Request Replication for FaaS Fault Tolerance

Yasmina Bouizem, Djawida Dib, Nikos Parlavantzas, Christine Morin, Fedoua Lahfa

Request Replication for FaaS Fault Tolerance

Yasmina Bouizem , Djawida Dib , Nikos Parlavantzas , Christine Morin , Fedoua Lahfa

RESEARCH REPORT

N° 9444

January 2022

Project-Team MYRIADS

ISSN 0249-6399 ISRN INRIA/RR--9444--FR+ENG

Request Replication for FaaS Fault Tolerance

Yasmina Bouizem *, Djawida Dib †, Nikos Parlavantzas ‡,

Christine Morin §, Fedoua Lahfa †

Project-Team MYRIADS

Research Report n° 9444 — January 2022 — 19 pages

Abstract: Function-as-a-Service (FaaS) is a popular programming model for building serverless applications, supported by all major cloud providers and many open-source software frameworks. One of the main challenges for FaaS providers is providing fault-tolerance for the deployed applications. The basic fault-tolerance mechanism in current FaaS platforms is automatically retrying function invocations. Although the retry mechanism is well suited for transient faults, it incurs delays in recovering from other types of faults, such as node crashes. This paper proposes the integration of a Request Replication mechanism in FaaS platforms and describes how this integration was implemented in a well-known, open-source platform. The paper provides a detailed experimental comparison of the proposed mechanism with the retry mechanism and an Active-Standby mechanism under different failure scenarios.

Key-words: Fault tolerance, FaaS, availability, serverless

* Y.Bouizem is with Tlemcen University, LRIT, Univ Rennes, Inria
† D. Dib and F.Lahfa are with Tlemcen University
‡ N.Parlavantzas is with INSA Rennes
§ C. Morin is with Inria

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Réplication de requêtes pour la tolérance aux pannes de FaaS

Résumé : Le Function-as-a-Service (FaaS) est un modèle de programmation populaire pour la création d'applications sans serveur, pris en charge par tous les principaux fournisseurs de cloud et de nombreux frameworks logiciels open source. L'un des principaux défis pour les fournisseurs de FaaS est de fournir une tolérance aux pannes pour les applications déployées. Le mécanisme de base de tolérance aux pannes des plates-formes FaaS actuelles réessaie automatiquement les appels de fonction. Bien que le mécanisme de nouvelle tentative soit bien adapté aux pannes transitoires, il entraîne des retards dans la récupération d'autres types de pannes, telles que les pannes de nœuds. Cet article propose l'intégration d'un mécanisme de réplication de requêtes dans les plates-formes FaaS et décrit comment cette intégration a été implémentée dans une plate-forme open source bien connue. L'article fournit une comparaison expérimentale détaillée du mécanisme proposé avec le mécanisme de nouvelle tentative et un mécanisme Active-Standby sous différents scénarios de panne.

Mots-clés : Tolérance aux pannes, FaaS, disponibilité, serverless

1 Introduction

Serverless computing is a new computing paradigm for developing distributed cloud-based systems [1, 2]. This paradigm is principally supported by Function-as-a-Service (FaaS) platforms, which allow developers to write and deploy functions without being concerned with provisioning, configuring, and managing servers. Developers can thus concentrate on the logic and business value of their applications while the cloud provider takes full responsibility for managing the underlying infrastructure. Several FaaS platforms are commercially available, such as Amazon Lambda [3], Google Functions [4], and Azure functions [5], or distributed in open source, such as Fission [6], OpenFaaS [7], Kubeless [8], and OpenWhisk [9].

One of the main challenges for FaaS providers is ensuring high availability for the deployed functions. Indeed, high availability and built-in fault tolerance are touted as main features of commercial FaaS platforms (e.g., [3]). Most current FaaS platforms support a basic form of fault-tolerance through retrying function executions [4, 6, 10–13]. However, while the retry mechanism allows coping with network delays, it incurs delays in recovering from other kinds of failures such as node failures.

In the work described in the present paper [14], we propose to integrate an active replication approach in FaaS frameworks in order to make failures transparent to the applications. The proposed approach consists in replicating function call requests. We implemented it in Fission, a popular open-source FaaS framework and compared it with existing fault-tolerance approaches. This work significantly extends our previous work reported in [15] and in which we studied the integration of the Active-Standby fault-tolerance approach in FaaS platforms. In this paper we bring the following novel contributions:

- Study of the integration of an active replication fault-tolerance approach in a FaaS environment;
- Implementation of an active replication scheme in the Fission FaaS platform by introducing a Request Replication mechanism (Fission Request Replication);
- Comparative evaluation according to several metrics of Fission Vanilla (native retry mechanism), a new version of Fission Active-Standby (enhanced implementation of the fault-tolerance approach proposed in [15]), and Fission Request Replication, using a stateless computational application both in normal functioning and in various failure scenarios, including pod and node failures and network delays;
- Insights on how to select a fault-tolerance approach according to the application type and user requirements in terms of performance, resource consumption, and availability.

The remainder of the paper is organized as follows. In Section 2 we discuss related work. In Section 3, we present Fission, an example of an open-source FaaS environment, which we used for implementing and evaluating our proposed fault-tolerance approach. We describe two existing fault-tolerance approaches in Section 4, namely the retry mechanism natively implemented in Fission and the Active-Standby approach we studied in [15], and their implementation in Fission. We present in Section 5 the Request Replication approach in the context of FaaS platforms and its implementation in Fission. Section 6 is devoted to the experimental setup, and experimental evaluation results are analysed in Section 7. We unfold lessons learnt in Section 8 and conclude in Section 9.

