Extending the Mochi Methodology to Enable Dynamic HPC Data Services

Matthieu Dorier,[*] Philip Carns,[*] Robert Ross,[*] Shane Snyder[*], Rob Latham[*], Amal Gueroudji[*],
George Amvrosiadis[†], Chuck Cranor[†], Jerome Soumagne[‡]
[*]*Argonne National Laboratory* [†]*Carnegie Mellon University* [‡]*Intel Corporation*

***Abstract*—High-performance computing (HPC) applications and workflows are increasingly making use of custom data services to complement traditional parallel file systems with fast transient data management capabilities tailored to application-specific needs. In the Mochi project we provide methodologies and tools that enable rapid development of custom HPC data services, including a collection of composable software components that can be combined to build complex distributed data services. Our initial version of Mochi targeted data services deployed with *static* configurations with a fixed number of nodes and minimal fault tolerance. However, there is a growing need for *dynamic* services that can adapt while running in response to changing workloads and system conditions.**

In this paper we present our work to extend the Mochi architecture to support the development of dynamic data services. We achieve this by providing new Mochi components that support unified bootstrapping and online reconfiguration, fault detection, monitoring, and consensus. We also provide a methodology for deriving service-wide resilience from the resilience of each of the service's components.

1. Introduction

High-performance computing (HPC) systems execute increasingly complex workflows that enable scientific discoveries in many areas, from molecular dynamics to astrophysics, by employing diverse computational techniques including numerical methods, data mining, and artificial intelligence. This diversity calls for an approach to data management that is tailored to each specific use case as opposed to the one-size-fits-all solution offered by parallel file systems. Because of the additional layers of customization, however, tailored *HPC data services* can be difficult for research teams to develop and put into production.

In the Mochi project we aim to democratize methodologies and tools for the rapid development of custom HPC data services. We offer a collection of interoperable software components that can be assembled together to build complex distributed data services. A growing community of researchers rely on Mochi to develop systems ranging from transient filesystems and data management middleware to in situ analysis and performance monitoring systems. These services are typically deployed in a dedicated static set of nodes alongside the application they support.

Similar to cloud services, HPC data services need to be able to more dynamically adapt to changes in access patterns from the workflows that use them. For example, the high-energy physics NOvA workflow [1], for which the HEPnOS data service was developed [2], presents steps with vastly different I/O patterns. Our work in autotuning HEPnOS [3] showed that the best configuration of the service for one step of the workflow is not necessarily the best for other steps. Rather than compromising and using a static configuration of HEPnOS that provides satisfactory overall performance, a dynamic version of HEPnOS that reconfigures at run time for each individual step's I/O pattern could be used. Other recent studies have confirmed similarly diverse and dynamic I/O demands within other workflow environments [4], [5].

We believe that many applications would benefit from data services designed to be dynamic. We define a dynamic data service to have four key properties. First, dynamic services should support *performance introspection* capabilities to drive configuration decisions based on performance feedback. Second, dynamic services should allow *online reconfiguration* so that the application's data services can be reconfigured without service interruptions. Third, the service must be *elastic*. Elastic services are able to scale up or down with workload demand. We expect elastic data services to pair well with high-level HPC resource managers such as Flux [6] that support the elastic allocation of cluster resources such as compute and power. Fourth, dynamic services should be *resilient* to failures.

Developing dynamic data services can be daunting. Ideally, dynamic properties should be enabled incrementally in existing static data services without having to redesign or refactor existing code. In this paper we extend the Mochi methodology and provide new components to enable data service developers to rapidly develop dynamic data services. Our contributions are threefold:

- We derive requirements for the rapid development of dynamic services (§2.3).
- We define a methodology for the design and composition of dynamic components (§3 to §7).
- We provide a set of reusable components for bootstrapping, online reconfiguration, data migration, consensus, and failure detection that we will make available to users to ease data service development.

2. Motivation

In this section we provide an overview of the Mochi project, its adoption in the community, and its methodology [7] for building static data services. We then identify additional requirements needed to extend Mochi to support dynamic services.

2.1. The Mochi project and community adoption

The Mochi project [8] was formed in 2015 as a collaboration of Argonne National Laboratory, Carnegie Mellon University, Los Alamos National Laboratory, and The HDF Group with the goal of facilitating rapid development of distributed data services in support of scientific HPC applications. Such services have the potential to radically improve productivity and usability but historically have been challenging to develop because of HPC storage complexity and scientific use-case diversity.

Mochi addresses the challenges of developing custom HPC data services using *composability*. Mochi provides a collection of robust, reusable, modular, and connectable data management components and microservices along with a methodology [7] for composing them into specialized distributed data services. Agile, specialized data services can be rapidly constructed to cater to the data management needs of a particular science domain or computing platform while still effectively leveraging the underlying capabilities of cutting-edge HPC hardware.

Mochi has proven successful in the broader community, with key elements serving as the foundation for a diverse array of data services including transient / burst buffer file systems [9], [10], [11], object storage systems and middleware [12], [13], application workflow coupling frameworks [14], in situ analysis tools [15], and performance monitoring systems [16], [17], [18]. This breadth of adoption has revealed common service development patterns and guided the evolution of the Mochi methodology over time. These use cases also provide numerous examples of unpredictable or bursty workloads that would further benefit from dynamic online adaptation to prevailing workloads and system conditions.

