Reliable Session Handling in Distributed Environments - DIRE

Balint Bombitz

June 13, 2025

Abstract Session-based applications often struggle to maintain reliability and consistency in distributed systems especially in harsh, unstable environments. Even if Kubernetes is primarily designed for scaling, and thus, for robustness, if an application does not support mid-session error handling and redundancy, can cause serious errors in critical applications. This paper aims to introduce the reader to DIRE (Distributed Redundancy) and its capabilities, which aims to be a universal Kubernetes extension and a tool to offer a solution to the for-mentioned issue. However, such a universal tool has drawbacks, especially under heavy-load which can cause the network-throughput to slow down. The initial experiments confirm this, but also that when using DIRE, a session- and delay-critical application can continue its session without failing in case the session-initiating backend fails.

1 Introduction

Modern computing environments often rely on distributed systems to handle large-scale workloads. These systems coordinate multiple nodes to work together, usually made out of numerous, cheap machines instead of a few expensive ones. Managing these numerous nodes enables the opportunity to provide scalability, fault tolerance, transparency while providing the same, high-performance. However, these systems are heavily reliant on inter-networking and communication.

Kubernetes is one of the most popular such tools that can manage and coordinate a group of nodes. It is highly customizable, offering developers a wide-palette of tools and options to alter its behavior. Thanks to the active community behind it and to the heavy industrial usage, it is a constantly updated, stable but still a complex application. Kubernetes provides scalability and fault-tolerance, but by default, no runtime redundancy.

There exist session-critical applications that heavily rely on one specific runtime for low-delays and high-throughput (eg. game-servers, streaming-applications, robotics controllers) and are particularly sensitive to connection state, timing, or consistency. Even if Kubernetes offers a level of redundancy, the progress of a halted/failed runtime without additional intervention is completely lost.

However, to provide true fault-tolerancy, some applications do need a level of runtime-redundancy which for its managed levels of abstraction, cannot be done by Kubernetes itself by default. Applications has to rely on their own runtime-redundancy, which can add large overhead and complicate the application itself.

DIRE aims to offer a universal solution to this problem by eliminating a need for applications to break their abstraction levels and handle runtime-redundancy on their own in a distributed environment.

2 Background

2.1 Kubernetes

2.1.1 Containers and Docker

Containers provide a lightweight and consistent way to package and run applications along with all their dependencies. Instead of relying on the host system's configuration, a container includes everything the application needs (including libraries, tools, etc.) to run their designated code. This ensures it behaves the same in any environment. Container-runtimes are for building, distributing, and managing containers. They simplify development and deployment by reducing issues related to environment differences, improving portability, and enabling faster startup times and efficient resource usage. One of the most popular and notable container-runtime is Docker. However, container-runtimes on their own provide little container-management options such as scaling, advanced routing, etc.

2.1.2 Architecture

Kubernetes [1] is a system designed to manage and automate the deployment of containerized applications across a group (in Kubernetes terminology, cluster) of physical and virtual machines (in Kubernetes terminology, nodes). Its architecture is divided into two main parts: master-nodes and the worker-nodes.

The master-nodes are responsible for making global decisions about the cluster, such as scheduling and responding to changes. Key components run on a master-node include:

- **kube-apiserver** – the central API that all components and users interact with
- **controller manager** – monitors the cluster and ensures it stays in the desired state
- **scheduler** – assigns new workloads to suitable worker nodes
- **etcd** – a distributed key-value store used to store the cluster's configuration and state

Worker nodes are the machines where applications actually run. Each node includes:

- **kubelet** – an agent that communicates with the control plane and ensures containers are running as instructed
- **kube-proxy** – handles networking and routes traffic to the correct workload
- **container runtime** – such as Docker or containerd, which actually runs the containers

Figure 1 illustrates these components and their connections. Together, these components allow Kubernetes to deploy, scale, and maintain applications reliably and efficiently.

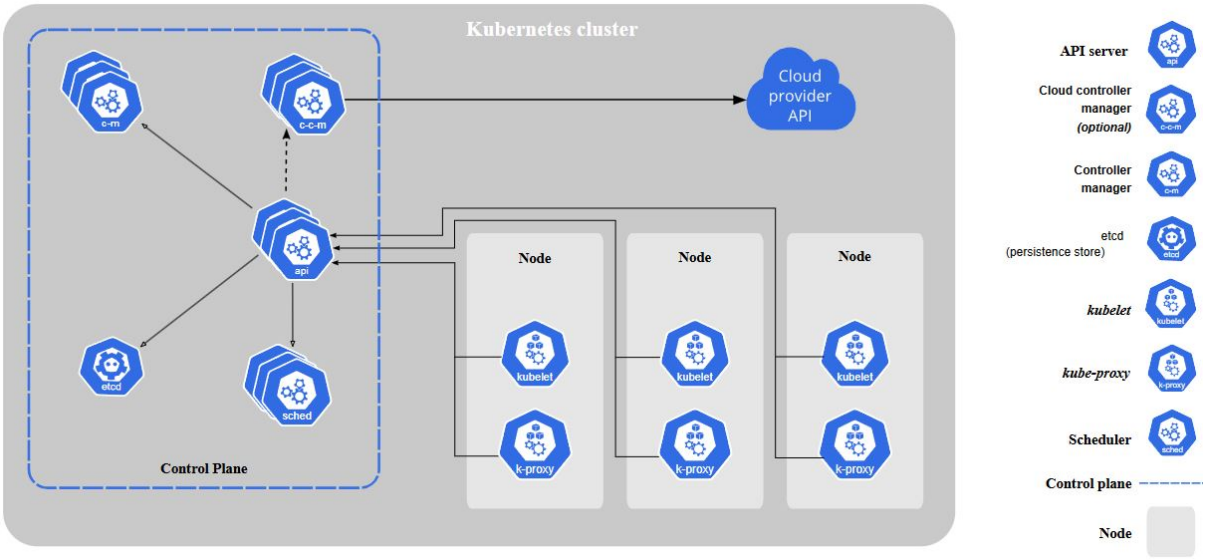

Figure 1: Kubernetes Architecture

Source: https://kubernetes.io/docs/concepts/overview/components/

2.1.3 Resources

Kubernetes state and management is based on so-called resources. These resources are descriptors of objects and their states which the controllers and controller-manager handle, ensuring the correct state and functioning of the cluster.