2 Related Work

A wide range of mechanisms have been applied to support fault-tolerance in cloud systems [16]. We consider next only mechanisms related to serverless systems. The basic fault-tolerance mechanism in current commercial and open-source FaaS platforms is automatically retrying invocations. All major commercial platforms, such as AWS Lambda [10,11], Google Cloud Functions [4] and Microsoft Azure Functions [12], provide automatic retry functionality to handle failures and timeouts. For instance, AWS Lambda retries asynchronous invocations up to two times with a delay between such retries. Some open-source FaaS platforms also support the retry mechanism, including Fission and OpenFaaS, which retries asynchronous invocations with an exponential back-off [13]. Our work adds two fault-tolerance mechanisms to FaaS platforms beyond automatic retry.

Fault-tolerance in serverless systems can also be realised through using additional services provided by cloud platforms. For instance, using Azure load-balancing and event ingestion services, developers can deploy functions in different regions in an active-active or active-passive pattern, which provides protection against disaster scenarios [17]. Using serverless orchestration services (such as Google Workflows [18], AWS Step Functions [19], or Azure Durable Functions [20]), developers can define workflows that coordinate functions, automatically retry failed or timed-out invocations, and run custom code to handle different types of errors. For instance, using AWS Step Functions, developers can resume failed workflows from the state at which they failed [21]. Similar capabilities are provided by open-source orchestration frameworks, such as Apache OpenWhisk Composer [22] or Faas-flow for OpenFaaS [23]. Our work provides fault-tolerance mechanisms implemented within FaaS platforms without involving external services.

Recent research is investigating fault-tolerance for stateful serverless applications, composed of multiple functions and interacting with storage services. [24] proposes inserting a layer between commodity FaaS platforms and key-value stores to ensure atomic visibility of storage updates. The proposed system assures fault tolerance by enforcing the read atomic consistency guarantee. [25] describes a library and runtime for building transactional, fault-tolerant workflows on existing serverless platforms. The system supports transactions within and across functions through applying a log-based fault-tolerance approach. Our work focuses on ensuring fault-tolerance for individual, stateless functions, rather than for stateful function compositions.

3 Fission FaaS Framework

This section presents Fission as an example of an open-source FaaS platform, which will be used in all the experiments presented in this paper. The Fission [6] framework is dedicated to serverless computing and is built on Kubernetes [26], an open-source container orchestrator. The core Fission components are: Function Pods, Router, and Executor (see Figure 1). Function Pods contain function-specific containers to serve requests coming from users. The requests are also named function calls.

The Router receives a function call (message 1 in Figure 1) and forwards it to the corresponding function pod, if it exists (message 2.a in Figure 1). Otherwise, the Router requests a function pod from the Executor (message 2.b in Figure 1) and then forwards the function call to the provided pod (message 5 in Figure 1). The Executor provides the requested function pods in two different ways according to the used Executor type: PoolManager or NewDeploy. PoolManager maintains pools of warm generic containers and warm function containers in order to provide low cold start latencies [27] and start functions quickly (message 3.a in Figure 1). However, PoolManager doesn't allow selecting multiple pods per function, which limits its usefulness during high traffic. NewDeploy creates Kubernetes service to loadbalance the requests between

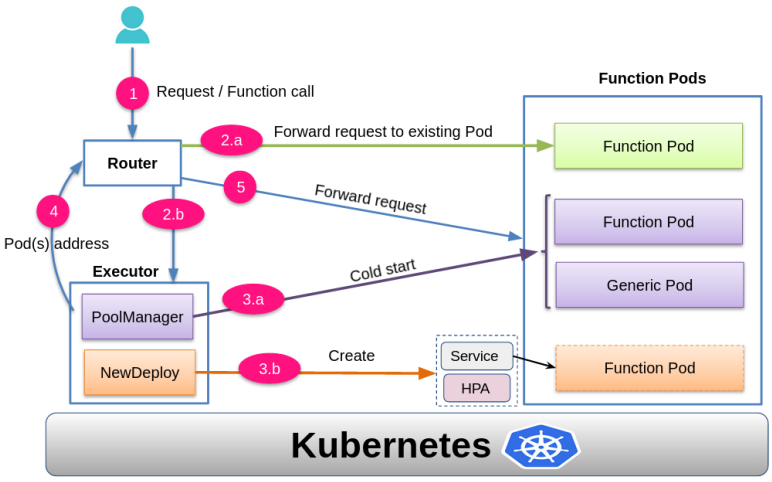

Figure 1: Simplified Fission Architecture

function pods and provides a Horizontal Pod Autoscaler to execute a function and adjust the number of pods to the traffic (message 3.b in Figure 1). The official version of Fission comes with a retry-based fault-tolerance mechanism that we describe in the next section.

4 Existing Fault-tolerance mechanisms in Fission

We present in this section two existing fault-tolerance mechanisms implemented in Fission: the native Retry mechanism implemented in most FaaS platforms including Fission (Section 4.1), and the Active-Standby approach, an enhanced version of the one proposed in our previous work [15] (Section 4.2).

4.1 Retry

The retry fault-tolerance mechanism is the native fault-tolerance mechanism in Fission and consists basically in restarting the entire submission process of a failed request. The retry mechanism used in Fission works as shown in Figure 2. When a function call is received, the Router forwards it to the corresponding function pod, as described in Section 3. If the function execution fails, the Router retries to forward again the function call until receiving a response from the function execution or reaching the maximum number of retries set by the administrator [28]. If all the retries fail or the received response is an error, Fission assumes that the function pod doesn't exist anymore. Thus, the Router asks the Executor for a new service for the function. Then, it retries to forward the function call to the new function service and so on until the request is served.