2.2. Lessons learned: static service methodology

Mochi defines a methodology for rapidly developing static HPC data services tailored to specific applications [7]. This methodology involves four steps. First, establish user requirements such as data models, access patterns, and semantics. Second, translate the user requirements into service requirements to determine data organization, hardening policies, and user interface. Next, develop or reuse building blocks needed to implement these service requirements. Fourth, compose these building blocks into a complete user-facing service. Repeated use of this methodology led us to a number of lessons learned that we summarize below.

Flexibility vs. productivity. Flexibility is critical to innovation but can lead to inconsistency across components and loss of productivity if developers are presented with too many initial options. This situation led us to standardize best practices for service development. We accomplished this by creating microservice templates [19] and used them to refactor the most popular core components: key-value store and blob store. Feedback from external users indicate that using these templates saves several weeks of development [15].

Configurable composability. Our initial methodology included writing "glue code" to connect components. However, composability is more powerful if it can be accomplished at run time without glue code recompilation. This allows faster exploration of the configuration space and provides users with an easy way to share their configuration for reproducing experiments or diagnosing performance issues. To enable runtime composability, we implemented Mochi's Bedrock dynamic bootstrapping and configuration component.

Community support. The diversity of HPC platforms and applications and the rapid evolution of technology make centralized community support a bottleneck to adoption. Mochi users must be able to rapidly diagnose behavioral and performance problems on their own in order to accelerate adoption. Thus, we created easy-to-install Mochi packages (e.g., Spack recipes) for high-profile platforms, command-line diagnostic tools, and monitoring infrastructure.

2.3. New requirements: toward dynamic services

Based on our experience to date, we have identified four core requirements for composable dynamic services.

Performance introspection. Performance introspection is the ability of a service to provide fine-grained performance information and statistics at any time. It is a prerequisite to any service dynamicity. It provides the empirical data necessary for informed decisions about changes made to the service. Unified performance introspection within the Mochi methodology will enable all Mochi components to automatically participate in the analytics that drive dynamic service decisions.

Online reconfiguration. Online reconfiguration is the ability to enact live configuration changes in a running service. The existing Mochi methodology already provides a hierarchical service configuration mechanism that encompasses tuning parameters for each component as well as the mapping of those components to local hardware resources. To enable dynamic services, the Mochi methodology must go one step further and provide the means for online modification of this representation so that configuration changes can be enacted immediately without taking the service offline. This will make it possible to alter component mapping, provisioning, and prioritization across hardware resources in response to workload changes as part of the dynamic optimization process.

Elasticity. Elasticity is the ability to dynamically add resources to or remove them from a running service. Online reconfiguration is a prerequisite for elasticity. Elastic services must also be able to migrate data within or across nodes and modify load balancing in response to reconfigurations. These capabilities were not present in the original static Mochi methodology, but they are crucial to enabling services to grow or shrink in scale in response to prevailing conditions.

Resilience. Resilience is the ability to continue to provide service in the face of failures. This includes reacting to both transient failures (e.g., a service process crashed, but its data is still available in local storage) and permanent failures (e.g., hardware faults resulting in data loss). Resilience extends elasticity by ensuring persistent and consistent data protection and by providing mechanisms to detect faults. It also requires responding to faults at the scale of the entire service (e.g., rebalancing data across nodes or spinning up capabilities on additional nodes). Incorporating resilience into the Mochi methodology enables specialized data services to provide the same level of robust availability that users expect from traditional systemwide services.

These four requirements are naturally ordered from a dependency perspective, each requiring the previous one as a prerequisite. Conveniently, they are also ordered in terms of the breadth of dynamicity they enable and how difficult they are to implement. The requirements should be enabled in a *composable* manner *with the least engineering impact* on existing components and minimal *tight coupling* between components.

As an example, consider the design of a resilient key-value store built by using multiple instances of Yokan (Mochi's node-based key-value store). A consensus algorithm such as RAFT [20] is needed to provide data consistency for key-value pairs replicated across the nodes running Yokan. For this we provide Mochi-RAFT, a Mochi-based implementation of RAFT. In a composable design, individual Yokan instances are unaware of their database being RAFT-replicated across nodes, while Mochi-RAFT itself does not need to know that the commands it logs represent Yokan key-value pairs. Yokan database replication should be transparent to end users. In correctly designed components it will be relatively simple to enable online reconfiguration, elasticity, and resilience in an existing Mochi-based service whose design was originally static.

3. Mochi component design

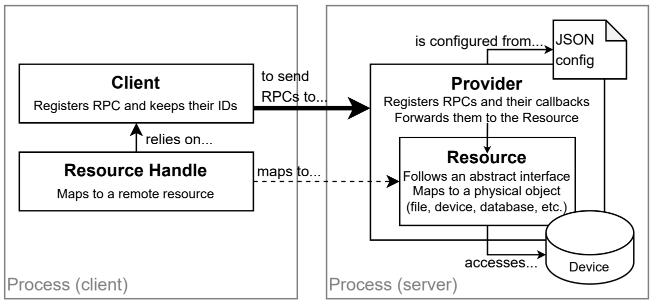
Before diving into the requirements of dynamic services listed in Section 2.3, this section illustrates the architecture of a typical Mochi component.

