Caladan: A Distributed Meta-OS for Data Center Disaggregation

Lluís Vilanova
Imperial College London

Lina Maudlej
Technion

Matthias Hille
Technische Universität Dresden

Nils Asmussen
Barkhausen Institut

Michael Roitzsch
Barkhausen Institut

Mark Silberstein
Technion

ABSTRACT

Data center resource disaggregation promises cost savings by pooling compute, storage and memory resources into separate, networked nodes. The benefits of this model are clear, but a closer look shows that its full performance and efficiency potential cannot be easily realized. Existing systems use CPUs pervasively to interface arbitrary devices with the network and to orchestrate communication among them, reducing the benefits of disaggregation.

In this paper we present *Caladan*, a novel system with a trusted *universal resource fabric* that interconnects all resources and efficiently offloads the system and application control planes to SmartNICs, freeing server CPUs to execute application logic. Caladan offers three core services: capability-driven distributed name space, virtual devices, and direct inter-device communications. These services are implemented in a trusted *meta-kernel* that executes in per-node SmartNICs. Low-level device drivers running on the commodity host OS are used for setting up accelerators and I/O devices, and exposing them to Caladan. Applications run in a distributed fashion across CPUs and multiple accelerators, which in turn can directly perform I/O, i.e., access files, other accelerators or host services. Our distributed dataflow runtime runs on top of this substrate. It orchestrates the distributed execution, connecting disaggregated resources using data transfers and inter-device communication, while eliminating the performance bottlenecks of the traditional CPU-centric design.

1 INTRODUCTION

Operators are moving towards a data center model where compute, storage and memory resources are disaggregated in order to improve TCO [5, 10, 11, 13, 16, 18, 20, 21, 23, 24, 27, 29]. The primary benefits stem from pooling resources onto separate nodes, which allows more flexible allocation and sharing of hardware, thereby leading to improved utilization and reduced hardware redundancy.

While at the high-level the cost benefits of the disaggregated resource model are clear, current software stacks do not allow realizing the model's full performance and efficiency potential. For example, consider a network service that performs image classification using a machine learning model. Its high-level organization is shown in Figure 1. The application runs as a regular CPU process (*App*). For each classification request, it sends out a read request to the file system (❶ and ❷; running in *FS* and *SSD*) to read the appropriate model from a file (if it has not been cached already); e.g., to classify types of cats or dogs it reads a cat or a dog model respectively. The SSD then sends the file contents into a custom machine learning accelerator to load the model (❸; in *MLAccel*), computes the result and sends it back to the application (❹; in *App*).

We make three primary observations about an idealized disaggregated system. First, achieving high performance in such an application requires direct data and control path between the *SSD* and *MLAccel* resources that does not involve *App*. Second, the *MLAccel* resource runs on a hardware accelerator; the model inference thus does not require any CPU computations. Third, the whole system operates in a form of a dataflow where resources are invoked in a pipeline by their predecessors in the flow graph. While *App* leverages the same logical execution path for each request, it may invoke different hardware resources each time, allocated on-demand by the system to meet the load.

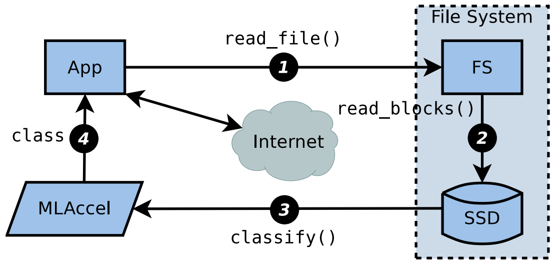

Figure 1: Example machine learning service with a main user application (*App*) processing external requests, a file system service (*FS* and *SSD*), and a machine learning accelerator (*MLAccel*). The numbered arrows (used in the text) correspond to network messages across resources in the data center.

Unfortunately, the idealized design of Figure 1 is not what existing *systems software* supports. First, with *MLAccel*, *FS* and *SSD* all located in different nodes, *each node deploys its own CPUs* to execute device access logic, provisioned to enable efficient control of the devices and scalable RPC interfaces for applications. Second, current software stacks would force centralized management of all the resources (*FS*, *SSD* and *MLAccel*) from the application's CPU (*App*). This 1-to-all control and data path topology would involve more network transfers to run the model inference task compared to the peer-to-peer topology in Figure 1. These two factors imply that a *practical* disaggregated system will quickly lose its performance and TCO benefits; centralized resource management requires an increased number of network messages, and device access logic requires additional CPUs to execute the system's and application's control plane, instead of using them to execute critical application business logic.