4.2 Active-Standby

In the context of FaaS, the Active-Standby mechanism consists in creating two function service instances. The first one is active and serves all requests during normal usage. The second one is passive (on standby). The two instances are connected by a heartbeat mechanism that continually checks their connectivity and status. If the heartbeat of one instance is not received

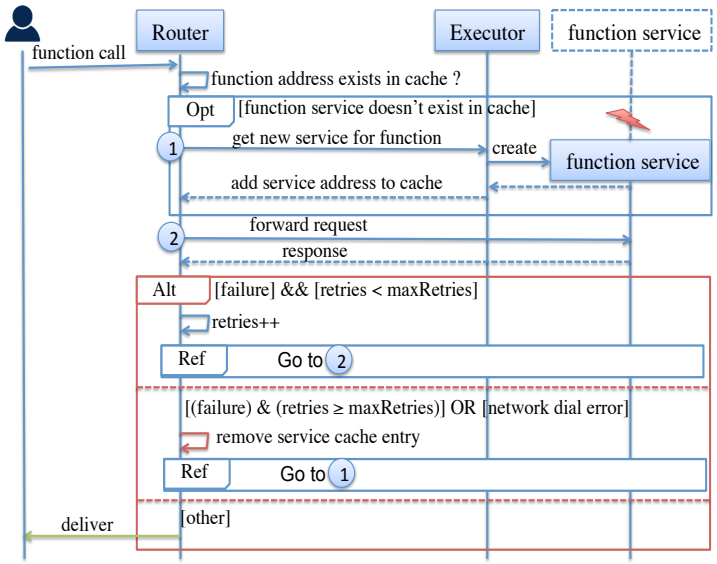

Figure 2: Fault-tolerance protocol with the Retry mechanism

within a configured amount of time, an action is triggered depending on the identity of the unreachable instance. If the passive instance is unreachable, another passive instance is created. If the active instance is unreachable, the standby instance is activated to serve incoming requests and another passive instance is created.

To implement the Active-Standby mechanism in Fission, we use the NewDeploy executor type as it supports creating replicas of function pods. In this approach, two function pods are created (ie,. active and passive) and both support the Kubernetes Readiness Probe [29] that indicates when the container is ready to receive requests. For instance, the active pod is marked in ready state and is therefore ready to receive and serve traffic. The passive pod is in standby and is marked in not-ready state, so no traffic is forwarded to it. We implemented a new router, called Router Active-Standby (Router AS), and used it instead of the default Fission Router (that's what makes this implementation differs from the one presented in [15]). The Router AS forwards all received function calls specifically to the active pod, as shown in Figure 3. The Active-Standby mechanism implemented in Fission works as shown in Figure 4. While the request is being processed, both active and standby pods send and receive heartbeats to and from each other for health checks. The heartbeats are performed each second (the minimum configurable value using Kubernetes Readiness probes). When the active pod is running, the passive pod fails the readiness probe and stays running in a not-ready state. If the active pod fails, the passive pod passes the readiness probe and becomes active. Then, another pod is created to replace the passive pod. The same action happens if the passive pod crashes for any reason.

5 Request Replication for FaaS

This section presents the Request Replication fault-tolerance mechanism and its implementation in Fission.

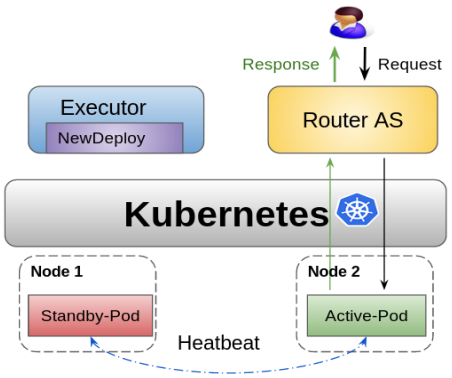

Figure 3: Overview of the Active-Standby mechanism in Fission

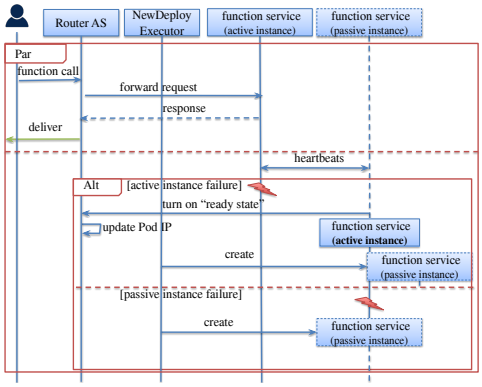

Figure 4: Fault-tolerance protocol with the Active-Standby mechanism

5.1 Request Replication Principle

Request Replication consists in having K replicas process a request at the same time. The number of replicas depends on the number of simultaneous failures to be tolerated.

The Request Replication (RR) solution is divided into two phases. First, the client sends a request, and the request is received and processed simultaneously by all replicas. Second, the first response produced by any replica is delivered to the client. The client can thus receive a response despite replica failures.

5.2 Implementation in Fission

To implement the RR approach in Fission, we used the NewDeploy Executor as it allows to create many pod replicas. We replaced the default Router with a new implemented one, called Router Request Replication (Router RR). This Router replicates each received request on all function pod replicas, in order to be processed in parallel. Then it sends the first received response to the user, as shown in Figure 5. To tolerate K failures using this approach, it is necessary to have a minimum of K+1 replicas, so that the Router can ensure that the user always receives a response. Figure 6 illustrates the implemented request replication in Fission.

6 Experimental Setup

In this section we describe the experimental setup for evaluating the effectiveness of the proposed RR fault-tolerance approach and comparing it with the retry mechanism and the AS approach in the context of their implementation in Fission.