3.1. Anatomy of a Mochi component

Mochi emphasizes separating capabilities into independent components that can be composed together and share a common threading and networking runtime within the same process. Such capabilities include managing a key-value database, a raw storage device, an interpreter for a scripting language, and so on.

Figure 1 shows the anatomy of a typical Mochi component. This component provides two libraries. The server library lets users create *providers* that manage and provide remote access to a *resource*. Since multiple providers can live in the same process, they are uniquely identified by a *provider ID*. A provider and its resource are configured by using a JSON-formatted string.

A *resource* will generally follow an abstract interface so that the functionality provided by the component can be implemented in various ways. For instance, Yokan provides key-value storage on top of backends such as RocksDB, LevelDB, and Berkeley DB.



Client
Registers RPC and keeps their IDs
to send RPCs to...
relies on...
Resource Handle
Maps to a remote resource
maps to...
is configured from...
JSON config
Provider
Registers RPCs and their callbacks
Forwards them to the Resource
Resource
Follows an abstract interface
Maps to a physical object (file, device, database, etc.)
accesses...
Device
Process (client)
Process (server)

Figure 1. Anatomy of a Mochi component.

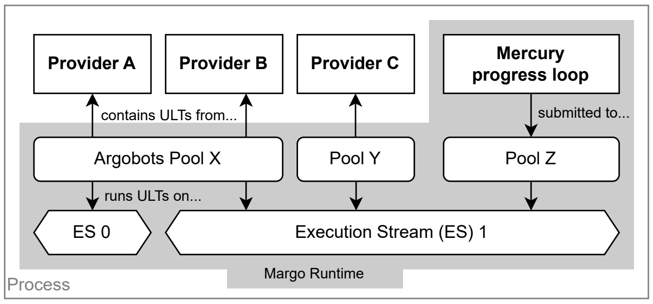


Figure 2. Providers within a process sharing a common Margo runtime.

The client library lets users instantiate *resource handles*, which map to a remote resource by encapsulating the address and provider ID of the provider holding that resource. It provides an API to access the resource, for instance putting and getting key-value pairs in the case of Yokan.

3.2. Runtime sharing and dependencies

Components within a service use remote procedure calls (RPCs) to interact. The Margo runtime, based on Mercury [21] and Argobots [22], takes care of running these RPCs in user-level threads (ULTs), turning them into function calls if the source and destination are on the same process, using shared memory if they are on the same node, and relying on the high-performance network transport across nodes.

Figure 2 shows an example of a process containing three providers. These providers are associated with pools, which contain ULTs. Execution streams (or ESs, the Argobots term for the operating system threads that will execute ULTs), mapped to cores, pull ULTs from the pools they are associated with and execute them. Arbitrarily complex mappings from providers to cores can be set up thanks to Argobots. In the present example, the network progress loop runs exclusively on ES 1 through Pool Z. Upon receiving an RPC, it submits a ULT to either Pool X if the RPC targets Provider A or B, or Pool Y if it targets Provider C.

Composition in Mochi is achieved by having providers depend on resource handles pointing to other providers. For example, one can imagine a Mochi component M managing "datasets" by storing their metadata in a key-value store (managed by the Yokan component) and their data in a blob storage target (managed by the Warabi component). This component M could be further composed with Mochi's embedded language interpreter component (Poesie), to execute scripts on datasets or with Colza [23] to run in situ visualization pipelines on them. This flexible design methodology allows components to evolve independently, to be reused across many services, and to benefit from community input.

In the the next four sections we delve into the four requirements for dynamic services identified in 2.1.

4. Performance introspection

Observation 1: *Performance monitoring plays an important role in motivating any reconfiguration and elasticity decision. It should therefore be available at no engineering cost to any component and should expose performance information in a unified manner to make its analysis as simple as possible.*

Design principle: a customizable monitoring infrastructure provided by the runtime. Since most of the Mochi components rely on Margo, it was natural to implement performance monitoring at the Margo level. Margo has knowledge of all the RPCs being sent and received and all the RDMA operations being carried out, as well as the context in which they are performed. We developed a customizable monitoring infrastructure in Margo to capture the relative timings and duration of all relevant parts of an RPC, from the serialization of input and output data to the scheduling of ULTs and potential RDMA transfers. This infrastructure lets users inject callbacks to be invoked at various points in the lifetime of an RPC, for example when the RPC is sent, when it is received, and when it starts and stops executing. The default implementation of this monitoring system captures statistics and outputs them as JSON when shutting down the service. It also makes them available at run time via an API. It periodically tracks the number of in-flight RPCs and the sizes of user-level thread pools so as to provide users with a complete view of what is happening inside a Mochi process at any time. Using this monitoring infrastructure allows users to precisely pinpoint the source of performance degradations: while an upper-level component would measure only that an RPC is slow, this Margo-level monitoring can tell us whether data serialization, RDMA transfers, pool scheduling decisions, or some other bottleneck is at fault. This monitoring inherits and improves on many of the works done by Ramesh et al. with SYMBIOSYS [17] and SYMBIOMON [18], our early take on composable monitoring for Mochi.

