Research report on FractOS

Maryia Lapatsina
Technical University of Munich

Vladislavs Viktorovs
Technical University of Munich

Abstract

A modern data centers (DC) tend to be disaggregated since it offers faster, less-expensive server upgrades and pooled server resources.

A modern disaggregated heterogeneous data center offers faster, less-expensive server upgrades, as well as server resource pooling. This data center design separates different devices, such as CPUs, memory devices, and accelerators into distinct interconnected via network nodes, which helps with maintenance, allocation, and sharing of hardware resources. Although its benefits, to successfully pass the execution flow, this design introduces redundant communication over the data-center network. On the contrary to traditional data center architectures with communication between devices over a fast local bus, which makes this redundancy not crucial, a disaggregated system has to use a much slower shared network, which brings higher latency into communication.

In order to benefit from the disaggregation of DCs, we require a novel approach in device communication to allow peer-to-peer data and control transfer, minimizing network overheads.

In the following report, we would like to present the work, that aims to solve the stated problems, as well as our own research.

1 Introduction

While devices in traditional server architectures communicate through a fast local PCI bus [6]; in disaggregated systems they are performed over a shared network, which leads to much higher latency by such transfers [7].

Through the years, several disaggregated system architectures were developed, trying to solve this design issue in several ways. To summarize the core differences between approaches, we can state, that a disaggregated system can have two kinds of application and OS models: centralized and decentralized. In the following, we inspect all 4 possible designs, to see, which combination of these models has the best performance by the disaggregated heterogeneous data centers.

The first design combination we would like to discuss is a popular solution in disaggregated DCs, namely centralized design both for OS and application. This design allows treating remote devices as local ones, with the difference in connection only - shared networks instead of PCI bus. Although this model allows running applications without any modifications relative to the system architecture, it also hides the distributed nature of disaggregated systems, by centralizing control on both levels.
The next approach combines distributed OS and centralized application designs, introduces a high-level Remote Procedure Call interface that bonds nodes with *Controllers* (e.g., monitors in LegoOS [8]), that are monitoring and managing the network messaging. Although this design reduces the overheads of remote device access, some arbitrary nodes, such as for example, the GPU and SSD, cannot directly interact with each other and need to communicate through the centralized communication moderator (e.g., application CPU).
In both approaches, communication between the disaggregated nodes requires a centralized moderator, which leads to redundancy in communication.
As an alternative, there are architectures with centralized OS design but distributed applications. However, these designs are not suitable for multi-device

distributed heterogeneous systems, because the disaggregated nodes do not support the high-level interfaces of such frameworks. Though, it is possible to attach a CPU to each node, this would defeat the purpose of resource disaggregation.
Since previously developed disaggregated system architectures overlook the tradeoffs stated above, we need a new environment which would optimize the workflow in the disaggregated data center. The key requirement for this architecture would be the possibility of a direct data and control transfer between arbitrary nodes, in order to get rid of the redundant network communications inherent to centralized designs.

2 Proposed solution

In the following we would like to explore FractOS [1], an operating system for disaggregated heterogeneous data centers, that tends to combine both distributed application and OS models. This design enables fully decentralized execution of application logic, minimizing the networking costs of disaggregation. It is fairly suitable for disaggregated data centers, since it is modular and gives priority to both compute and storage devices, so the devices are treated as first-class citizens.

To better understand the structural difference in device communication between centralized design and distributed model that is being achieved by FractOS, we will consider the following application example. Application requests some input data from a remote SSD, transfers it to the GPU node where it performs some calculations, and then passes it to some remote file server, where the output data will be stored. To achieve the result with the traditional architecture, we need to pass the message back and forth to the application CPU node, so the application itself performs all the control and data transfers, forming a star-shaped topology. On the other hand, the FractOS model provides another workflow: the application defines how each device must be used in turn, but devices communicate directly, forming a ring topology. It is apparent, that the centralised design requires much more control and data transfers than the distributed one and that the peer-to-peer communication flow, as in the FractOS design, would be optimal for this type of application.

Building such a system introduces some challenges. First, all disaggregated nodes should be able to successfully use the FractOS API and to pass the control flow to the following nodes. It is also obvious, that there are security issues, that pivot on peer-to-peer node interactions without a layer of a trusted OS in between. Finally, there is a challenge in authorising nodes to access some objects and revoking this authority. To overcome these challenges, FractOS proposes several solutions.

To begin with, FractOS introduces two abstractions: *Request* objects, that are used to pass the RPCs between devices, and *Memory* objects, that manage memory references in RPCs.

