a high level of reliability of the results of this scientific research and the possibility of their practical use for the development of systems with a large number of active users in order to increase the effectiveness of counteracting network attacks.

5. CONCLUSIONS

Understanding of the key concepts of handling heavy API requests at a high frequency allows to consistently develop ways to handle these attacks from network traffic anomalies for architectural solutions with actor models based on the Scala programming language. That being said, the main challenge should not be learning Scala or Akka programming so much as understanding the key principles of handling such complex HTTP-based attacks and gaining the ability to implement a behavioral actor while processing such heavy requests at the same time. In the course of this scientific research, it has been proven that messaging entities model a more fault-tolerant model than shared memory application systems in the case of request handles. Actors can precede more data, and the post-fall error handling policy is less painful. A clear messaging strategy from one actor to another can share the high load of "receiving" heavy HTTP Flood requests that are aimed at solving the problem of building an attack infrastructure as a network traffic and database side.

The practical application of the Akka model provides for the management of actors in hierarchies, were parent actors control child actors and handle exceptions. Applying the acquired knowledge in real situations allows simulating the communication necessary to obtain information from users. In addition, the achievement of a high level of control of actors in individual groups is ensured. Taken together, using the Akka Actors model to handle an HTTP Flood attack in high-load applications is an objectively justified solution in terms of optimizing asynchronous type service-oriented programming techniques to counter DDoS attacks. Prospects for subsequent scientific research in the direction stated in the subject of this scientific paper include searching for additional opportunities for the practical application of service-oriented asynchronous programming methods to counter DDoS attacks within the context of the development of the digital infrastructure of the Republic of Kazakhstan.

REFERENCES

[1] Radavičius, T., Tvaronavičienė, M. (2022). Digitalisation, knowledge management and technology transfer impact on organisations' circularity capabilities. Insights into Regional Development, 4(3): 76-95. http://doi.org/10.9770/IRD.2022.4.3(5)

[2] Kovács, A.M. (2022). Ransomware: A comprehensive study of the exponentially increasing cybersecurity threat. Insights into Regional Development, 4(2): 96-104. https://doi.org/10.9770/IRD.2022.4.2(8)

[3] Fakunle, S.O., Ajani, B.K. (2021). Peculiarities of ICT adoption in Nigeria. Insights into Regional Development, 3(4): 51-61. http://doi.org/10.9770/IRD.2021.3.4(4)

[4] Lavrov, E., Pasko, N., Siryk, O., Burov, O., Natalia, M. (2020). Mathematical models for reducing functional networks to ensure the reliability and cybersecurity of ergatic control systems. Proceedings - 15th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering, TCSET 2020: 179-184. Lviv-Slavske: Institute of Electrical and Electronics Engineers. http://doi.org/10.1109/TCSET49122.2020.235418

[5] Order of the Government of the Republic of Kazakhstan No. 827 "On approval of the State program Digital Kazakhstan". (2017). https://adilet.zan.kz/rus/docs/P1700000827, accessed on Oct. 14, 2022.

[6] Aizstrauts, A., Ginters, E., Lauberte, I., Eroles, M.A.P. (2013). Multi-level architecture on web services based policy domain use cases simulator. Lecture Notes in Business Information Processing, 153: 130-145. http://doi.org/10.1007/978-3-642-41638-5_9

[7] Agha, G.A. (1985). Actors: A model of concurrent computation in distributed systems. Cambridge, Massachusetts Institute of Technology.

[8] Lavrov, E.A., Paderno, P.I., Volosiuk, A.A., Pasko, N.B., Kyzenko, V.I. (2019). Automation of functional reliability evaluation for critical human-machine control systems. Proceedings of 2019 3rd International Conference on Control in Technical Systems, CTS 2019: 144-147. St. Petersburg: Institute of Electrical and Electronics Engineers. http://doi.org/10.1109/CTS48763.2019.8973294

[9] Barlybayev, A., Sabyrov, T., Sharipbay, A., Omarbekova, A. (2017). Data base processing programs with using extended base semantic hypergraph. Advances in Intelligent Systems and Computing, 569: 28-37. http://doi.org/10.1007/978-3-319-56535-4_3

[10] Weiser, M. (2021). Ubiquitous computing. Computer, 26(10): 71-72.