An example of statistics is shown in Listing 1. Such statistics are available to any Mochi-based service at no engineering cost. It has helped us countless times diagnose the root cause of performance issues in our users' services.

```
"rpcs": {
  "65535:65535:2924675071:65535": {
    "rpc_id": 2924675071,
    "provider_id": 65535,
    "parent_rpc_id": 65535,
    "parent_provider_id": 65535,
    "name": "echo",
    "origin": { ... },
    "target": {
      "received from na+sm://28885-0": { ...
        "ult": {
          "duration": {
            "num":3,
            "avg":0.083172719,
            "max":0.134156227,
            ...
          }, ...
```

Listing 1. Fragment of performance monitoring data from Margo showing the ULT duration for an "echo" RPC received by the process from a client at address `na+sm://28885-0`. Note that these statistics also include the context (parent RPC and parent provider), in which an RPC was issued, as well as who it was received from or sent to, allowing fine-grain analysis.

5. Online reconfiguration

Observation 2: *Online reconfiguration involves modifying the way these components map to physical resources (e.g., hardware threads and cores) and how their operations are prioritized.*

Design principle: a more dynamic run time. Figure 2 shows that, in Mochi terms, this observation translates into being able to add and remove pools and ESs in the Margo run time, which all components share. Just like performance introspection, it is therefore natural that this property be implemented in Margo.

The Margo run time can be initialized with a JSON configuration string describing the initial setup of pools and ESs (e.g., Listing 2). At run time, this setup can be queried with functions such as `margo_find_pool_by_name` and modified by using functions such as `margo_add_pool_from_json`. Margo ensures that the changes are always valid, such as not allowing adding multiple pools with the same name or removing a pool that is in use by an ES.

```
{ "argobots": {
    "pools": [ { "name": "MyPoolX",
                 "type": "fifo_wait"
                 "access": "mpmc"
               }, ...
             ],
    "xstreams": [ { "name": "MyES0",
                    "scheduler": {
                         "type": "basic",
                         "pools": ["PoolX"]
                       }
                }, ...
```

Listing 2. Example Margo configuration.

Observation 3: *Once the mapping of components to hardware resources is made dynamic, online reconfiguration can enable starting and stopping components in a given process.*

The functionalities of a Mochi service are implemented in providers. The next step after being able to add/remove pools and ESs is to be able to add/remove providers. Margo has no knowledge of the providers that use each pool, however, and cannot ensure that we are not leaving a provider without its pool during a reconfiguration. Hence, another component is needed to keep track of the providers running in a process and their mapping to thread pools.

Design principle: a "provider of providers." Bedrock is a component meant to manage other providers running in a Mochi process. It follows the same architecture as shown in Figure 1, with a server and a client library, but the "resource" it manages is the configuration of the process it runs on. Bedrock takes as input a JSON file describing the desired composition, from the list of providers and their configuration, down to the Argobots setup in Margo. It checks the validity of this configuration and takes care of resolving dependencies between providers, both within and across processes.

```
{ "margo": { ... },
  "libraries": { "A": "libcomponent_a.so" },
  "providers": [
    { "name": "myProviderA",
      "type": "A",
      "pool": "MyPoolX",
      "config": {...},
      "dependencies": {...}
    }, ...
```

Listing 3. Example Bedrock configuration.

Listing 3 shows an example of Bedrock configuration. The `libraries` section tells Bedrock which libraries to load to know how to instantiate a provider of type "A." This library contains a structure of function pointers that Bedrock will call to instantiate providers, clients, and resource handles, as well as to obtain their configuration. The `providers` section then lists providers to instantiate.

Bedrock's bootstrapping mechanism is already a powerful way to set up Mochi services without the need for glue code. Its configuration format enables the evaluation of potential configurations without recompiling code. Thus, it provides an effective way to perform parameter space explorations in Mochi [3]. The configuration can also easily be shared with the community to diagnose issues and bugs.

Bedrock's client library enables remote access to its functionalities, providing a standardized entry point for querying and altering a process's configuration at run time. Querying the configuration can be done by using Jx9, a lightweight, embeddable scripting language designed to handle queries on JSON documents. Listing 4 shows an example query that returns the names of all the providers in the process that receives it. Jx9 can also be used as input in place of JSON, allowing *parameterized* configurations.

```
$result = [];
foreach($__config__.providers as $p) {
    array_push($result, $p.name); }
return $result;
```

Listing 4. Example of Bedrock query in Jx9 to list the names of all the providers in a process.,basicstyle=

To modify the configuration of a process, Bedrock's client library provides a C++ API, exemplified in Listing 5. This API allows adding and removing pools, ESs, and providers. Bedrock will check that the resulting configuration remains valid before carrying out the changes. This includes carrying these checks across Bedrock processes and handling consistency. For example, if a client c_1 requests the creation of a provider p_1 in node n_1 and p_1 depends on another provider p_2 in node n_2, if a client c_2 requests the destruction of p_2 at the same time, then either c_1's or c_2's request will success, but not both, leaving the system in a state where either both p_1 and p_2 exist (with the former having a dependency on the latter) or p_2 was destroyed and p_1 was not created.

```
bedrock::Client client{...};
bedrock::ServiceHandle p =
  client.makeServiceHandle(address);
// p is handle to remote process. Following
// RPC calls change its configuration.
p.addPool(jsonPoolConfig);
p.removePool("MyPoolX");
p.loadModule("B", "libcomponent_b.so");
p.startProvider("myProviderB", "B",...);
```

Listing 5. Simplified overview of Bedrock's C++ API to manipulate a process's configuration remotely and at run time.