The crux of the problem is the inherently ***CPU-centric*** model of existing systems; CPUs play a central role in managing devices and in orchestrating operations across them. This is because only CPUs know a device's wire protocol (i.e., how to operate its bare-metal interfaces). Instead, peer-to-peer interactions assume that every device in the system will know the protocol of every other device, which seems unrealistic. For example, we cannot expect every SSD vendor to implement the logic to push requests into our custom *MLAccel* device.

Ideally, we would expose all disaggregated resources through the network by co-locating networked device access logic with every resource. This would allow efficient implementations of a wealth of use-cases like secure direct device assignment of disaggregated resources (e.g., directly accessing file contents in *SSD* from *App*), application and device RPC interfaces, or even distributed dataflow execution models [30] like the one shown in Figure 1. In fact, distributed dataflow is increasingly used in data center-scale programming frameworks [2, 3, 12, 32], making its optimized execution particularly useful in a disaggregated setting.

In this paper we describe *Caladan*, a data center-scale ***distributed meta-OS*** that eliminates the limitations of existing CPU-centric devices using a novel ***universal resource fabric***. Caladan's fabric interconnects all resources in the system, and exposes generic device access logic through a small set of abstract resource access and messaging primitives. Importantly, it regains the lost benefits of the disaggregated resource model without the need for additional CPUs nor changes in the devices it interconnects.

Caldan's meta-OS takes the role of *securely* managing access to and communication among resources anywhere in the data center, while its users can take advantage of existing OSes and software stacks to deploy their application logic and device-specific drivers. The fabric is provided as a trusted component that executes in per-node SmartNICs [22], to which device access logic is offloaded. All resources are *globally addressed*, routing operations to them regardless of their physical location and resource type. Since multiple tenants can share the same data center, the fabric also has strong *security* guarantees through the use of object capabilities [9], which can be seen as protected resource handles (akin to file descriptors in POSIX systems).

Caladan follows well-known μkernel principles and offers three basic objects that form its universal resource fabric: *memory*, *device slices* and *device requests*. Device slices are a virtual instance of a resource, like a CPU process, a portion of an SSD disk that an application can directly access, or even a virtual function of a PCIe-attached device [26]. Device requests are the only messaging primitive in Caladan, and contain immediate values (e.g., used to represent a device-specific request) and references to other resources or requests using capabilities.

The contributions of this paper are a description of the security aspects and functional design of the Caladan meta-OS (§ 2), and a discussion of three key features that make it possible: (1) the application of μkernel principles to minimize trust and allow complex, higher-level models (§ 3), (2) the design of untrusted device drivers to adapt Caldan's universal resource fabric primitives to device-specific operations (§ 4), and (3) the capability management operations that make Caladan efficient (§ 5).

2 THE CALADAN META-OS

Caladan provides a ***universal resource fabric*** that interconnects all resources in the data center. This fabric is designed to offload critical device access logic, accelerating access to and communication among physical devices and software services alike regardless of their physical location. The fabric can thus accelerate the system's and application's control plane, eliminating the limitations of existing CPU-centric technology.

Caladan is a ***distributed meta-OS***; it allows the coexistence of both existing, full-stack systems like Linux as well as new or experimental bare-metal devices directly used by applications, and the

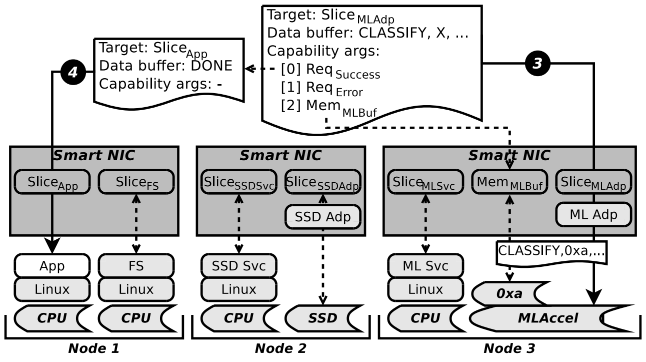
Target: $\text{Slice}_{\text{MLAdp}}$
Data buffer: CLASSIFY, X, ...
Capability args:
[0] $\text{Req}_{\text{Success}}$
[1] $\text{Req}_{\text{Error}}$
[2] $\text{Mem}_{\text{MLBuf}}$
Target: $\text{Slice}_{\text{App}}$
Data buffer: DONE
Capability args: -
4
3
Smart NIC
$\text{Slice}_{\text{App}}$
Slice_{FS}
$\text{Slice}_{\text{SSDSvc}}$
$\text{Slice}_{\text{SSDAdp}}$
SSD Adp
$\text{Slice}_{\text{MLSvc}}$
$\text{Mem}_{\text{MLBuf}}$
$\text{Slice}_{\text{MLAdp}}$
ML Adp
CLASSIFY,0xa,...
App
FS
SSD Svc
ML Svc
Linux
0xa
CPU
SSD
MLAccel
Node 1
Node 2
Node 3

Figure 2: Placement of the hardware and software components of the application in Figure 1; slices correspond to logic components, and numbers to the messages used by them in Figure 1. Dark gray components are part of Caladan's meta-kernel, trusted by the data center operator. Light gray components are higher-level services and interfaces trusted by the application only if used.

critical path operations of its universal resource fabric are offloaded into per-node SmartNICs [22].