In addition to these objects, we also need *Controllers* and *Adaptors*. *Controllers*, that run on CPUs or SmartNICs, are used to build a secure layer, to provide all trusted mechanisms for the RPC interface. They are isolated from services through a message-passing interface. Utilizing this isolation, each service and *Memory* object is associated with a single Controller and uses it to pass messages to other services. Besides, each device is also bonded with an *Adaptor*, that implements the device's RPCs, and transforms RPCs into device operations. FractOS treats *Adaptors* same as the services, so they are untrusted by the Operating System and run in *user-mode*.

In pursuit of decentralized execution, FractOS uses a continuation-based procedure-passing protocol. In a nutshell, each RPC refers to the other RPCs, forming a continuous call sequence. Such references are called *Requests*, that in turn consist of the parameters needed for the invocation of the next RPC. The chain of *Requests* is called a task graph, that is defined by the application. On top of that, *Requests* are dynamically composable and can be refined with new arguments and capabilities at each of the end points.

Finally, FractOS implements a capability system, to grant and revoke authority to access objects. The capability model is token-based, so the owner of the object can grant a *token* to a node, and take it back, when the access should be revoked. This system design requires a scalable revocation system. Frac-

tOS makes use of a new distributed capability design, which in comparison to other capability systems (e.g., SemperOS [2]) integrates both efficient delegation and immediate selective revocation in a distributed manner.

3 Research context

For our research work, we decided to imitate the distributed system to research its behavior and ability to run distributed applications. We found the idea of FractOSs promising and chose to focus on its design. In pursuit of this design to be applicable to distributed data centers, we decided not to use local PCI bus to transfer data between nodes, but rather build the communication over a shared network, by the use of gRPC [3], as the RPC protocol.
In addition to that, we want to test the system on some rather simple, but representative "real-world" application scenarios. To achieve this, it is not enough to develop the system itself, but also provide some distributed applications and inspect their execution flow. We also want to see how the system reacts to changes, node failures and hence how it regroups and continues its work.
The challenges by conducting this research are similar to the ones, stated in the initial paper [1], i.e. creating isolated communication layer, and distributing the control flow between different nodes.
Another challenge we see, is creating several similar nodes and choosing the most suitable executor for the operation (*Leader*).
Finally, we need to connect our distributed system to an external monitoring service and observe its behavior.

4 Implementation

As it was already mentioned, we have decided to imitate the distributed system from scratch. In our implementation, each node run on separate server and communicate with each other only through RPCs, in our case gRPCs [3], encoding data with Protocol Buffers [9], that makes it possible to use passed RPC arguments as objects (e.g., string, int, list) directly in the source code. Each node represents a microservice that contains a part of the distributed operating system logic. In our implementation we decided to implement the following nodes, to be able to run various applications: data sender, storage, several computational and of course application nodes. Each of the nodes has a service, that is connected to a single *Controller*, that receives RPCs from other nodes and send the following to the next and *Adaptor*, that transforms the incoming RPCs into device operations.
Our mock services are an abstraction of hardware, e.g., a storage service represents an SSD, or a computational node mocks the CPU/GPU.
Unlike the FractOS design, where *Controllers* only implement the RPC mechanism, and *Adaptors* send requests to the next node, in our system, *Controllers* send requests themselves. As the whole system runs in *user-mode*, isolation from *Adaptors* is not crucial, but only complicates the representation of it.
Adaptors perform two main tasks: parse the *Request* arguments to direct execution flow to the next node and translate incoming RPC into device operations, as it was designed in FractOS.
Our *Requests* are a *repeated string* protocol in gRPC request, that has the following structure:

```
Request:
["NodeType,ip:port,agr1,...,argn", ...]
```

We implement a unidirectional flow of RPCs; hence the previous node receives only *response code* and *description*, that show, how successful, the sended RPC was.