6. Elasticity

Bedrock provides a good basis for starting new processes with a specific configuration and reconfiguring existing processes at run time. However, elasticity in a data service goes beyond just spinning up new processes.

Observation 4: *Adding new nodes to a service provides an opportunity for some of the existing data to be redistributed. Removing nodes first requires their data to be sent to remaining nodes. Data migration is therefore a requirement for elasticity.*

Most data managed by Mochi components resides in files stored in a local storage device. Migrating a resource from a node to another often comes down to transferring files between two nodes.

Design principle: a component to migrate files. REMI (Mochi's REsource MIgration component) was designed to handle such file migrations. It does so either by memory mapping the files and using RDMA to transfer the data or by using a series of RPCs to send the files in chunks. The former method is more efficient for large files. The latter is more efficient when sending multiple small files, since they can be packed together into larger chunks and the transfer of chunks can be pipelined.

Thanks to REMI, the migration of a component can be reduced to the migration of its files to a new location, the instantiation of a new provider to manage these files, and the removal of the provider previously holding the files.

Observation 5: *Component migrations need to be triggered remotely and controlled for correctness.*

Design principle: migration control and triggers. Since Bedrock manages components within each process, it is the natural place to enable control and trigger of component migration. In their Bedrock module, components can declare a dependency on a REMI provider to be able to carry out such a migration and expose a `migrate` function pointer for Bedrock to call. The migration of a provider hence becomes an operation managed by Bedrock that can be triggered remotely and checked for consistencies. For instance, Bedrock can assert that migrating a provider will not break dependencies.

Observation 6: *Rebalancing data requires knowing where the data is and what the best placement should be.*

With a composable mechanism in Mochi to migrate a provider to another node, the next step is to enable rebalancing decisions in a composable manner as well.

Design principle: externalized rebalancing decisions. In [24] we presented Pufferscale, a Mochi component that implements heuristics to decide which pieces of data to migrate and where in order to achieve load balance (balance of accesses to the data), data balance (balance of their volume on each node), rebalancing time, or a compromise between these three objectives. Pufferscale is a great example of a composable design for an elastic storage service: Pufferscale does not require any knowledge of the nature of the resources being migrated or how they will be migrated. It simply works out a rebalancing plan and carries it out by calling functions provided via dependency injection. We intend to integrate this as a core Bedrock capability based on this proof of concept, using the performance introspection tools presented in Section 4 to guide load rebalancing.

Observation 7: *If the nodes used by a data service can change, we need a mechanism to track the service location at all times.*

Another challenge in supporting elasticity is keeping the applications that use the service up to date with the service's location (i.e., the list of addresses of the nodes making up the service).

Design principle: dynamic group membership. To address this issue, we provide the SSG group-membership component. SSG maintains a dynamic view of a group of processes and allows this view to be retrieved by client applications. A group can be bootstrapped from PMIx, MPI, or simply a list of initial addresses. Should the group change, for example when adding or removing a node, the view will be updated in all the service's processes.

From a client application's perspective, several strategies can be put in place to react to a change in the service's group. The simplest one consists of an explicit function that the application needs to call to query the current view of the group. Another possible strategy is the one used in Colza [23], which implements elastic in situ visualization using Mochi. Colza providers declare a dependency on SSG to keep track of the group's view and maintain a hash of this view. Any RPC sent by client applications has this hash as an argument. A mismatch between the hash sent by the client and the hash maintained by a Colza provider informs the latter that the client's view of the group is outdated.

SSG only provides eventual consistency of the group's view to all group members. This can be problematic when components or applications need such a view to be consistent. To solve this problem, Colza uses a two-phase commit approach, with the application itself acting as a controller. In the future, however, we plan to build a consistent view by using the RAFT protocol [20] to coordinate configuration changes across a set of Bedrock-managed processes.

7. Resilience

Designing a resilient distributed system is a difficult task. In the following, we show how thinking in terms of composable building blocks can make this more tractable. We first present two perspectives on resilient designs, before diving into Mochi components and design methodologies that can help implement each of them.