Caladan follows well-known ***μkernel principles*** in its design. The data center operator must trust the *Caladan meta-kernel*, Caladan's trusted computing base (TCB), which implements the basic resource abstraction and messaging primitives of Caladan's universal resource fabric. Higher-level components like device drivers, system services (e.g., management of resource placement and scheduling), and even constructs for complex execution models like dataflow reside outside of the meta-kernel. § 3 further develops the reasoning and affordances behind these principles, while § 4 contains a description of physical device drivers in Caladan.

Caladan's universal resource fabric abstracts all resources as objects. Caladan's primitives must be *secure*, since multiple tenants must co-exist in the data center. The meta-kernel provides a sound security model through the use of capabilities [9], unforgeable tokens managed by the meta-kernel that are the sole mechanism to identify and authorize the use of objects in Caladan. Internally, the meta-kernel accesses objects through a *global addressing* scheme, which is used to route operations to objects (including messages) regardless of their physical location and resource type.

Caladan's capabilities act as protected object reference handles that can be communicated through Caladan's messages. This operation is known as *capability delegation* in the literature, making the meta-kernel a distributed object-capability system [14, 31]. Since delegation is expected to be used frequently, the meta-kernel makes some explicit trade-offs in capability management that aim at optimizing this operation, further discussed in § 5.

The rest of this section describes the basic objects and messaging primitives of the Caladan meta-kernel, using Figures 2 and 3 to describe the physical component placement and application programming API for the example in Figure 1. Note that Caladan provides a minimal, trusted *meta-kernel API*, whereas third-party user-level libraries and services are used to implement higher-level APIs like the one used in Figure 3.

```
1   // initialization
2   auto ctx = caladan::context::factory();
3   auto s_fs = filesystem::factory(ctx);
4   auto s_mgr = resource_manager::factory(ctx);
5   auto s_ml = mlaccel::factory(s_mgr);
6   auto ml_buf = s_ml->create_buffer();
7   // read file and train with it
8   ctx->start()
9       ->then(s_fs->read("/path/to/file", ml_buf)),
10      ->on_error([]() {
11          throw std::exception("error␣reading");
12        })
13      ->then(s_ml->classify(ml_buf, ...))
14      ->wait();
```

Figure 3: Example implementation of the application in Figure 1 using Caladan's runtime API.

2.1 Resource Abstraction: Memory and Slices

The Caladan meta-kernel has two objects to represent resources: memory and device slices. Like all objects in Caladan, these can be communicated in messages using capabilities.

Memory objects represent memory anywhere in the data center, encoded as the physical location or a memory block, its size, and access permissions. The meta-kernel offers three operations: (1) create a new memory object from a local memory buffer; (2) create an object with *diminished* permissions and buffer extents (e.g., get a read-only object from a read-write one, or get one pointing to a memory sub-region by increasing its start address or decreasing its size); and (3) copying data across two memory objects.

Device slice objects (or *slices*, for short) represent any non-memory resource, regardless of where it is physically located and whether it is a physical device or a service implemented by software. Slices are akin to processes in a traditional OS; every slice can receive Caladan messages (see § 2.2 below) and has its own *capability namespace*[1] that it can use to operate memory objects and send messages to other slices. Slices thus provide a homogeneous representation of wildly different resources such as CPU processes (e.g., $Slice_{App}$ or $Slice_{FS}$ in Figure 2), an execution context in a GPU (similar to the process abstraction), a portion of an SSD disk that applications can directly access (e.g., $Slice_{SSDAdp}$), or even a virtual function of a PCIe-attached device [26] (e.g., $Slice_{MLAdp}$).

Figure 3 shows how to implement a simple dataflow version of the application in Figure 1 using Caladan's high-level runtime API.

1) Bootstrap: The main application is itself a slice running on a CPU, $Slice_{App}$ in Figure 2, and will first create a communication channel to Caladan's meta-kernel, associating all meta-kernel operations to this slice's capability namespace (`ctx` object in Line 2). Note that the following operations (outside the `caladan` namespace) are implemented outside of Caladan's TCB.