After we built a system with these controllers already communicating through gRPC, we decided to reproduce some nodes, combine them in clusters, and attach a distributed monitoring service. This way, when some of the controllers fail, the system does not corrupt, but it reroutes the execution through the controllers that are still standing. As a distributed monitoring service, we decided to use ZooKeeper [4], that implements *distributed synchronization*. Our clusters use it to maintain shared resources and decide, which node will be the *Leader* and will run the service, as well as, pass the control flow to the next

node. ZooKeeper ensures that every operation performed by a node in a cluster either succeeded or failed to do the job - never performed part of task.
It also follows the client-server architecture, where our nodes from the cluster are clients and ZooKeeper is a server. If any server loses connection, its client will be simply redirected to another one. Consider having 2n + 1 with $n > 0$ servers. This group of servers (*ensemble*) can handle exactly n node failures to keep the cluster running. The amount of nodes needed for ensemble to run is called a quorum [5].
ZooKeeper introduces a shared hierarchical namespace for distributed processes, which is organized similarly to a standard file system; thus it can achieve high throughput and be fairly suitable for distributed systems.
This monitoring service notifies clients about any changes in the ensemble via watches. Watches send a notification to the registered client for any of the server changes. E.G. server became a leader. Watches for a particular server are removed as soon as its server is disconnected.

To observe the workflow of our system, we have developed a few applications. First, an extremely simplified convolutional neural network (*CNN*) with the help of Keras [10] for *CNN* training and binary classification of an image. The execution flow of this application is quite representative: application sends name of an image to the *Application Controller*, it redirects the request to *Storage Controller*, *Storage Controller* encodes image and sends it to the *CNN Controller*. When the image is received, *CNN Service* classifies the image and sends the result to the *Storage Controller*, where it will be stored. Our second application calculates the *n-th* power of a number in the *Storage*, by the use of our *Mathematical Node*. The execution flow is quite similar to the previous one. The interesting part, that the task graph is dynamically constructed in by the application, after the argument n is received.

For the workflow to be more representative, we have implemented optional arguments for our *Controllers*.

-z or --zookeeper	to run *Controllers* with zookeeper
-v or --verbose	to see logs in console
-n or --name	to set server name

To test the system, we have crashed some of the *Controllers* and observed the behavior of the system. As soon as our environment gets a message from ZooKeeper that a node (e.g., *Leader*) is disconnected, server chooses another *Leader* from the cluster and goes on with the application execution. Therefore, execution fails only when our last node is disconnected and we have no other service to run the operation.

Further research

As we already mentioned, we have made a decision not to have *Memory* objects as originally were introduced by FractOS, and therefore, not to implement a capability system in order to keep system fairly representative. The implementation of the capability model can be considered as future research into this work.
Furthermore, since we have developed and integrated some nodes and applications, we leave integration of the rest of other node types (e.g., remote File Server) and other applications for further research as well.

Conclusions

We have imitated the disaggregated distributed system that can run various types of distributed applications following the design of FractOS. We also had an opportunity to use new frameworks such as ZooKeeper and gGRPC. On top of that, the intriguing part was to observe how several services exchange control flow, choose their *Follower/Leader* nodes during the application execution or during service crashes, that we have injected.

Acknowledgments

This research was inspired by the paper on FractOS [1]. We would like to thank our Professor on Modern Data Center Systems Seminar, Pramod Bhatotia, for introducing to us this paper.

Availability

The implementation of both mock-system and applications in python you can find under the following link: `https://github.com/Voltorane/sys-micro-fractos-mock`.

References

[1] Lluís Vilanova, Lina Maudlej, Shai Bergman, Till Miemietz, Matthias Hille, Nils Asmussen, Michael Roitzsch, Hermann Härtig and Mark Silberstein. 2022. Slashing the Disaggregation Tax in Heterogeneous Data Centers with FractOS. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France*. ACM, New York, NY, USA, 16 pages. `https://doi.org/10.1145/3492321.3519569`.

[2] Matthias Hille, Nils Asmussen, Pramod Bhatotia, and Hermann Härtig. Semperos: A distributed capability system. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 709–722, 2019.

[3] gRPC is a modern open source high performance Remote Procedure Call framework that can run in any environment, that was initially created by Google; `https://grpc.io/`.

[4] Apache ZooKeeper is an open source volunteer project under the Apache Software Foundation; `https://zookeeper.apache.org/`.

[5] Quorum is the number of nodes that should be up and running in the ensemble so they would be considered a cluster. `https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2008-R2-and-2008/cc731739(v=ws.11)?redirectedfrom=MSDN`.

[6] Keith G Erickson, M Dan Boyer, and David Higgins. Nstx-u advances in real-time deterministic pcie-based internode communication. *Fusion Engineering and Design*, 133:104–109, 2018.

[7] Chuanxiong Guo. RDMA in data centers: Looking back and looking forward. `https://conferences.sigcomm.org/events/apnet2017/slides/cx.pdf`.

[8] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed os for hardware resource disaggregation. *In USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, October 2018.

[9] `https://developers.google.com/protocol-buffers/`.

[10] `https://keras.io/`.