Observation 8: *Our answers to the requirements of online reconfiguration and elasticity were often contained in a component that other parts of a service could rely on. Resilience, on the other hand, is harder to tackle in a composable manner. A fault, by nature, will cut across components, break dependencies, and render data unavailable. Hence, reaction to a fault is also cross-component by nature. Yet design principles for resilient HPC data services must strive to avoid coupling components more than necessary.*

Design principles: top-down vs. bottom-up design. Imagine a distributed service made up of a collection of databases spread across multiple nodes. Realizing that any node can crash, a "top-down" approach to resilience would consist of implementing a component that has knowledge of all the databases in the service and that can react to one of them becoming unavailable. Similar to how Pufferscale dealt with rebalancing by having a global view of the components to rebalance, such a resilience component could, for instance, request that data be replicated across multiple databases, control how this replication is done, and trigger further copies when nodes die. The resilience component itself could be replicated to avoid becoming single points of failure or points of contention.

Another approach, which we term "bottom up," consists of addressing resilience at the lowest level possible first. In the above example, the question would become *How do we make each individual database resilient?* This bottom-up approach is more composable: other components do not need to know how they each manage their own resilience, and implementing resilience in one component benefits all the services that rely on this component. The bottom-up approach is usually insufficient, however, and often some global knowledge of the full service is still required, for example to rebalance data or rewire component dependencies.

In designing composable distributed services, we advocate for considering a bottom-up approach first, leaving the top-down approach for what the bottom-up approach could not solve. In the following, we present multiple approaches and components that can help with both bottom-up and top-down approaches.

Observation 9: *Periodically saving a provider's state (its data) to persistent storage is a first way to enable some form of resilience. When crashing, the component at worst will lose the modifications done since its last checkpoint. Depending on the use case, such a loss could be acceptable.*

Design principle: leveraging parallel file systems. Just as data services complement parallel file systems, parallel file systems can support specialized Mochi-based services by storing checkpoints in a way that makes them accessible from any node. Should a node die, another node can be provisioned and restarted with the same components restoring their respective checkpoint. This solution is fully "bottom-up": each component individually implements how its checkpointing is done. We enable control over the checkpointing of components via their Bedrock module, which provides `checkpoint` and `restore` function pointers. Hence checkpointing can be triggered remotely via Bedrock.

Observation 10: *Data replication is another way of protecting data. In order to be composable, replication and recovery from replicas should involve the least amount of coupling between components.*

Design principle: virtual resources. The concept of virtual resource helps implement bottom-up resilience by having a provider manage a resource that forwards its requests to other components that hold the actual data. For instance, a Yokan "virtual database" could forward the data it receives to N other actual databases living on other nodes. The client accessing this virtual database does not know that the provider it contacts does not actually hold data itself or that the data is replicated. Hence this approach provides a way to implement data replication in a bottom-up manner.

Observation 11: *HPC data services must maintain distributed, shared data structures as a core aspect their functionality. These distributed data structures in turn require mechanisms for maintaining consensus and performing coherent updates.*

Protocols such as PAXOS [25] and RAFT [20] ensure that multiple components agree on commands to apply to their internal state, for example on data to store in a database.

Design principle: composable consensus. To enable consensus across multiple Mochi components, we developed Mochi-RAFT, a RAFT implementation based on C-RAFT [26] and Margo. Mochi-RAFT can help with the bottom-up approach to resilience by implementing state machine replication across components of the same type. For instance, multiple Yokan providers could use a Mochi-RAFT instance as a dependency to ensure that the content of their key-value databases is consistent. It can also help with the top-down approach, for instance by ensuring that a set of RAFT-replicated "controller" providers apply the same commands to an underlying collection of other, nonresilient Mochi components.

Observation 12: *Reacting to faults requires detecting them first. Such a detection should be done by one component with the ability to notify other components of faults.*

Design principle: extending group membership. We already mentioned Mochi's group membership component SSG as a means for client applications to track changes to an elastic service. SSG also includes a fault detection mechanism based on the SWIM gossip protocol [27], [28]. SSG helps with the top-down approach to resilience by providing a way for any member to be notified if any other member dies.

8. Related Work

8.1. Dynamic distributed systems

In HPC. Task-based workflow management systems (WMS) provide the most natural paradigm for elasticity in HPC. Dask [29] and its extended in situ version DEISA [30], [31] are such WMS. They support adaptive deployments by manually adding/removing workers to/from the cluster or by relying on the `adapt(minimum, maximum)` method offered by some Dask deployment solutions such as Slurm, Yarn, and Kubernetes. Dask keeps information about the tasks, their historical runtime, available memory, the status of the workers (idle or saturated), and the reasons for that (e.g., the presence of specialized hardware). The scaling heuristics in adaptive deployments use these to determine a target number of workers to use by scaling up or down.

Ray [32] is a WMS for scaling AI and Python applications, providing static and dynamic resource management. Similarly to Dask, users can add/remove workers manually or use the *autoscaler* feature to react to the application's logical resource demand (requested in `@ray.remote`) but not to accurate metrics or resource utilization.

Parsl [33] implements a cloudlike elasticity module. It provides an elasticity component connected to an extensible *strategy* interface, in which users can implement their elasticity logic. This strategy module periodically tracks the running tasks and available compute capacity, triggers scaling events to match workflow needs, and communicates with the connected providers to scale up and down.

PyCOMPSs [34] is another task-based workflow system. Although PyCOMPSs applications are dynamically scheduled in available resources, thanks to the task-based aspect of this WMS, dynamic resource configurations are restricted only to the cloud deployment of PyCOMPS applications through Docker, Mesos, and others.