There are hundreds if not more resources available to Kubernetes, so this list only consists of some of the main and most important ones alongside resources that are relevant for the paper:

- **Pod** – the smallest deployable unit, typically one managed container, however there are cases when it is actually multiple containers (eg. a sidecar for initialization). Each of these pods is assigned a unique IP address when it is created, known as the PodIP. This address allows other applications within the cluster to communicate directly with the pod. However, this IP address is not persistent, and usually accessed by services instead of direct access.

- **ReplicaSet** – ensures a specified number of identical Pods are running at any given time. If a Pod fails or is deleted, the ReplicaSet automatically replaces it to achieve the specified state. They are usually managed by Deployments.

- **Deployment** – provides a higher-level abstraction for managing ReplicaSets and allows for updates to applications (eg. changing the container-image)

- **ConfigMap** – persistent resource used to store configuration data as key-value pairs. ConfigMaps are commonly mounted into Pods as files or environment variables but their key-value pairs can be updated anytime. However, this does not guarantee that the application itself will be notified about the update.

- **Service** – an abstract Kubernetes resource that defines a stable network endpoint to expose a set of Pods. Since Pods have dynamic IPs and lifecycles, Services ensure

reliable communication by automatically routing traffic to healthy Pods that match a given label selector. There are multiple types of services:

- **ClusterIP** – exposes the Service on a virtual IP reachable only from within the cluster. Ideal for internal communication between Pods.
- **NodePort** – exposes the Service on a static port on each node's IP. Useful for accessing the service from outside the cluster using NodeIP:NodePort. However, it is usually used for testing purposes, since it breaks the cluster-level abstraction (you access specific nodes directly).
- **LoadBalancer** – provisions an external load balancer (if supported by the cloud provider) and exposes the Service publicly. It is responsible for distributing incoming traffic across multiple backend Pods

Services provide a decoupled and reliable way to access applications, balancing traffic and ensuring that backend Pods can change freely without affecting clients.

2.1.4 Custom resources and controllers

Due to the extensibility of Kubernetes, developers can define their own resources [2]. These resources then needs to be registered with the Kubernetes API-server, so it can accept changes and the creation of such resources while storing their desired state.

What a custom-resource defines and how to translate it to actions onto the Kubernetes cluster depends on controllers. If a new resource (Custom Resource Definition) is introduced, a new controller also has to be developed alongside with it. Usually, this new controller will be the handler of the new resource.

A controller's Reconcile (technically handling) function is triggered whenever there is a change to a resource it watches; creation, update, or deletion of a custom resource or any related object the controller depends on. The Reconcile loop is usually a logic that ensures a specific state, meaning it may also run periodically or in response to changes in secondary resources like Pods, Services, or ConfigMaps that are referenced by the primary object. Even if nothing changes, some controllers implement periodic re-queues to handle edge cases or missed events.

2.1.5 Distributions - K3S

There are many different distributions of Kubernetes with different defaults, tools, and configurations to match the needs of specific environments or use cases. Some distributions are built for large-scale, production clusters (eg. Amazon EKS), while others focus on ease of setup (eg. K3S), security hardening, or edge computing. Despite their differences, all distributions aim to remain compatible with the Kubernetes API and ecosystem.

K3S [3] (the Kubernetes distribution used for the context of this paper) is a fully-compliant Kubernetes distribution, but it throws away non-essential components to reduce memory and CPU usage; it replaces etcd with a simpler embedded datastore (by default), bundles common tools, and is packaged as a single binary. K3s is especially popular for running Kubernetes on Raspberry Pi clusters, in CI pipelines, or used in testing developed Kubernetes-features due to its extremely simple setup.

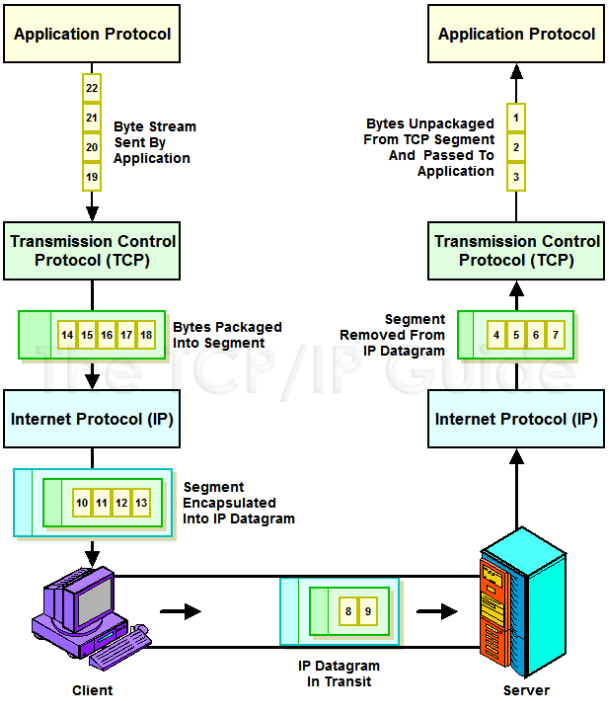

Figure 2: TCP Stream

Source: http://www.tcpipguide.com/free/t_TCPDataHandlingandProcessingStreamsSegmentsandSequ-2.htm

2.2 Networking

Networking is essential in Kubernetes because it provides the primary mechanism for communication between distributed components. Without reliable networking, services cannot coordinate or scale effectively. Since pods are isolated, individual containers running unique runtimes, one way to effectively influence their states is through network communication. Most applications are developed receive both control and data messages through this medium as well.