6.1 Environment

We performed our experiments on the Grid'5000 [30] testbed, an experimental platform that supports research on all areas of computer science. We used 5 nodes on the Lyon site, each node having 2 CPUs Intel Xeon E5-2620 v4 with 8 cores/CPU and 64 GB memory, to deploy Kubernetes [31] (version 1.19). In our cluster we have one node for the Kubernetes master and we setup another node for Fission AS (Active-Standby), Fission RR (Request Replication) and the original version of Fission (vanilla). The three other nodes are workers, where the function

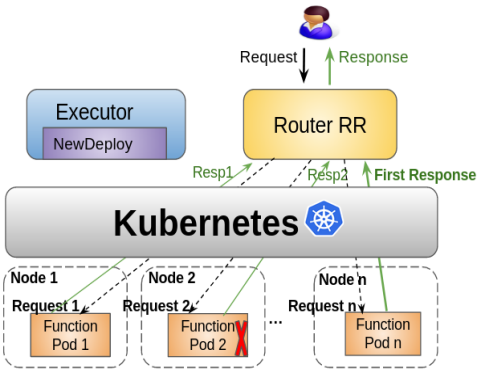

Figure 5: Overview of the request replication mechanism in Fission

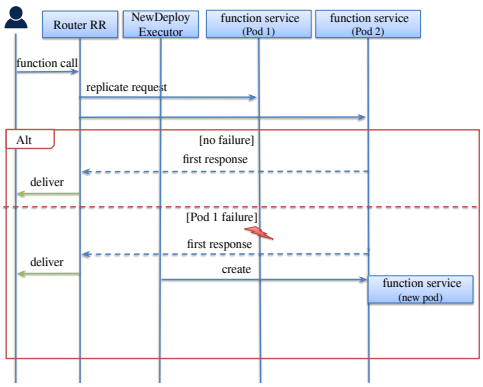

Figure 6: Fault-tolerance protocol with the Request Replication mechanism

pod is placed. For each experiment we use either RR, AS or vanilla with version 1.10.0 (the latest stable release at the time of their implementation). We setup 2 additional nodes; one is used as client in order to invoke functions and another one to inject faults.

6.2 Applications

We used a CPU-intensive HTTP-Triggered function that computes the Fibonacci sequence (a series of numbers where each number is the sum of the two preceding ones). Our function takes n=15 as an input, computing the 15th term of the sequence.

6.3 Workload

The workload is generated with Tsung [32], a high-performance benchmark framework. In our test, we generated 60000 requests during 10 minutes with 100 concurrent users created every second.

6.4 Failure Scenarios

We defined three sets of failure scenarios:

- Pod failure: where an application failure is due to a pod failure.
- Node failure: where an application failure is due to a node failure.
- Network delay: where we inject latency to see the impact of network delay on the deployed application.

In the pod failure and network delay scenarios, we use the Chaos Mesh tool [33] to inject faults to pods. In the first scenario, the failure is simulated by killing the function pod at the 5th minute from the beginning of the workload execution. In the second scenario with node failures, we use a script to crash nodes. The failure is simulated by killing the node hosting the function instance 5 minutes after the beginning of the workload execution. In the third scenario, we injected latency at the 5th minute and it lasts for 10 seconds. The injected latency values are 50ms, 100ms and 200ms. We note that the injected latency causes a delay for all responses

coming from the function pod. In the three scenarios, the failure is injected in the active pod for AS and in one of the two pods for RR. Each scenario has been repeated at least 5 times for each version of Fission (i.e,. vanilla, AS and RR). The averages of the measurements are shown in the illustrated results.

6.5 Metrics

We evaluate our solution using three categories of metrics:

- Performance: The performance is measured using throughput and response time values. The throughput is the number of requests served per second, and the response time is the time between a user request and the system response.
- Availability: The availability is measured using the recovery time, which is the time between the first reaction to failure and the time when the service is available again. We also capture the failed requests (those with HTTP 5xx response code) to calculate the error rate.
- Resource consumption: The resource consumption is measured as the amount of CPU and memory consumed by the 5 nodes during the execution of the workload.

7 Experimental Results

This section presents the results obtained from the experimental comparison of our proposed strategy RR with the existing approaches AS and retry.

7.1 Performance Results

7.1.1 Results with no failures

Figure 7 shows the response time for Fission AS, vanilla and Fission RR with no failures. In this figure we can notice that Fission RR is slightly faster than Fission AS and vanilla because once it receives the first response from one of the function replicas, it forwards it to the user. Vanilla is slightly slower. This maybe be explained by the use of the Kubernetes service to send the request to the pod belonging to the function, which adds another hop compared to AS and RR.

For the throughput, AS, vanilla and RR handle the same throughput with values around 100 request/sec (the expected throughput)

As a conclusion we can say that RR performs better than AS and vanilla in terms of response time when there are no failures.

7.1.2 Results with failures

1. Pod Failure Scenario

   Figures 8 and 9 illustrate the throughput and response time of AS, vanilla and RR with a pod failure. In Figure 8, we can observe a small degradation in the throughput of AS. This is because of the failover of the active pod to the standby pod. We also notice a degradation in the throughput of vanilla when the pod fails at 300s as there is no available pod to serve the requests. RR provides stable throughput despite the pod failure since all traffic is executed by the healthy replica.

   In Figure 9, we notice some spikes in the response time of vanilla during almost 30 seconds. This is attributed to that once the pod failure is detected, the router starts the retries.

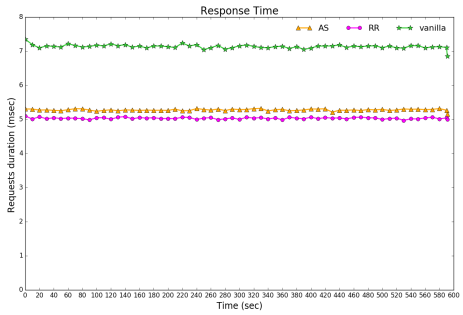

Figure 7: Response time of AS, vanilla and RR with no failure

When the function pod recovers, we see that the response time drops off at around 7ms. In contrast, RR and AS provide stable response times with values around 5ms.