While resource reconfiguration and elasticity are easily achievable on the cloud, they are still complicated in HPC because of the static nature of the jobs that contain these workflows and because of queue delays when attempting to provision the workflows. New HPC resource managers such as Flux [6] attempt to solve these problems.

In the cloud. The properties of reconfigurability, elasticity, resilience, and performance introspection described in this paper are at the core of cloud services, where monetary incentives push for high availability, resilience, and resource efficiency. The large ecosystem of microservices available in this space also forces the use of composable designs, using service-oriented architectures, or event-driven architectures. In this space, Kubernetes [35] (K8s) is the work most relevant to us. It runs applications in containers, grouped into pods, and handles the deployment, reconfiguration, and scaling of these pods to physical resources. K8s's YAML-based description of a service is not unlike Bedrock's JSON configuration, although the latter describes components sharing resources within a process, at a much finer granularity, while components in K8s-based services are containerized processes that need to talk to each other via a network. K8s provides an overlay network, which simplifies migration and scaling. Our use of specialized HPC networks and RDMA makes such an overlay network nearly impossible to implement, forcing us to implement mechanisms to track process locations, such as SSG.

8.2. Elastic and resilient storage services

Resilience in parallel file systems is typically handled by using a RAID strategy or erasure coding, which corresponds to a bottom-up approach. Parallel file systems are generally not designed to be elastic or reconfigurable, in large part because they are deployed on a fixed hardware and shared by all the users of a supercomputer. Weka is a notable exception in that it is based on a proprietary data distribution mechanism [36] that includes both rebalancing and autoscaling capabilities. This level of dynamicity is motivated by Weka's targetting both HPC and cloud applications.

While elastic storage is common in the cloud, finding open source implementations of such services is difficult. Elastic storage is generally provided as a service by vendors such as Amazon AWS or Microsoft Azure. The move to rely on cloud resources to run HPC applications may change this in the future and may also highlight the benefit of elastic data services in HPC workflows, motivating their use in traditional supercomputers. We hope that our methodology and set of tools will be a driver in this direction.

Some elements of elastic data service techniques can be found in storage virtualization, a long-standing technique for decoupling physical media from its provisioning and presentation to facilitate storage pool expansion and service migration. This technique, in conjunction with fabric-attached storage, can serve as a foundation for dynamic load-balancing algorithms in data center environments [37].

9. Conclusion

While HPC data services increase productivity by providing interfaces and functionalities tailored to their target applications, dynamic data services for have the potential to provide better performance by adapting to changing access patterns and by being resilient. In this work we have extended our Mochi methodology to enable the rapid development of *dynamic* data services.

Our immediate next step will be to provide quantifiable evidence of these performance improvements by enabling dynamic behaviors in existing Mochi-based services and with real-life applications.

We then plan to investigate authentication, authorization, and encryption methods necessary for Mochi services to become secure in a multiuser environment. Similar to dynamicity, security needs to be enabled in a composable manner, that is, by providing security components to form secure building blocks and by enabling encryption and authentication transparently in existing components.

Acknowledgment

This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357.

References

[1] D. S. Ayres *et al.*, "The NOvA Technical Design Report," October 2007.

[2] S. Ali, S. Calvez, P. Carns, M. Dorier, P. Ding, J. Kowalkowski, R. Latham, A. Norman, M. Paterno, R. Ross *et al.*, "HEPnOS: a specialized data service for high energy physics analysis," in *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2023.

[3] M. Dorier, R. Egele, P. Balaprakash, J. Koo, S. Madireddy, S. Ramesh, A. D. Malony, and R. Ross, "HPC storage service autotuning using variational- autoencoder -guided asynchronous Bayesian optimization," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep 2022.

[4] D. Benjamin, P. Gartung, K. Herner, S. Snyder, R. Wang, and Z. Dong, "Darshan for HPE applications," in *The 26th International Conference on Computing in High Energy & Nuclear Physics (CHEP2023)*, 2023.

[5] J. Lüttgau, S. Snyder, P. Carns, J. M. Wozniak, J. Kunkel, and T. Ludwig, "Toward understanding I/O behavior in HPC workflows," in *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*, 2018.

[6] D. H. Ahn, J. Garlick, M. Grondona, D. Lipari, B. Springmeyer, and M. Schulz, "Flux: A next-generation resource management framework for large HPC centers," in *43rd International Conference on Parallel Processing Workshops*, 2014.

[7] M. Dorier, P. Carns, K. Harms, R. Latham, R. Ross, S. Snyder, J. Wozniak, S. K. Gutiérrez, B. Robey, B. Settlemyer, G. Shipman, J. Soumagne, J. Kowalkowski, M. Paterno, and S. Sehrish, "Methodology for the rapid development of scalable hpc data services," in *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*, 2018.

[8] R. B. Ross, G. Amvrosiadis, P. Carns, C. D. Cranor, M. Dorier, K. Harms, G. Ganger, G. Gibson, S. K. Gutierrez, R. Latham *et al.*, "Mochi: Composing data services for high-performance computing environments," *Journal of Computer Science and Technology*, 2020.