2) File system service: In Line 3, the application gets access to the third-party file system service. This service is itself a Caladan application with its own internal resources ($Slice_{FS}$, $Slice_{SSDSvc}$, and $Slice_{SSDAdp}$, where the latter is the critical-path device access logic serving *SSD* read requests). In this case, `filesystem::factory` uses `ctx` to request access to a slice object that the file system service has registered in advance ($Slice_{FS}$). When $Slice_{App}$ sends a message to $Slice_{FS}$ (❶ in Figure 1), it triggers the service's internal dataflow; finishing the SSD read request in $Slice_{SSDAdp}$ (❷) will trigger the continuation of the application's own dataflow (❸). Note that the global addressing of Caladan's unified resource fabric allows co-locating the main application ($Slice_{App}$) and the file system's core logic ($Slice_{FS}$) in CPUs of the same physical node without neither being aware of it.

3) Resource management service: Similarly, Line 4 gets access to a pre-registered slice that implements a third-party, global resource management service (not shown in Figures 1 and 2). The application then requests a new *MLAccel* slice to the resource manager, Line 5, who responds with the resulting slice objects after applying its physical placement policy. Note that device drivers are usually built from more than one slice object to accommodate separate control and data paths, like $Slice_{MLSvc}$ and $Slice_{MLAdp}$ for the *MLAccel* resource in Figure 2 (more detailed information is provided in § 4).

4) Device-specific resource management: The application then sends a request to the device driver in Line 6 to create a memory buffer in *MLAccel*, which returns it as a memory object (Mem_{MLBuf}).

5) Dataflow computation: Once the application has initialized all resources it needs, it can build a dataflow graph with messages and start a computation based on it (Line 8), which is explained next.

2.2 Messaging: Device Requests

The Caladan meta-kernel provides ***device request objects*** (or *requests*, for short) to send messages across slices. Each request has three elements: (1) a target slice, (2) a raw data buffer to send, and (3) a list of capability arguments to install in the target slice's capability namespace (e.g., to pass memory object references like we would do with pointers). Caladan has a single messaging primitive that sends the data buffer and capability arguments contained in a request to the target slice specified in it.

Requests act as immutable, opaque *continuations* [28] that can be efficiently used to build secure direct resource access primitives. Applications can create new requests from either a slice or from an existing request. Request contents can be partially set when creating a new request, as long as the newly set contents had not been already set in the originating request; i.e., applications can set a subset of the raw data buffer (offset, size and value), as well as specific capability argument indices. For example, we could implement a *FS* service that gives applications direct, read-only access to specific blocks in the *SSD* device. The *FS* would simply create a new request for $Slice_{SSDAdp}$ and set the read command, block number and size on the raw data buffer, and pass that request to the application. The application can later directly access the *SSD* device by simply setting a capability argument for the output memory object, without further interaction with the *FS* service.

Caladan's ***meta-kernel API*** has a single *one-way asynchronous messaging* primitive that simply sends the raw data buffer and the list of capability arguments. Nevertheless, more complex messaging models can be built as a convention on how to use request contents.

Requests can themselves be communicated through the list of capability arguments of a request (they are Caladan objects). Caladan's *untrusted* ***runtime API*** takes advantage of this fact to build a simpler, higher-level interface that implements a *continuation passing style (CPS)* model [28], which is the one shown in Figure 3. By convention,

[1] Also known as a *C-list* in the literature.

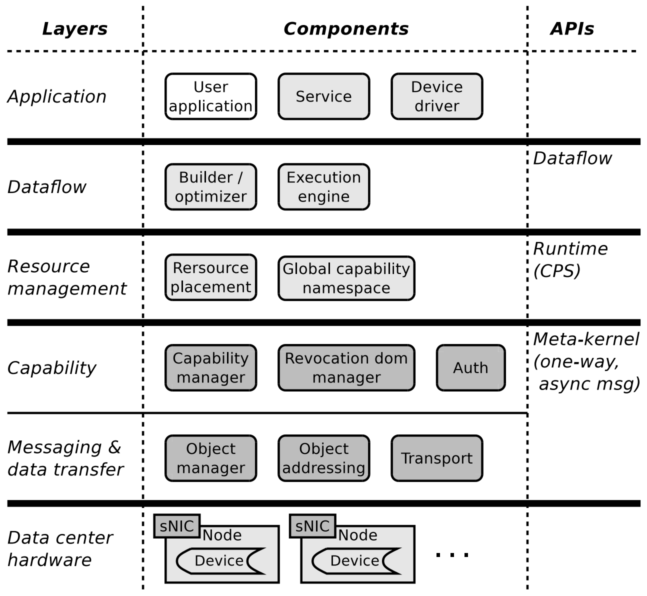

Figure 4: Layers, components and APIs in the Caladan meta-OS. Dark gray components are part of Caladan's meta-kernel, and trusted by the data center operator (the focus of this paper). Light gray components are higher-level services and interfaces trusted by the application only if used.

the runtime API uses the first two capability arguments to pass request objects that act as success and error continuations, respectively.