[11] Yoshioka, H., Yoshioka, Y., Tanaka, T., Hashigichi, A. (2022). Stochastic optimization of a mixed moving average process for controlling non-Markovian streamflow environments. Applied Mathematical Modelling, 116: 490-509. https://doi.org/10.1016/j.apm.2022.11.009

[12] Bianchini, R., Dagnino, F. (2023). Asynchronous global types in co-logic programming. Science of Computer Programming, 225: Article ID: 102895. https://doi.org/10.1007/978-3-030-78142-2_9

[13] Armstrong, J. (2003). Concurrency oriented programming in Erlang. Stockholm, Swedish Institute of Computer Science.

[14] Akka Documentation. (2020). https://doc.akka.io/docs/akka/current/typed/guide/tutorial_1.html, accessed on Oct. 14, 2022.

[15] Li, M. (2003). Decision analysis of statistically detecting distributed denial-of-service flooding attacks. II International Journal of Information Technology and Decision Making, 2(3): 397-405.

[16] Cabrera, J.B.D. (2001). Proactive detection of distributed denial of service attacks using mib traffic variables – a feasibility study. II Proceedings of International Symposium on Integrated Network Management, Piscataway, pp. 609-622.

[17] Wang, H., Zhang, D., Shin, K.G. (2004). Detecting SYN flooding attacks. Proceedings of Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies Piscataway, pp. 1530-1539.

[18] Mozer. M.C. (2005). Lessons from an Adaptive Akka. New York, Wiley.

[19] Technology set, Akka in examples. (2020). http://www.sics.se/~joe/apachevsyaws. accessed on Oct. 14, 2022

[20] Garcia-Valls, M., Dubey, A., Botti, V. (2020). Introducing the new paradigm of social dispersed computing: Applications, technologies and challenges. Journal of Systems Architecture, 91: 83-102.

https://doi.org/10.1016/j.sysarc.2018.05.007

[21] Pineiro, C., Pichel, J.C. (2022). A unified framework to improve the interoperability between HPC and Big Data languages and programming models. Future Generation Computer Systems, 134: 123-139. https://doi.org/10.1016/j.future.2022.04.002

[22] Pineiro, C., Martinez-Castano, R., Pichel, J.C. (2020). Ignis: An efficient and scalable multi-language Big Data framework. Future Generation Computer Systems, 105: 705-716. https://doi.org/10.1016/j.future.2019.12.052

[23] Pianini, D., Casadei, R., Viroli, M., Natali, A. (2021). Partitioned integration and coordination via the self-organising coordination regions pattern. Future Generation Computer Systems, 114: 44-68. https://doi.org/10.1016/j.future.2020.07.032

[24] Tesone, P., Polito, G., Fabresse, L., Bouraqadi, N., Ducasse, S. (2020). Preserving instance state during refactorings in live environments. Future Generation Computer Systems, 110: 1-17. https://doi.org/10.1016/j.future.2020.04.010

[25] Ankam, S., Reddy, D.N.S. (2023). A mechanism to detecting flooding attacks in quantum enabled cloud-based lowpower and lossy networks. Theoretical Computer Science, 941: 29-38. https://doi.org/10.1016/j.tcs.2022.08.018

[26] Castafieda, A., Fraigniaud, P., Paz, A., Raisbaum, S., Roy, M., Travers, C. (2021). A topological perspective on distributed network algorithms. Theoretical Computer Science, 849: 121-137. https://doi.org/10.1016/j.tcs.2020.10.012

[27] Niknejad, N., Ismail, W., Gham, I., Nazari, B., Bahari, M., Hussin, A.R.B.C. (2020). Understanding Service-Oriented Architecture (SOA): A systematic literature review and directions for further investigation. Information Systems, 91: 101491. https://doi.org/10.1016/j.is.2020.101491

[28] Li, L., Liu, D., Bouguettaya, A. (2021). Semantic based aspect-oriented programming for context-aware Web service composition. Information Systems, 36(3): 551-564. https://doi.org/10.1016/j.is.2010.06.003