[9] M.-A. Vef, N. Moti, T. Süß, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, "Gekkofs – a temporary distributed file system for HPC applications," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018.

[10] M. J. Brim, A. T. Moody, S.-H. Lim, R. Miller, S. Boehm, C. Stanavige, K. M. Mohror, and S. Oral, "UnifyFS: A user-level shared file system for unified access to distributed local storage," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023.

[11] O. Tatebe, K. Obata, K. Hiraga, and H. Ohtsuji, "CHFS: Parallel consistent hashing file system for node-local persistent memory," in *International Conference on High Performance Computing in Asia-Pacific Region*, 2022.

[12] Z. Liang, J. Lombardi, M. Chaarawi, and M. Hennecke, "DAOS: A scale-out high performance storage stack for storage class memory," in *Supercomputing Frontiers*, D. K. Panda, Ed. Cham: Springer International Publishing, 2020.

[13] H. Tang, S. Byna, F. Tessier, T. Wang, B. Dong, J. Mu, Q. Koziol, J. Soumagne, V. Vishwanath, J. Liu *et al.*, "Toward scalable and asynchronous object-centric data management for HPC," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2018.

[14] C. Docan, M. Parashar, and S. Klasky, "DataSpaces: An interaction and coordination framework for coupled simulation workflows," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10, 2010.

[15] S. Ramesh, H. Childs, and A. Malony, "Serviz: A shared in situ visualization service," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2022.

[16] C. Kelly, S. Ha, K. Huck, H. Van Dam, L. Pouchard, G. Matyasfalvi, L. Tang, N. D'Imperio, W. Xu, S. Yoo *et al.*, "Chimbuko: A workflow-level scalable performance trace analysis tool," in *ISAV'20 In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, 2020.

[17] S. Ramesh, A. D. Malony, P. Carns, R. B. Ross, M. Dorier, J. Soumagne, and S. Snyder, "SYMBIOSYS: A methodology for performance analysis of composable HPC data services," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021.

[18] S. Ramesh, R. Ross, M. Dorier, A. Malony, P. Carns, and K. Huck, "SYMBIOMON: A high-performance, composable monitoring service," in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2021.

[19] "Margo and Thallium microservice templates," https://github.com/mochi-hpc/margo-microservice-template and https://github.com/mochi-hpc/thallium-microservice-template, viewed on January 17, 2024.

[20] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX annual technical conference (USENIX ATC 14)*, 2014.

[21] J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross, "Mercury: Enabling remote procedure call for high-performance computing," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, 2013.

[22] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault, S. Iwasaki, P. Jindal, L. V. Kalé, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. Beckman, "Argobots: A lightweight low-level threading and tasking framework," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[23] M. Dorier, Z. Wang, S. Ramesh, U. Ayachit, S. Snyder, R. Ross, and M. Parashar, "Towards elastic in situ analysis for high-performance computing simulations," *Journal of Parallel and Distributed Computing*, 2023.

[24] N. Cheriere, M. Dorier, G. Antoniu, S. M. Wild, S. Leyffer, and R. Ross, "Pufferscale: Rescaling HPC data services for high energy physics applications," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020.

[25] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column)*, 2001.

[26] "C-RAFT," https://raft.readthedocs.io/, viewed on January 17, 2024.

[27] A. Das, I. Gupta, and A. Motivala, "Swim: Scalable weakly-consistent infection-style process group membership protocol," in *Proceedings International Conference on Dependable Systems and Networks*, 2002.

[28] S. Snyder, P. Carns, J. Jenkins, K. Harms, R. Ross, M. Mubarak, and C. Carothers, "A case for epidemic fault detection and group membership in HPC storage systems," in *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 5th International Workshop, PMBS 2014, New Orleans, LA, USA, November 16, 2014. Revised Selected Papers 5*. Springer, 2015.

[29] M. Rocklin *et al.*, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th python in science conference*, 2015.

[30] A. Gueroudji, J. Bigot, and B. Raffin, "DEISA: dask-enabled in situ analytics," in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2021.

[31] A. Gueroudji, J. Bigot, B. Raffin, and R. Ross, "Dask-extended external tasks for HPC/ML in transit workflows," in *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023.

[32] R. Nishihara, P. Moritz, S. Wang, A. Tumanov, W. Paul, J. Schleier-Smith, R. Liaw, M. Niknami, M. I. Jordan, and I. Stoica, "Real-time machine learning: The missing pieces," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*, 2017.

[33] "Parsl: Pervasive parallel programming in."

[34] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, and J. Labarta, "PyCOMPSs: Parallel computational workflows in Python," *The International Journal of High Performance Computing Applications*, 2017.

[35] "Kubernetes," https://kubernetes.io/, viewed on January 17, 2024.

[36] "WekaIO Distributed Data Protection," https://www.weka.io/resources/white-paper/distributed-data-protection/, viewed on January 17, 2024.

[37] A. Singh, M. Korupolu, and D. Mohapatra, "Server-storage virtualization: Integration and load balancing in data centers," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.