2.2.1 TCP

TCP [4] (Transmission Control Protocol) is a reliable, connection-oriented protocol used for sending data between two endpoints. It guarantees in-order delivery, retransmits lost packets. It is used by applications that require data integrity and reliable communication, but can tolerate the overhead coming with it.

TCP Streams It is important to note that a TCP connection on application-level is a stream representing a continuous flow of data in both directions over a single connection. It is quite a frequent mistake [5] that developers consider TCP connections on application-level as discrete messages. Once established, the connection persists as long as both parties maintain it, allowing stateful communication. In distributed systems, understanding the timing of responses over TCP is important for measuring backend performance, detecting unresponsive endpoints, and managing failover. Figure 2 showcases how a TCP stream is handled.

2.2.2 Proxying Difficulties

Proxying in distributed systems is complex. Maintaining TCP sessions, handling authentication, and forwarding data without interruptions are non-trivial.

UDP in Kubernetes UDP is connectionless and does not guarantee delivery, ordering, or even arrival at all. Kubernetes services provide basic UDP support, but features like load balancing, NAT traversal are far more limited than for TCP. UDP traffic directed to external clients outside of the cluster is difficult without a mechanism like a STUN server or tools like STUNner [6], which help establishing connections despite the numerous address-abstractions. Without such a mechanism, establishing and maintaining reliable UDP communication especially peer-to-peer, becomes infeasible in many cluster environments.

Session-based authentication Many applications require session-based authentication to track users and permissions across multiple requests. This involves assigning a session token after login and associating it with a client connection. This adds complexity for proxy-applications: the proxy must either terminate and inspect connections (e.g., TLS termination) or forward and preserve authentication headers or tokens. In case of DIRE (see section X), this would add excessive complexity for the goals of this paper.

2.3 True Randomness

Most applications use pseudo-random numbers. True randomness, which comes from physical phenomena, can be important for fairness or security. There exist services that provide true-randomness through API calls or different methods (eg. Random.org [7]). Well-developed applications also usually have the option to use an external random-service instead of generating their own random-number. This is an important and necessary design for DIRE to work with non-deterministic applications.

2.4 Parallel Computing

Parallel computing enables a system to perform multiple operations at once. It is key to speeding up tasks in scientific simulations, video rendering, and large-scale web services. It is also used frequently to provide redundancy runtime-level for critical applications, but unfortunately in a distributed scenario this is not possible, as even if a single instance can technically provide itself runtime-redundancy, if the instance itself fails (most common in distributed systems) this kind of redundancy has no real effect. However, the idea itself elevated to another abstraction level to work for instances as well, can provide a way for a solution to this difficult problem.

2.5 Tools used for the experiments

Iperf3 Iperf3 [8] is a lightweight tool used to benchmark network performance between two endpoints. It follows a simple client-server model, where one side runs in server mode and listens for incoming tests, while the other acts as the client and initiates the measurement. Iperf3 can simulate both TCP and UDP traffic and provides detailed statistics such as throughput, jitter, and packet loss. This makes it useful for evaluating network capacity and diagnosing performance issues.

vmstat vmstat [9] is a Unix/Linux command-line tool that provides real-time summaries of system performance by reporting statistics on CPU usage, RAM, swap activity, I/O operations, and system processes. It works by sampling kernel counters to show resource usage either as averages since boot or at regular intervals when specified, helping users quickly identify bottlenecks like memory shortages, high CPU load, or heavy disk I/O.

3 Problem definition

3.1 Importance of redundancy

As mentioned in [2.4], for critical applications, it is necessary to provide even session-level fallback in case of failure (e.g., processes that cannot be interrupted or might result in catastrophic consequences). Achieving this on a single machine or within a single runtime is feasible and often done for result-validation or resilience.

However, now that most applications are being deployed on the cloud or in orchestrator-clusters on top of many, unstable nodes, this abstraction level does not guarantee session-level redundancy.

3.2 Session-redundancy in Kubernetes

Kubernetes does offer redundancy - but not on this level. It launches multiple application pods, so in case one fails, another can take its place. However, the runtime-session that pod was having is permanently lost and a new one has to be started. Some delay-critical tasks however does not have the time/capabilities to start a whole new session and then continue where the logic reaches where they left off. The only solution to this problem is to ensure that all pods are in the same, synchronized state. This is an extremely challenging and broad task, if not impossible. Not to mention the Kubernetes implementation of such solution. However, in case of deterministic applications (applications that provide the same results based on the same external-inputs) and applications that do not generate unique random values for their runtimes (eg. random-id, tokens, hashes) but instead use external services for that, this can be feasible. Kubernetes does not natively support multicasting traffic to multiple pods simultaneously (forwarding the same input-data to all backends), load-balancer only forwards each request to a single pod.

The goal of this paper to provide a prototype application/kubernetes extension that will allow applications to have multiple, horizontally scaled, in-sync sessions, providing a fallback-option in case of failure to ensure seamless experience from the client side.

3.3 Problem statement

This paper sets the goal to solve this problem; provide a prototype of an application that can handle runtime redundancy in a Kubernetes environment by means of input duplication and runtime-syncing.

4 Methods - DIRE

To provide a solution to the aforementioned issues, the paper introduces the DIRE Kubernetes extension prototype that can successfully multicast incoming input-data between selected pods, handle random-variable request and doing all that while aiming to be transparent to the client.