For example, the application builds a request in Line 9 that tells the file system to read the selected file into a memory object (Mem_{MLBuf}). The specific formatting of the request is established by the client-side service library (method `s_fs->read`), but lines Lines 10 and 13 instruct the runtime API to set the success and error continuations to a request to *MLAccel* and one to the specified application's error handling routine, respectively. When $Slice_{FS}$ receives this request in ❶, it will get these two continuations as capability arguments. If the file does not exist, $Slice_{FS}$ will use the error continuation. If the file exists, it will send a message to the corresponding slice for *SSD* with the requesting application's continuations, such that *SSD* will transparently compose its success/error state with the application.

Note that with the CPS model of Caladan's untrusted runtime API, we can build a more complex ***dataflow API*** on top. We can do so by deploying a dataflow engine service that can be composed, through continuations, with the application's dataflow graph to provide more complex dataflow-specific semantics.

3 μKERNEL PRINCIPLES

We designed Caladan around three μkernel principles: we make an effort to keep the TCB small, we implement high-level communication protocols on top of low-complexity primitives, and we factor out infrastructure components into services.

Figure 4 shows the various layers, components, and API stack in the Caladan meta-OS. We first discuss the Caladan meta-kernel, which defines the TCB for isolation, shown in dark gray in the figure. We then describe how higher-level layers implement communication protocols and infrastructure components on top of the meta-kernel interface, shown in light gray. These services need to be relied upon by applications for availability, but not for isolation.

3.1 The Caladan Meta-Kernel

The Caladan meta-kernel provides a ***universal resource fabric***, a low-level layer dedicated to provide uniform addressing and isolation support for heterogeneous resources. Whereas μkernels implement isolation using a privileged CPU mode, Caladan should support storage and compute devices without native isolation features. But for maintainability reasons we also want a homogeneous implementation of isolation that is device-agnostic.

Our solution — inspired by the M^3 microkernel for heterogeneous manycores [6] — is to place a hardware gatekeeper in front of every resource. We connect every device in the data center to Caladan's universal resource fabric via a SmartNIC, placing it in a privileged position to control communication. Each SmartNIC runs an instance of the Caladan meta-kernel, which maintains a distributed capability system [14]. Therefore, the meta-kernel code and the SmartNIC it runs on constitute the entire TCB with regard to isolation of resource accesses.

3.2 Kernel Responsibilities

Caladan uses capabilities to interact with the meta-kernel, which supports operations for sending messages and for memory accesses (implemented through remote direct memory access — RDMA). Messages, or device requests, realize the control plane and are sent by pushing them to an associated remote device slice. Remote memory accesses via RDMA implement the data plane. Consequently, the three major capability types are device slice capabilities, device request capabilities, and memory capabilities. Some additional capability types needed for setup and capability revocation are explained later in §§ 4 and 5.

As in any capability system, authority is managed in a white-list fashion: a capability is needed to invoke the respective operation on it. Since the SmartNIC is located in a privileged position to oversee all traffic to and from devices, the kernel can ensure that any communication not explicitly allowed by a capability is denied. Traditional data center security mechanisms like VLANs and network ACLs can be augmented or completely replaced by a capability system [7]. At the same time a node's IOMMU [1, 4, 15, 17] can be used to ensure devices on the same node can only interact through the meta-kernel.

This capability-based design thus enables a small isolation TCB with a tight set of responsibilities, but its features are aligned with our goals of a device-agnostic, high-performance infrastructure.

3.3 High-Level Interface

The meta-kernel provides one-way asynchronous message passing, a fitting communication primitive for a low-complexity TCB, but inconvenient to use. We implement the CPS model discussed earlier (§ 2) on top of the kernel primitives, but outside of the isolation TCB. From the meta-kernel's perspective, the CPS model is just a convention that applications have agreed upon. Even if an application violates these conventions and somehow sends a malformed continuation, the isolation properties enforced by the meta-kernel would still hold. These higher-level components must thus be trusted by applications with regard to availability of services, but have no impact whatsoever on isolation between data center clients.