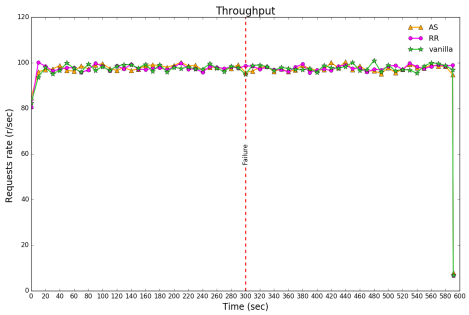

Figure 8: Throughput of AS, vanilla and RR with pod failure

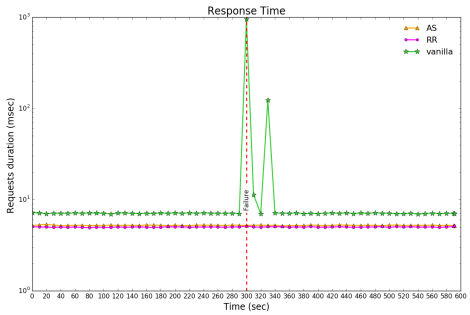

Figure 9: Response time of AS, vanilla and RR with pod failure

2. Node Failure Scenario

   Figures 10 and 11 present performance results of AS, vanilla and RR with a node failure. Figure 10 shows a degradation in the throughput with AS when the node hosting the active pod crashes. This is because of the required actions to switch the passive pod to active. In vanilla, the throughput drops when the function pod stops serving requests. The router then starts the retries and the requests are queued until a new pod starts running on a healthy node. This causes a spike in throughput that reaches 1300 requests/sec, and then drops back to a normal state. The throughput of RR remains constant because the failure is masked by the presence of the other replica that continues to process the user's requests.

   Figure 11 shows spikes in the latency of vanilla. This is because the router retries many requests, where the wait time is increased exponentially after every attempt. We assume that the response time of the queued requests is increased when the pod recovers. The response time of AS and RR is stable since the requests are served by the standby pod in AS and by the second replica in RR.

3. Network Delay Scenario

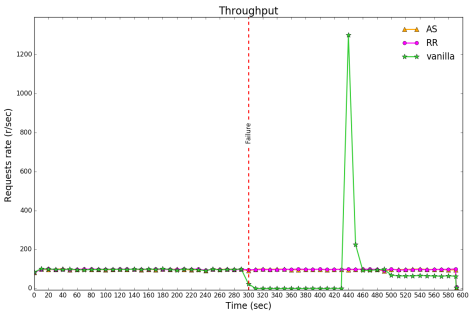

Figure 10: Throughput of AS, vanilla and RR with node failure

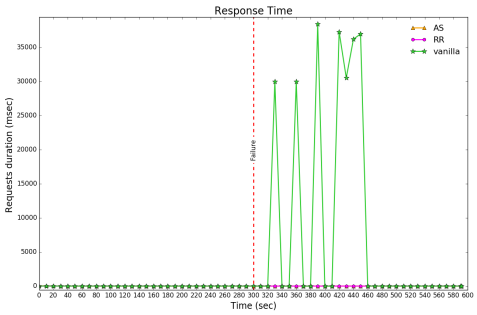

Figure 11: Response time of AS, vanilla and RR with node failure

We injected separately three latency values: 50ms, 100ms and 200ms. Figures 12, 13, and 14 show the response time of AS, vanilla and RR with the injected latency values. When we increase the value of latency, we can see a significant change in the response time of vanilla. For example, 200ms of latency doubles the response time of vanilla from 500ms to 1000ms (see Figures 13 and 14). The reason for this behaviour is that the router retries requests with exponential backoff, increasing the waiting time between retries which leads to performance degradation. Looking at the response time of AS, we notice a peak when the latency is added because the active pod responds too late.

In RR, we see no impact on the response time when we add latency on one of the replicas, because the delay of a single replica is masked by the fast response of the other replica.

7.2 Availability Results

7.2.1 Recovery time

The recovery time is the time required for a service to recover from failures and becomes available again. This covers the time between failure detection and the time when the service becomes fully operational. It is measured for the three approaches as follows: For vanilla, after the failure, the pod becomes unhealthy (see Figure 15). In reaction to that failure, the router retries the failed requests. When the maximum number of retries is reached, the pod is considered as failed and the service URL is deleted from the router cache. The service becomes available again when a new pod is created and added to the router cache.

For AS, the failure is detected by the heartbeat mechanism (see Figure 16). The reaction is the failover to the standby pod and the update of the router cache. Once the router cache is updated with the IP address of the active pod (Active-IP), the service becomes available.

For RR, no recovery is necessary as the failure of one of the replicas does not effect service availability (see Figure 17). The service remains available because the Pod2 serves the requests.

1. Pod Failure Scenario

   Table 1 presents the recovery time of AS, vanilla and RR with a pod failure. The measured recovery time for AS is significantly lower than for vanilla. The reason is that with AS, there is already a standby pod, and the service is recovered as soon as the standby detects the failure of the active pod. In contrast, recovery with vanilla depends on the repair of the failed pod. For RR, the second replica continues to serve requests. Therefore, for this