4.1 Architecture

DIRE is made out of a few, different components for different purposes in a Kubernetes cluster;

- **DIRE Controller** – the deployed controller watching scheduled DIRE instances, responsible for managing them as well
- **HIVE deployment** – a standard deployment. Each replica will run an instance of the backend-application. To enable extensive configuration, the hive-deployment is not included in the DIRE CRD, it has to be deployed separately.
- **DIRE Instance** – the resource managed by the DIRE controller, consisting three main components:
  - **Distributor** – the distributor that functions as a TCP proxy, containing networking and redundancy-handling logic
  - **ConfigMap** – a configmap that contains important configuration values for the Distributor, either parsed from the DIRE Instance resource itself, or from the cluster (pod IPs)
  - **Service** – a service (NodePort, LoadBalancer) put atop of the Distributor to expose it for clients

Figure 3 illustrates the components and their relations in a Kubernetes cluster.

4.2 Process

To develop such a tool, I collected I set of milestones to achieve;

- **Custom Resource Definition** – the DIRE must be easily deployed and configurable, thus a CRD to handle it in a Kubernetes environment is necessary.
- **Controller** – to handle a CRD, a custom Controller needs to be developed; this should be able to deploy and update the resources needed for DIRE.
- **Distributor** – once the environment is set for an application that has the relevant information (in this case, the PodIPs of the endpoints, and the ports used by the services running on them), this application should be developed in a way that enables a TCP session takeover - thus, providing runtime-redundancy for applications that depend on network-input.
  - **TCP proxying** – first and foremost, the Distributor should be able to act as a functioning TCP proxy.

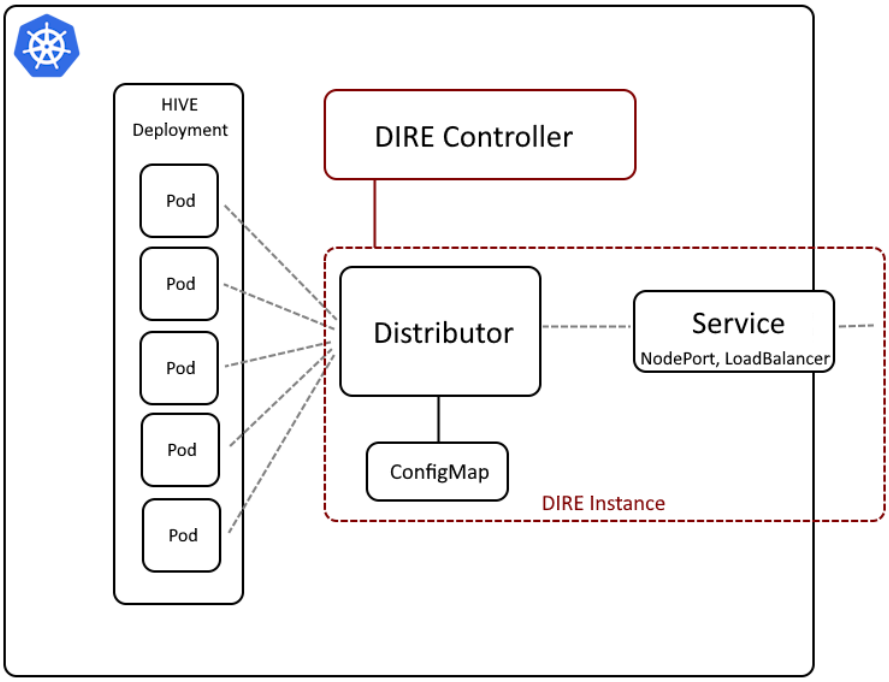

Figure 3: DIRE architecture

- **TCP stream 'replicating'** – once the functionality of a basic proxy is achieved, the Distributor should be updated in a way that enables it to forward an incoming TCP stream to all specified endpoints - without altering the content of the stream.
- **Primary election** – to be able to respond in an efficient way, only one endpoints response should be forwarded back to the client. To determine which endpoint this will be, a method has to be introduced that selects a primary-endpoint through some logic, preferably choosing the most reliable one.
- **Switchover** – in case a primary endpoint fails, a switchover should occur that elects a new primary and preferably seamlessly keeps forwarding the responses from the newly elected primary.
- **External random variables** – to ensure compatibility with even some non-deterministic applications, the Distributor should have the functionality to serve random variables to endpoints. These variables should be stored and attached a TTL - so consecutive requests will be answered with the same value.
- **Disabling endpoints** – to lower the resources used and the communication delay the Distributor introduces (especially in case of a new primary-election), failed endpoints should be marked as disabled. Disabled endpoints should not participate in a primary-election, and no data should be forwarded to them.

4.3 DIRE Controller

As aforementioned, the DIRE Controller is responsible for handling and updating DIRE resources. The DIRE CR (Custom Resource) holds the following values alongside the standard metadata-fields (name, labels):

- **Kind** – This is the field describing that the resource is of type 'DIRE'. This is a static value.
- **spec.distributorImage** – a string, the container-image to use for the deployed distributor
- **spec.hiveDeploymentName** – a string, the name of the deployment that should act as a HIVE behind the Distributor, optionally already synchronized and waiting for session-start
- **spec.forwardedPorts** – a list of integers, which represent the ports that should be handled by the Distributor (these ports will be read from, and will be written to the endpoints). This usually matches the ports the hive-applications use for communication.

The DIRE Controller runs a reconciliation function that handles scheduled DIRE CRs in the following order:

1. The controller fetches the newly-found/updated DIRE CR
2. It tries to accumulate a list of pods that belong to the specified hive-deployment. If it fails, retries the reconciliation in 5 seconds.
3. It then collects all podIPs of these pods into a list
4. It creates a ConfigMap that contains all the endpoints, so as the ports that were specified in the CR values
5. It creates a deployment which includes a single Distributor replica, mounting the created ConfigMap for it as a volume
6. It creates a Service (currently NodePort) that targets the Distributor, exposing it outside of the cluster

It is important to note that building network logic on PodIPs in a Kubernetes context can lead to lack of flexibility, however, since in our context the Pods do have to maintain their state, if any of them rescheduled - is "failed" - we are unable to use it as a redundancy-option anymore anyway.