3.4 Remote Services

While high-level messaging primitives are implemented as components outside the meta-kernel, but on the same machine, we implement other infrastructure functionality as remote services reachable through messages. Resource allocation (*resource placement* component in Figure 4) becomes a service that applications call when they require more resources. The service replies with device request capabilities to additional device slices and bills the client accordingly. Capabilities to this and other services can be obtained from the *global capability namespace* service, a simple key-value store mapping strings to capabilities[2].

Sending requests to device slices is akin to invoking a method on an object. The functionality behind these objects can be implemented by hardware accelerators or by software running on a device, but this detail does not change the way services are invoked. Device drivers are, in fact, services that translate messages from one format to another. Device slices thus represent the endpoints to talk to any kind of service in Caladan. Requests constitute a generic invocation of such a service. Together, they implement mechanisms similar to processes and inter-process communication in a μkernel, abstracting services of all kinds behind a common interface.

4 DEVICE DRIVER SUPPORT

Physical devices can be attached into Caladan's universal resource fabric through a device driver component that acts as any other Caladan-capable application. As explained, regular Caladan applications can use the raw data buffer and capability arguments of a device request arbitrarily. To bridge the gap between Caladan's device requests, the conventions under which they are used, and device-specific protocols, device drivers are split into three components: an application-side *driver library*, a *device service*, and a *device adaptor*.

A ***driver library*** is a regular third-party library that applications can link with to avoid dealing with the low-level details of each device's request formats and Caladan's low-level primitives. Each of the services in Figure 3 is accessed through such driver libraries (e.g., Lines 3 to 5 and all the methods called on these objects).

A ***device service*** is a regular, Caladan-aware, CPU application that initializes the physical device (or devices) it manages. Figure 2 shows two device services, $Slice_{SSDSvc}$ and $Slice_{MLSvc}$, which execute in a CPU co-located with the physical device they manage (*SSD* and *MLAccel*, respectively). After initialization, the service will create a series of request objects for each of the operations it exports to its client applications, including slice creation for the corresponding physical device. In the case of Caladan's runtime API, the service will register a slice creation request with the resource placement service, together with the type and properties of the device it is serving. When a client application requests a new slice for a specific device, Line 5 in Figure 3, the resource placement service will invoke the registered device service request. In turn, the device service will create a new device slice object, perform the necessary device-specific operations for the new slice (e.g., create new request/response queue pairs to the device), and return a series of request objects to the client, representing the operations the client can invoke for the new device slice.

A ***device adaptor*** implements the accelerated data path operations for a slice of a physical device. Note that device services alone are sufficient to incorporate a physical device into Caladan's universal resource fabric; the service can demultiplex all per-slice requests and adapt them to the device-specific protocol. Nevertheless, this puts inefficient CPUs on the critical path and reduces the benefits of disaggregation. To solve this problem, when a device service creates a device slice, it can associate it with a device adaptor that will execute inside the SmartNIC and process all requests directed to that slice. For example, when the $Slice_{SSDAdp}$ receives a disk read request, it will: (1) verify that it has received a capability argument for an output buffer to place the read data in; (2) create a local temporary buffer for the read contents; (3) create a device-specific SSD read request, pointing to the temporary buffer; (4) poll the SSD queues for a read response; (5) copy the the contents of the temporary buffer onto the previous output memory capability; and (6) invoke the success continuation of the processed device request. The device driver implementer has complete freedom on how to handle each request, since both the device service and adaptor are slices themselves that can communicate via Caladan.

In the current prototype, a device adaptor is a shared library that the service loads into the meta-kernel running on the SmartNIC, and is directly triggered when the associated slice receives a request. Even if device drivers are outside Caladan's TCB, the operator must trust the device adaptors (Caladan controls which applications are authorized to load adaptors). One could envision more sophisticated mechanisms to isolate adaptors using protected libraries in the meta-kernel [25], or trusted, hardware-accelerated intermediate code representations like eBPF [19].

5 CAPABILITY MANAGEMENT

Caladan uses distributed object capabilities for fine-grained authorization. As mentioned in § 2, device slices can delegate (i.e., send to a receiver) and obtain capabilities via requests. Delegation has thus to be highly efficient, since it is on the critical path of request processing, a high-frequency operation in Caladan. Caladan must also support capability revocation; before a device slice object can be destroyed, Caladan must ensure there are no capabilities referencing it anywhere in the data center. This means delegated capabilities need to be tracked to support later revocation, a costly operation when performed on every delegation. Hence, the fundamental design principle for Caladan's capability system is to provide fast delegation at the expense of potentially sacrificing the speed and granularity of revocation.

Caladan ensures capabilities are unforgeable by employing *partitioned capabilities* where the meta-kernel acts as the privileged component; all capability operations are initiated by applications, but are executed by the meta-kernel running on the SmartNICs.