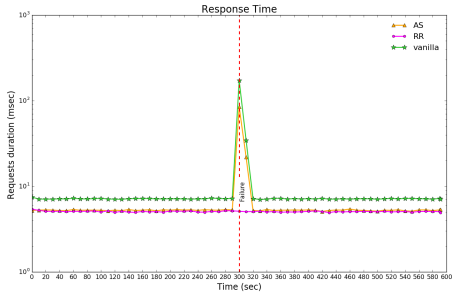

Figure 12: Response time with 50ms of latency

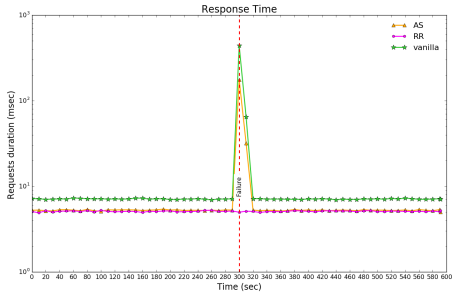

Figure 13: Response time with 100ms of latency

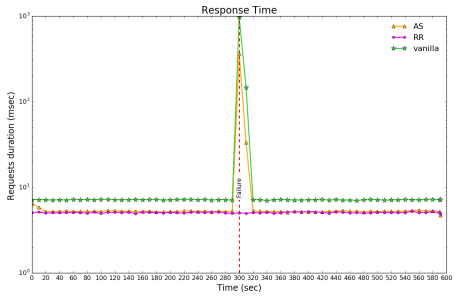

Figure 14: Response time with 200ms of latency

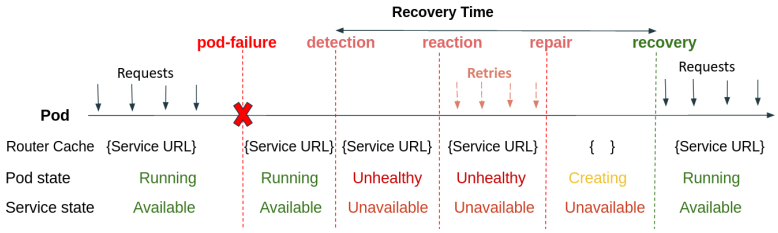

Figure 15: Recovery time in vanilla

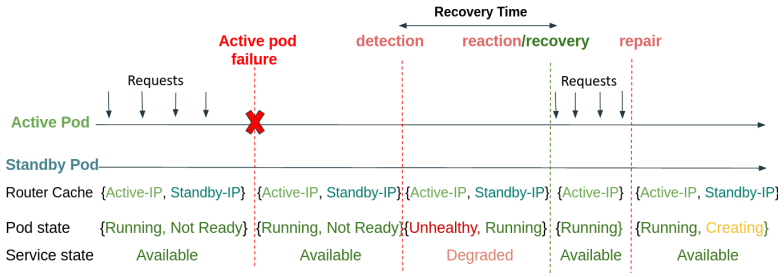

Figure 16: Recovery time in AS

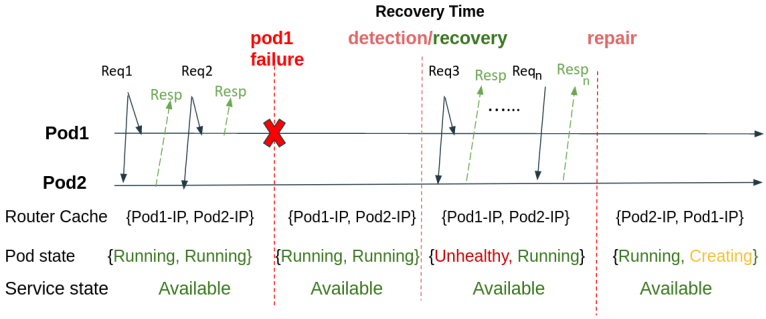

Figure 17: Recovery time in RR

approach, recovery time is zero.

Table 1: Recovery Time with AS, vanilla and RR in pod failure

Fission vanilla	Fission AS	Fission RR
7s	1.81s	0s

2. Node Failure Scenario

   Table 2 presents the recovery time of AS, vanilla and RR with node failure. The recovery time for vanilla is significantly higher than that for AS and RR. AS takes seconds to recover from a node crash, whereas vanilla takes more than 2 minutes. RR recovery time is 0 (no unavalability of the service).

Table 2: Recovery Time with AS, vanilla and RR in node failure

Fission vanilla	Fission AS	Fission RR
2m19s	2.80s	0s

7.2.2 Error rate

1. Pod Failure Scenario

To analyze the number of failed requests, we count the number of requests with a response of 5xx. Table 3 summarises the error rates of AS, vanilla, and RR with pod failures. Vanilla has a 0.01% error rate (i.e., some HTTP requests failed with code 502). The error rate for AS and RR is 0% (i.e., all requests succeeded with a returned code 200).

Table 3: Error rate of all requests for vanilla, AS and RR with pod failures

Fission vanilla	Fission AS	Fission RR
0.01%	0%	0%

2. Node Failure Scenario

   Table 4 shows the error rate of AS, vanilla, and RR with a node failure.

   In this scenario, errors occur with the requests due to the node crash in vanilla with an error rate of 1.26%. Once the requests are retried, some of them return a 502 status code. AS and RR have 0% of error rate, as both tolerate better a node crash by the presence of a replica, so all requests are served with success and return the code 200

Table 4: Error rate of all requests for vanilla, AS and RR with node failure

Fission vanilla	Fission AS	Fission RR
1.26%	0%	0%

7.3 Resource Consumption Analysis

We measure CPU and memory usage in order to analyse the amount of resources that are required to realize fault tolerance for each approach. Figures 18 and 19 show CPU and memory consumption, respectively, for AS, vanilla and RR without and with failures (pod and node failures). This is the overall CPU and memory usage of the 5 nodes hosting Kubernetes and the Fission platform during the execution of the workload.

In the three scenarios (i.e., no failure, pod failure, node failure), we observe that RR consumes more CPU and memory compared to vanilla and AS. In the case when there are no failures, for example, the overhead of using RR is 64% in CPU and 40% in memory consumption compared to vanilla. This is because of the additional resources allocated to the second replica. In vanilla, only one replica executes requests. AS has an overhead of 58% in CPU consumption and 31% in memory consumption compared to vanilla. Note that in AS, the standby replica is hot, which means that it is loaded in memory. The replica does not process requests (like the active replica of RR), but it does perform regular heartbeats, which consumes resources.

Figures 20, 21, and 22 show the average CPU consumption over time for the Kubernetes master node, the Fission node, and the 3 worker nodes for all approaches with a pod failure.