It is also important to note, that in case of a Distributor fail, even if the ConfigMap is updated accordingly to the deployment, the Distributor wont import it dynamically, so it has to be re-scheduled manually (by deletion of the Distributor instance).

4.4 DIRE Distributor

The DIRE Distributor is technically a hybrid multicasting TCP-proxy. It has a list of endpoints and ports based on the ConfigMap that was mounted to it. For each of the ports, it starts a TCP listener. Due to the challenges an UDP communication in a Kubernetes environment proposes, and the fact that this paper focuses on Kubernetes implementations rather than networking-details, it is not implemented as of 2025.06.09.

The features and methods of the distributor are the following:

- **Primary election** – To ensure that the client does not receive multiple responses, and to avoid clogging the network unnecessarily, one of the main functionality of the distributor is to constantly look for primary-endpoint to forward to the client from. There are multiple ways to elect a primary (from which there is only one per stream):
  - **Initial election** – at the start of the proxying phase, the distributor initiates an ICMP ping to all endpoints, that respond with an RTT value. This RTT value is used to elect the endpoint with the fastest response-time as the primary.
  - **Primary failed response** – in case a read from the primary results in an error, if there are endpoints that returned from the read with no-error, a new primary election is being initiated; it selects the second fastest responder.
- **Reading into buffers** – a go-routine is launched for each endpoint to read from - the results are stored in a limited 4096 byte in-memory buffer. Only the buffer-content of the primary endpoint will be written back to the client if the read resulted in no error.
- **Disabling endpoints** – in case an endpoint is not reachable during the initial primary-election or the primary- (or any other endpoint) read results in an error, the respective endpoint will be marked as disabled. Disabled endpoints can not participate to be a primary, and there will be no reading-writing happening to them to save resources. However this raises the concern of having limited amount of endpoints once we start the session, but this is an expected behavior, and originates to the same problem as [reference to podIP usage].
- **External true-random service** – currently DIRE supports non-deterministic applications that use random variables, if such random variables can be requested from external source. For such cases, the distributor is serving an HTTP endpoint on port 26666, with the request schema being the same as a random.org [reference] integer request. Incoming requests to the Distributor will be forwarded to random.org, requesting random integer(s) described by the HTTP request, and assigned a TTL (time-to-live) value. In a case where a request occurs but there is still a value with an alive TTL, we serve that value instead of requesting a new one, thus ensuring all the in-sync endpoints receive the same value.

4.5 Expected limitations

Since DIRE is currently a prototype for a large-scale application, there are a few serious limitations to it:

- **Corrupted endpoints** – since currently DIRE has no tools to confirm a validity of a TCP stream - byte stream, not packets that can be validated against other endpoints - in case an endpoint gets corrupted instead of failing or returning an error, the distributor has no means to detect that.
- **Locally generated variables** – some applications do not explicitly generate random variables, but assigns one to itself or the session, using some kind of seed, or through other methods a unique value. Since the Distributor has no influence over such tokens and values, if the application-layer is strict about it, it can cause errors.

5 Experiments

Despite its limitations, DIRE represents a significant improvement in handling failovers. This paper proposes an experiment focused on evaluating its effectiveness, including its impact on throughput, network characteristics, and resource usage as the number of endpoints increases.

5.1 Experiment method

In the experiment, the used application as backend will be iperf3 servers. One iperf3 client will conduct a request first with 1, then 10, finally 100 endpoint instances running. Each of these request will be a 160 seconds long TCP stream. Upon collecting the data, the experiment will be repeated, but now with the Distributor being replaced by a simple NodePort. By these iperf3 results, we will be able to summarize and reflect on the overhead caused by the Distributor. Finally, we repeat this experiment once again, but every few seconds an endpoint will be deleted from the backend, forcing the DIRE to initiate a primary-change. With this data, we can also approximate and reflect on the delays caused by the Distributor primary-switch.

Also, alongside the iperf3 client, on the testing machine vmstat will be running for 180 seconds (10 extra seconds before and after the iperf3 serving to establish a baseline) to approximate resource-usage and its effect on network-throughput and other attributes.

5.2 Experiment environment

The experiment is done on a single machine with a Intel i7-9700k 3.60GHz 8 core CPU and 32 GB of RAM. The testing-machine is running WSL (Windows Subsystem for Linux). The WSL is running a K3S cluster with a DIRE controller running, what already deployed a DIRE instance on top of a hive-deployment. The iperf3 client and vmstat will be started from a script inside the WSL.

5.3 Results

The following graph-groups illustrate the results of a number of experiments (please note, that not all results are included), comparing resource usage and network attributes of different test-scenarios. The metrics used are the following:

- **CPU usage(%)** – percentage of the CPU utilization - extracted from vmstat results
- **Context-switching** – shows how many times the CPU (central processing unit) switches from running one process or thread to another - extracted from vmstat results
- **Mbytes transmitted** – amount of the megabytes that has been successfully sent during the period (one second) - can be translated to throughput as well - extracted from iperf3 results
- **Retransmits** – in case a packet is lost during a connection, it is being re-sent. This metric counts the occurrences of such events - extracted from iperf3 results