5.1 Distributed Capabilities

Caladan's universal resource fabric connects various nodes in the data center; capabilities are thus distributed across nodes and each node maintains its own local set of capabilities, similar to other distributed capability systems [14, 31].

[2]Long random strings as keys are a common technique used to provide security in this type of services, typically referred to as password capabilities.

Revocation in Caladan's distributed capability system requires recursively tracking the delegation of capabilities across device slices, which are the smallest subject for isolation (as opposed to applications, which can encompass multiple nodes in Caladan's target disaggregated architecture).

5.2 Fast Delegation

To achieve fast delegation in a distributed system we need to consider what happens during delegation. There are two essential steps: (1) the capability is copied from one node to another, and (2) the delegation is recorded in a *capability derivation tree* that tracks the delegation origin of every capability in the system. The purpose of the first step is to achieve sharing of access rights between slices (within and across nodes). The second operation is done to enable the revocation of the capabilities later on. While the cost of capability copies is determined by implementation details like the size of a capability and the overhead to serialize and send it over the wire, the cost of tracking capability delegation depends on the chosen strategy for tracking and revoking capabilities.

A capability derivation tree can become a large data structure, subject to frequent modification during capability delegation and revocation operations. This is because delegation is tracked for each capability to enable fine-granular selective revocation [8]. Our hypothesis is that revocation does not always require the fine granularity of selective capability revocation, since revocation is often used to clean-up state after application termination or after cancelling connection to an entire service, which involves revoking multiple capabilities in one go. Therefore, we choose to *trade-off granularity for performance.*

5.3 Revocation Domains

Caladan assigns capabilities into buckets called ***revocation domains***, which offer a novel approach for efficient capability revocation at a coarse granularity. Revocation domains are objects in Caladan's meta-kernel, and delineate the scope of a revocation operation, meaning that all capabilities of one revocation domain are collectively revoked. Hence, Caladan allows creating new revocation domains and assigning capabilities to them. Revocation granularity is thus defined by a revocation domain; revocation is invoked on a revocation domain and not on the capabilities which are going to be revoked.[3]

Revocation domains are designed around three basic rules:

(1) Each capability is part of at least one revocation domain.
(2) Each device slice has at least one revocation domain.
(3) A capability can be in multiple revocation domains.

A capability that is part of multiple revocation domains is revoked whenever one of its domains is revoked. This enables revocation of shared resources either when one application terminates or when a revocation domain, created for sharing certain capabilities, is explicitly revoked. For example, imagine a file system service that delegates capabilities to directly access a storage device. The service will create one revocation domain per client application, and will assign the delegated capabilities to the appropriate domain. When the service wants to revoke access for a specific client application, it will simply revoke the appropriate revocation domain.

Revocation domains are designed to avoid expensive tracking of capability delegations. Without revocation domains (such as in SemperOS [14]), each delegation needs to create a link between the existing capability at the sender side and the new capability at the receiver side. This tracking requires coordination between the two involved nodes, leading to additional messaging for every delegation. Revocation domains avoid this coordination by only keeping a record of the revocation domains assigned to every delegated capability and the target nodes of that delegation operation. A delegation of a second capability on the same revocation domain will thus not need to update this record. When a revocation domain is revoked, only the nodes to which capabilities from this revocation domain have been delegated are informed. This technique thus prevents expensive broadcast operations and frequent updates to the data structure tracking capability delegations.

Note that in this scheme, applications can control the granularity of revocations by creating new revocation domains, including the use of one revocation domain per capability for the finest possible granularity when needed. However, we believe that applications will primarily revoke capabilities in a coarse-grained fashion when, for example, disconnecting from a service or shutting down.

In summary, the concept of revocation domains has sufficient flexibility to implement fine-grained selective revocation, but also offers a lot of optimization potential due to the reduced tracking overhead, thereby enabling fast delegation in Caladan's target distributed setting.

6 CONCLUSIONS

Data center resource disaggregation promises large TCO improvements, but existing systems preclude some of these benefits due to their inherently CPU-centric model. As a result, existing systems are bound to using multiple CPUs in the critical path of applications to interface every compute and storage device with the network, as well as need to centrally orchestrate communication across devices. These two factors diminish the performance and TCO benefits of an ideal disaggregated resource data center.

In this paper we present Caladan, a system that provides a *universal resource fabric* that uniformly interconnects all resources in the data center, regardless of their hardware and software nature. Caladan's fabric offloads the system's and application's control plane, freeing up resources for critical application business logic.