The CPU consumption of AS and RR are similar and vanilla shows the lowest CPU utilization. We notice that on average, the coordinator nodes (i.e., master and Fission nodes) need more resources compared to the worker nodes because the services that manage the cluster are located in the master and the services that manage the functions are located in the Fission node. Especially for AS and RR, their coordinator nodes are experiencing high CPU usage compared to vanilla. This is due to the CPU consumption of the Router in the Fission node when calling the Kubernetes API server to get updates on the IPs of the function pods.

When looking at the CPU consumption of worker 2 and worker 3 in AS (see Figure 21), we note that after the failure, the CPU usage of worker 3 starts to grow while the CPU usage of worker 2 goes down, which reflects the behavior of the failover to the standby pod.

In RR, when the pod1 fails, another one is created in the same node (worker2) and we observe a very short peak in the CPU at the 6th minute, as shown in Figure 22.

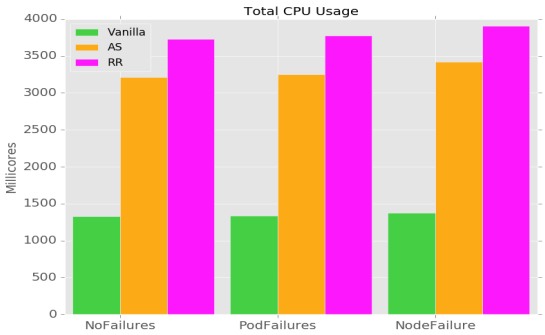

Figure 18: CPU consumption in AS, vanilla and RR without and with failures (pod and node failure)

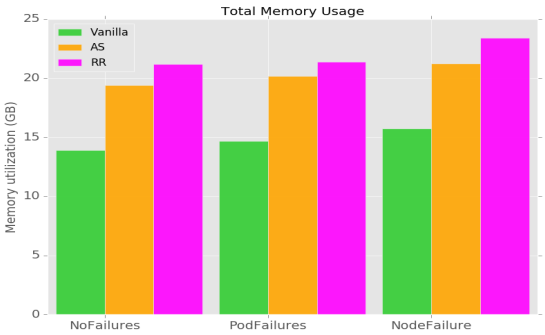

Figure 19: Memory consumption in AS, vanilla and RR without and with failures (pod and node failure)

8 Lessons Learned

From our experimental comparison of the three fault-tolerance mechanisms (i.e., Retry, Active-Standby and Request Replication), we note that each mechanism has different properties and is most effective under different conditions. The Retry mechanism is well suited for transient faults that last a short time. This approach consumes less resources and is thus more energy-efficient than the Active-Standby and the Request Replication mechanisms. The Active-Standby mechanism offers better availability for long-lasting faults, compared to the Retry mechanism, but at the cost of higher resource consumption. For instance, in our experiments, the Active-Standby mechanism consumes more than two times the CPU consumed by the Retry mechanism. The Request Replication mechanism offers the best availability for any fault duration. Indeed, when the fault does not affect all replicas, there is almost no impact on the overall availability. This mechanism also offers the best, and most stable performance. On the other hand, the mechanism incurs the highest resource consumption. In general, we observe that availability and resource consumption in the three approaches are inversely related.

In the presented experiments, the three approaches were tested with a stateless and idempotent application, where the same input always gives the same output. As future work, we plan to investigate the behavior of these approaches with other application types, such as stateful FaaS applications. With these applications, state is typically maintained in external storage services, such as NoSQL databases [25]. Using RR for such applications seems challenging. The reason is that concurrent access will increase the load on the storage service and introduce overhead for maintaining consistency. This may result in reduced performance in the case of normal, fault-free operation compared to using AS or Retry. Integrating caching into the stateful functions could mitigate this problem [34].

Given the trade-offs between the different fault-tolerance mechanisms, we believe that a FaaS platform should simultaneously support multiple mechanisms, such as Retry, AS and RR, and

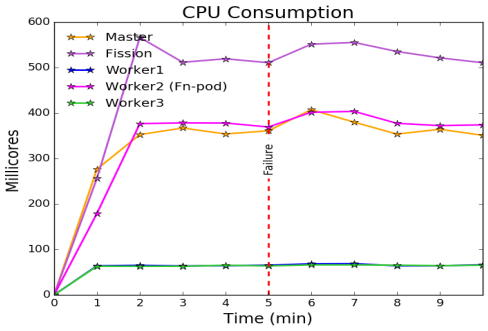

Figure 20: CPU consumption per node in vanilla with pod failure

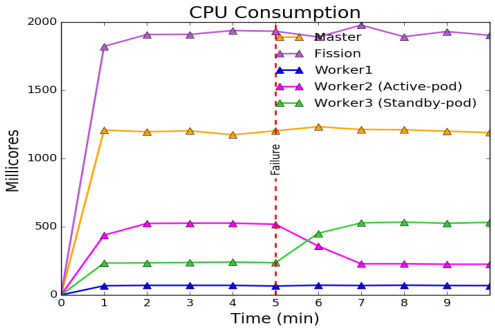

Figure 21: CPU consumption per node in AS with pod failure

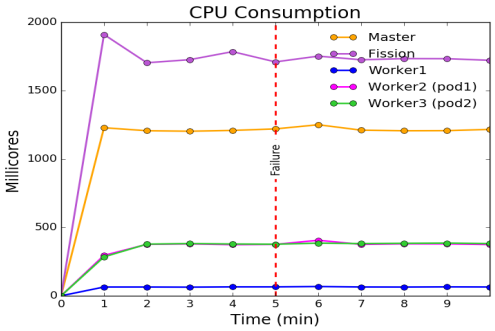

Figure 22: CPU consumption per node in RR with pod failure

use one or another according to specific factors. These factors may include performance, availability, and resource consumption requirements, application types, fault models, and operating conditions, such as network latencies.