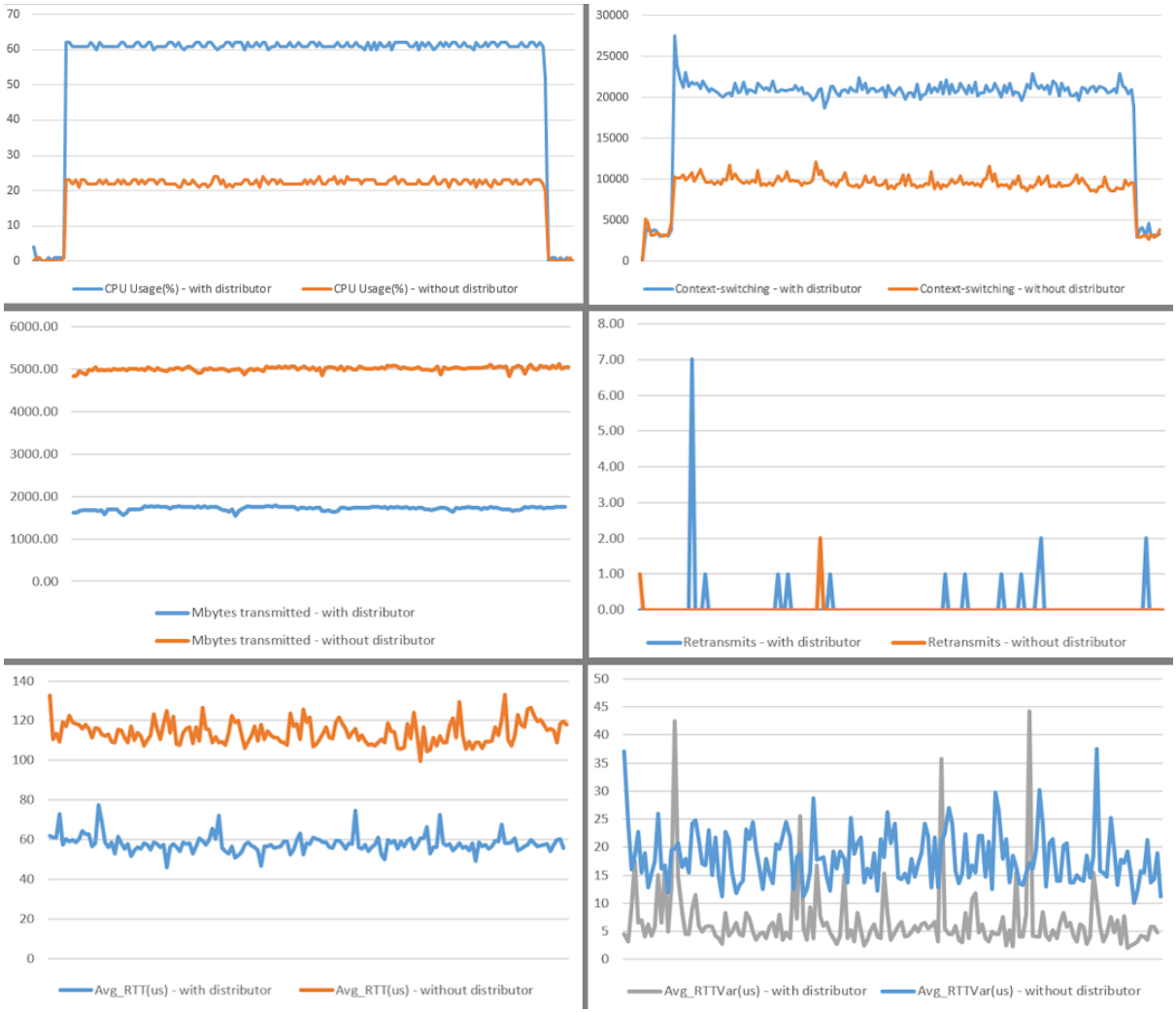

Figure 4: Resource usage and network impact of the distributor

- **Avg_RTT(us)** – measures the latency of a network connection in micro-seconds by averaging the delays of multiple packets over a period - extracted and computed from iperf3 results
- **Avg_RTTVar(us)** – measures the variance of the latency (jitter) of a network connection in micro-seconds by averaging the latency-variance over a period - extracted and computed from iperf3 results

Group 1 (figure 4) of results represent the comparison of the resource usage and network-attributes of a single iperf3 endpoint session - with and without a distributor over the course of a 180 seconds. Due to the Distributor being a prototype, it was expected that the resource usage of it is far from optimal yet, but the result of the test with distributor in place using more than twice as much CPU-computation is especially disappointing. RAM usage was constant, the Distributor logic only stores a few variables in memory, it was excluded from this measurement. The results of Avg_RTT measurement can be misleading; the way iperf3 measures RTT is from TCP socket to TCP socket - the fact the the Distributor functions as a proxy can influence the results.

Group 2 (figure 5) of results aims to focus to illustrate the scalability of the Distributor. Please note that three out of four graphs are using logarithmic scale for better illustration.