Caladan's universal resource fabric offers direct access to both software services and hardware devices without CPU mediation through a trusted *meta-kernel* that executes in SmartNICs deployed on each node. The meta-kernel is in charge of providing low-level secure primitives to access resources, enforcing isolation across tenants. It also follows well-known μkernel principles to provide higher-level features such as device drivers, resource management, third-party services, or even dataflow execution models outside of the trusted computing base, and allows applications and device drivers to use existing system stacks.

The design of the Caladan meta-kernel presented in this paper offers a glimpse at the basic tenets of Caladan: its core abstractions and primitives and the trade-offs that have been taken to deal with system security, efficiency and complexity. We believe these ideas will serve to spur productive discussions in establishing a solid foundation for future disaggregated data center designs, knowing that this is a long road that will require large concerted efforts.

[3]Strictly speaking, the capability pointing to a revocation domain is invoked.

REFERENCES

[1] AMD Inc. *AMD IOMMUcqrchitectural specification, rev 2.00*, March 2011.

[2] Apache. Apache airflow. https://airflow.apache.org.

[3] Apache. Apache beam. https://beam.apache.org.

[4] ARM Holdings. *ARM system memory management unit architecture specification — SMMU architecture version 2.0*, 2013.

[5] Krste Asanović. Firebox: A hardware building block for 2020 warehouse-scale computers. In *USENIX Conf. on File and Storage Technologies (FAST)*, February 2014.

[6] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. M3: A hardware/operating-system co-design to tame heterogeneous manycores. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2016.

[7] Anton Burtsev, David Johnson, Josh Kunz, Eric Eide, and Jacobus Van der Merwe. CapNet: Security and least authority in a capability-enabled cloud. In *Symposium on Cloud Computing (SoCC)*, September 2017.

[8] Ellis Cohen and David Jefferson. Protection in the Hydra operating system. In *Symposium on Operating Systems Principles (SOSP)*, November 1975.

[9] Jack Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, March 1966.

[10] José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *Intl. Conf. on High Performance Computing and Simulation (HPCS)*, June 2010.

[11] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. Beyond processor-centric operating systems. In *Workshop on Hot Topics in Operating Systems (HotOS)*, May 2015.

[12] Fission. Serverless workflows for kubernetes. https://fission.io.

[13] Google. Cloud TPU. https://cloud.google.com/tpu.

[14] Matthias Hille, Nils Asmussen, Pramod Bhatotia, and Hermann Härtig. SemperOS: A distributed capability system. In *USENIX Annual Technical Conference (ATC)*, July 2019.

[15] IBM Corporation. *PowerLinux servers — 64-bit DMA*, May 2014.

[16] Intel. Intel rack scale design architecture.

[17] Intel Corporation. *Intel virtualization technology for directed I/O, architecture specification, rev 2.2*, September 2013.

[18] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends. Rack-scale disaggregated cloud data centers: The dredbox project vision. In *Design, Automation Test in Europe Conf. Exhibition (DATE)*, March 2016.

[19] Jakub Kicinski and Nicolaas Viljoen. eBPF hardware offload to SmartNICs: cls bpf and XDP. In *Technical Conference on Linux Networking (NetDev)*, October 2016.

[20] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *European Conference on Computer Systems (EuroSys)*, April 2016.

[21] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Intl. Symp. on Computer Architecture (ISCA)*, June 2009.

[22] Mellanox. Bluefield multicore system on chip, 2018.

[23] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to zombieland: Practical and energy-efficient memory disaggregation in a datacenter. In *European Conference on Computer Systems (EuroSys)*, April 2018.

[24] NVM Express. *NVM Express over Fabrics*, July 2018. Revision 1.0a.

[25] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *USENIX Annual Technical Conference (ATC)*, July 2019.

[26] PCI-SIG. *Single Root I/O virtualization and sharing specification*, revision 1.1 edition, January 2010.

[27] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, Doug Burger, Jim Larus, Prashanth Gopi Gopal, and Simon Pope. A reconfigurable fabric for accelerating large-scale datacenter services. In *Intl. Symp. on Computer Architecture (ISCA)*, June 2014.

[28] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation - Special issue on continuations—part I*, November 1993.

[29] 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.

[30] R. M. Shapiro and H. Saint. The representation of algorithms as cyclic partial orderings. Technical report, New York: Meta Information Applications, Inc., July 1971.

[31] Andrew Tanenbaum, Sape J. Mullender, and Robbert Van Renesse. Using sparse capabilities in a distributed operating system. In *International Conference on Distributed Computing Systems (ICDCS)*, May 1986.

[32] TensorFlow. Generating big datasets with apache beam. https://www.tensorflow.org/datasets/beam_datasets.