9 Conclusion and future work

This paper proposed the integration of an active replication fault-tolerance approach (RR) in FaaS platforms. This RR approach was experimentally compared with a passive replication approach (AS) and the basic retry mechanism, in terms of different metrics, and under different failure scenarios.

The obtained results highlighted the differences among the three approaches. Notably, they showed that the retry mechanism is not sufficient for providing high availability. The reason is that the default behavior of retry results in significant recovery time in the case of node failures. The retry mechanism is better suited for transient failures as seen in the network delay scenario. With AS, recovery time is decreased because the service becomes available as soon as the standby replica detects the failure of the active replica. With RR, the service always remains available as long as another replica continues to respond to users and recovery does not depend on replacing the faulty replica.

In our future work, we plan to investigate adaptive techniques for fault tolerance in FaaS

environments. Applying such techniques will support automatically selecting the appropriate fault-tolerance mechanism based on the type of FaaS application (e.g., stateful or stateless), user requirements (e.g., performance, availability, resource consumption) and operating conditions (e.g., fault rates, network latencies).

Acknowledgement

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, 2019.

[2] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[3] "Aws lambda features," https://aws.amazon.com/lambda/features/, 2020, [Online; accessed 07-july-2021].

[4] "Retrying background functions," https://cloud.google.com/functions/docs/bestpractices/retries, 2019, [Online; accessed 07-july-2021].

[5] "Azure functions," https://azure.microsoft.com/fr-fr/services/functions/, 2020, [Online; accessed 07-july-2021].

[6] "Fission," https://docs.fission.io/docs/, 2019, [Online; accessed 07-july-2021].

[7] "Openfaas," https://www.openfaas.com, 2019, [Online; accessed 07-july-2021].

[8] "Kubeless," https://kubeless.io, 2021, [Online; accessed 07-july-2021].

[9] "Apache openwhisk," https://openwhisk.apache.org, [Online; accessed 07-july-2021].

[10] "Error handling and automatic retries in aws lambda," https://docs.aws.amazon.com/lambda/latest/dg/invocation-retries.html, 2020, [Online; accessed 07-july-2021].

[11] "Using aws serverless technology as an enabler for cloud adoption," https://aws.amazon.com/blogs/apn/using-aws-serverless-technology-as-an-enabler-for-cloud-adoption/, 2019, [Online; accessed 07-july-2021].

[12] "Retry pattern," https://docs.microsoft.com/en-us/azure/architecture/patterns/retry, 2020, [Online; accessed 07-july-2021].

[13] "Timeouts - asynchronous invocations," https://docs.openfaas.com/deployment/troubleshooting/#timeouts-asynchronous-invocations, 2019, [Online; accessed 07-july-2021].

[14] P. Felber and P. Narasimhan, "Experiences, strategies, and challenges in building fault-tolerant corba systems," *IEEE transactions on Computers*, vol. 53, no. 5, pp. 497–511, 2004.

[15] Y. Bouizem, D. Dib, N. Parlavantzas, and C. Morin, "Active-Standby for High-Availability in FaaS," in *Sixth International Workshop on Serverless Computing (WoSC6) 2020*, Delft, Netherlands, Dec. 2020. [Online]. Available: https://hal.inria.fr/hal-03043479

[16] M. A. Mukwevho and T. Celik, "Toward a smart cloud: A review of fault-tolerance methods in cloud systems," *IEEE Transactions on Services Computing*, vol. 14, no. 2, pp. 589–605, 2021.

[17] "Azure functions geo-disaster recovery," https://docs.microsoft.com/en-us/azure/azure-functions/functions-geo-disaster-recovery, 2020, [Online; accessed 07-july-2021].

[18] "Google cloud workflows," https://cloud.google.com/workflows, [Online; accessed 07-july-2021].

[19] "Aws step functions," https://aws.amazon.com/step-functions, [Online; accessed 07-july-2021].

[20] "Azure durable functions," https://docs.microsoft.com/en-us/azure/azure-functions/durable, [Online; accessed 07-july-2021].

[21] "Resume aws step functions from any state," https://aws.amazon.com/blogs/compute/resume-aws-step-functions-from-any-state/, [Online; accessed 07-july-2021].

[22] "Apache openwhisk composer," https://github.com/apache/openwhisk-composer, [Online; accessed 07-july-2021].

[23] "Faas-flow," https://github.com/s8sg/faas-flow, [Online; accessed 07-july-2021].

[24] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, and J. M. Faleiro, "A fault-tolerance shim for serverless computing," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387535

[25] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu, "Fault-tolerant and transactional stateful serverless workflows," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. Banff, Alberta: USENIX Association, November 2020. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/zhang-haoran

[26] K. Hightower, B. Burns, and J. Beda, *Kubernetes: up and running: dive into the future of infrastructure*. " O'Reilly Media, Inc.", 2017.

[27] M. Shilkov, "What is a cold start?" https://mikhail.io/serverless/coldstarts/define/, 2019, [Online; accessed 07-july-2021].

[28] "Fission router," https://godoc.org/github.com/fission/fission/pkg/router, 2020, [Online; accessed 07-july-2021].

[29] "Configure liveness, readiness and startup probes," https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/, [Online; accessed 07-july-2021].

[30] "Grid5000," https://www.grid5000.fr/w/Grid5000:Home, 2020, [Online; accessed 07-july-2021].

[31] "Kubernetes," https://kubernetes.io/, [Online; accessed 07-july-2021].

[32] "Tsung," http://tsung.erlang-projects.org/user_manual/, [Online; accessed 07-july-2021].

[33] "Chaos mesh," https://chaos-mesh.org/, [Online; accessed 07-july-2021].

[34] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful functions-as-a-service," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 2438–2452, Jul. 2020. [Online]. Available: https://doi.org/10.14778/3407790.3407836

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399