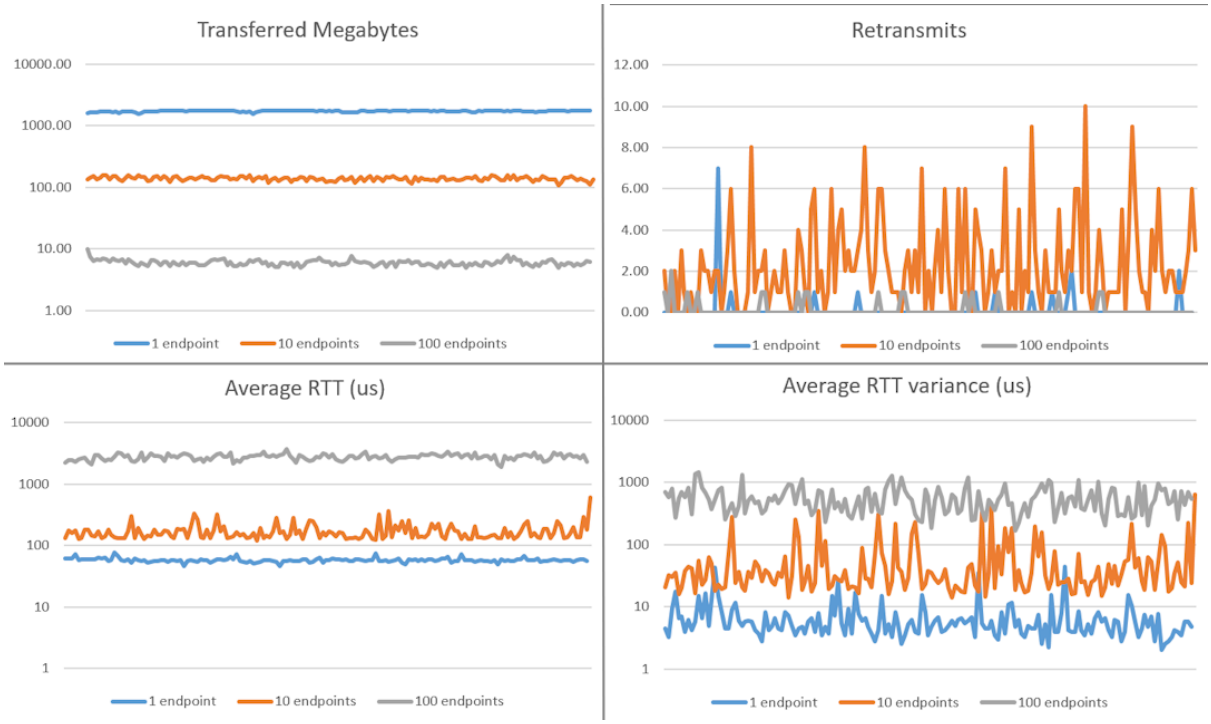

Figure 5: Network impact of the scaling of endpoints

Resource usage was not included in the comparison, the CPU usage with a distributor was averaging around 60-70% at each scenario, independent of the number of endpoints. The conclusion is from the results that even if the Distributor influence on the network is quite negative, this influence is linear with the number of endpoints - most probably due to the heavy resource-usage.

Group 3 (figure 6) of results provides insight of how the main functionality - the ability to switch endpoints for a seamless session in case of failure - influences the connection. During the 180 second session, endpoints were randomly killed, forcing a takeover-logic to happen. This can be the root cause of the large spikes of increased Avg_RTT and Avg_RTTVar in the first-half of the experiment, increased delay in case of a take-over. Another important clue is the increased amount of transferred megabytes over the second half of the experiment - this is caused by freeing up resources by killing endpoints forcing the Distributor to mark them as disabled. The differences in the amount of retransmits and the decrease of Avg_RTT can have the same explanation: with the number of the endpoints - and thus, with the number of active connection handled by the Distributor - decreasing, the Distributor can use these free resources to function faster.

5.4 Experiment limitations

Due to the lack of hardware, the experiments were conducted on a single bare-metal machine, NOT specifically set up for this experiment. Because of this, the experiment was running on a single node, leading to a virtualized networking environment, which can differ from a real-world one in terms of performance. However, due to the lack of extensive hardware-resources, the Distributor could be observed under heavy load and strict resource-restrictions - and how such resource-limitations affect the performance of the Distributor - in a way that would happen in a real-world scenario.

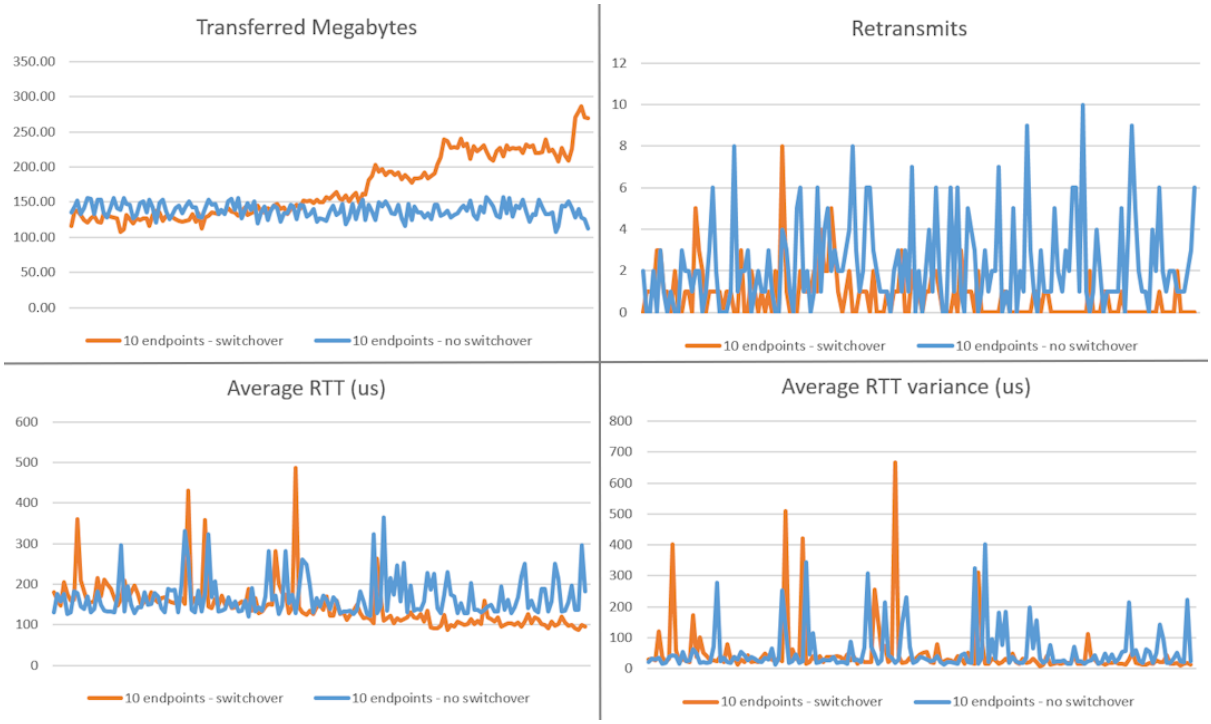

Figure 6: Network impact in case of endpoint-failure

6 Related work

"A Kubernetes Controller for Managing the Availability of Elastic Microservice-Based Stateful Applications" [10] proposes a custom Kubernetes controller designed to improve the availability of stateful microservices through intelligent runtime redundancy. Unlike Kubernetes' default self-healing mechanisms, which may incur delays during recovery, this controller maintains explicit primary–standby pod pairs and actively monitors their health. Upon detecting a failure in the primary pod, the controller orchestrates an immediate switchover to the standby, significantly reducing downtime—achieving up to 50% faster recovery in their evaluations. This approach is conceptually similar to the goals of the DIRE Distributor, which aims to maintain live TCP session continuity across multiple proxy pods. While the paper focuses on stateful applications like databases, the DIRE Distributor generalizes this redundancy pattern to network traffic, replicating sessions to all backends in parallel and dynamically electing a primary for response handling. Both systems highlight the importance of low-latency failover and health checks, reinforcing the value of purpose-built Kubernetes controllers for enhancing service resilience.

There also exist multiple TCP proxy applications (such as Miniproxy [11]) which achieve considerable better performance than the DIRE Distributor, however none or few of them considers the ability to make these sessions redundant - this makes DIRE to have a unique approach to the probelm.

7 Future work

Since DIRE is a prototype, it leaves much room to improve;

- **Support of other network protocols** – DIRE currently only supports TCP, but for it to be a truly universal tool for most applications support of UDP and QUIC

would be a huge improvement. The solution for the former could be an integration with STUNner [6], while the latter would require a Distributor-logic overhaul due to its need for TLS (Transport Layer Security)

- **Scalability of the Distributor** – currently even if the Distributor eliminates a single-point-of-failure, it creates another one - itself. To mitigate this issue, the Distributor could be scaled horizontally (by number of replicas). However, this would require a shared state-database for the endpoint-states and random-variable with their TTLs. It also raises the need to transfer the distributor functionality in case of failure to another - but this is a much more straightforward problem to solve than the one the paper aims to solve.

- **Extended configurability** – currently the external random service run by the Distributor is hard-coded for port 26666 and for the external endpoint random.org. In case the user would want to use a different service, it could be configured beforehand.

- **Sync-checks** – even if the primary-election cycle ensures no failed endpoints are being considered, the issue mentioned in [4.4 A] can still cause overall major failures. To mitigate this, occasional sync-checks could be considered (comparing incoming data to other endpoints to ensure validity), which even if bringing a large computational overhead and the need for a different Distributor approach would provide a safer method.

- **Graceful-shutdown of disabled endpoints** – in case the Distributor marks an endpoint as disabled, it will stop sending and receiving data from it. However, the pod will still be running, only its requests will be discarded. This wastes resources; in such cases, the affected endpoints should be shut down.

- **Dynamic configMap read - and restart** – currently the Distributor logic is for one cycle only - if the TCP stream ends, it closes all endpoints connections, and enter a dormant state. To restart, the DIRE CR has to be re-deployed. The distributor could have dynamic configMap access to be able to start a new TCP session with a different set of endpoints. Currently the configMap is read on the startup of the Distributor only.

- **Source client IP** – currently, when the endpoints receive any data, it will be labeled with the Distributors cluster-IP set as the source. This can influence and affect proper debugging, logging, or the capture of other business-side metrics.

8 Discussion and Conclusion

DIRE is a prototype designed to introduce session-level redundancy through a novel abstraction layer built on Kubernetes. Thanks to the extensibility of Kubernetes, the in-cluster management and configuration of such an application is possible; through customized controllers, custom resource definitions and dynamic endpoint-handling the DIRE Controller can provide a proxy-multicast application everything it needs for allowing session-level redundancy with extensible configuration. All this, without needing much in-depth understanding from the user using it. However, building such a Distributor — capable of synchronizing endpoints, forwarding traffic to all of them, and managing

shared random values, etc. — is a complex task that requires a deep understanding of networking and Kubernetes internals, especially when working with UDP traffic in Kubernetes environments. The experiments aimed to evaluate the Distributor yielded varying results. The Distributor - compared to simply run sessions - consumes a significant, more than twice as much computation power, and even with these resources only provides slightly reduced network performance. This performance degradation scales linearly with the number of endpoints and appears to correlate closely with CPU usage. Despite these limitations, the DIRE Distributor effectively demonstrates that Kubernetes-level session redundancy is feasible and practical, even if it comes with a price.

9 References

References

[1] Google. *Official Kubernetes Documentation*. https://kubernetes.io/docs/concepts/architecture/. Last modified April 20, 2024.

[2] Google. *Custom Resources*. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/. Last modified October 31, 2024.

[3] K3s Project Authors. *Official K3S Documentation*. https://docs.k3s.io/. Last updated on Jan 15, 2025.

[4] Ed. MTI Systems W. Eddy. *Transmission Control Protocol (TCP)*. https://datatracker.ietf.org/doc/html/rfc9293. August 2022.

[5] James Stanley. *How to read from a TCP socket (but were too afraid to ask)*. https://incoherency.co.uk/blog/stories/reading-tcp-sockets.html. Last modified February 10, 2024.

[6] L7mp Technologies Kft. *STUNner documentation*. https://docs.l7mp.io/en/stable/.

[7] RANDOM.ORG. *HTTP API to get true random numbers into your own code*. https://www.random.org/clients/http/. Accessed June 2025.

[8] Bruce A. Mah Jeff Poskanzer Kaustubh Prabhu Jon Dugan Seth Elliott. *iPerf / iPerf3*. https://iperf.fr/.

[9] IBM. *vmstat*. https://www.ibm.com/docs/en/ssw_aix_71/v_commands/vmstat.html.

[10] Leila Abdollahi Vayghan et al. "A Kubernetes controller for managing the availability of elastic microservice based stateful applications". In: *Journal of Systems and Software* 175 (2021), p. 110924. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2021.110924. URL: https://www.sciencedirect.com/science/article/pii/S0164121221000212.

[11] Giuseppe Siracusano et al. "On the Fly TCP Acceleration with Miniproxy". In: May 2016, pp. 44–49. DOI: 10.1145/2940147.2940149.