Rapid Development of Scalable, Distributed Computation with Abaco

Joe Stubbs, Matthew W. Vaughn, and Julia Looney
Texas Advanced Computing Center
University of Texas, Austin
Austin, TX 78758-4497
Email: jstubbs@tacc.utexas.edu

***Abstract*—Computational science experiments are growing increasingly distributed, both with respect to the computing resources they leverage and the teams of individuals carrying them out. Abaco [1] is a RESTful API and distributed computing platform built to enable rapid development and massive scalability of computational components across teams of developers working with heterogeneous physical compute resources. Abaco is an NSF funded project under active development that had its first major production release at the beginning of 2018. Leveraging Linux container technology and the Actor model of concurrent computation [2], Abaco provides functions-as-a-service to the general scientific computing community with three primary capabilities: *reactors* to enable event driven programming models; *asynchronous executors* to provide a map primitive for scaling pleasantly parallel workloads to the cloud from within running applications; and *data adapters* for the creation of rationalized microservices from disparate and heterogeneous sources of data. In this paper we describe the main features of Abaco, discuss some success stories from early adoption and describe directions of future work.**

I. Introduction

Through a rise of data-driven methods, modern computational experiments require a network of distributed components coordinating across heterogeneous physical infrastructure. A common approach involves refining raw data, often times collected in experimental facilities, wet labs, or via sensors deployed in the field, through a series of decision-based pre-processing pipelines to produce more pristine data artifacts. These artifacts are then fed into analytic models or other codes, producing results which are validated against observed phenomena, and the process is repeated. Building reproducible experiments in such a context involves distributed teams with various areas of expertise working with different technologies. The Abaco system [1] is a distributed computing platform designed to accommodate the needs of such computational experiments. Specifically, Abaco attempts to enable the rapid development of portable, loosely coupled components developed by distributed groups that can be integrated together and independently scaled with minimal effort.

Abaco (Actor Based Containers) is an NSF-funded project combining Linux container technology with the Actor model of concurrent computation [2] to provide functions-as-a-service to the national research community. In the Actor model, computational primitives called actors are assigned a unique mailbox capable of receiving messages. In response to a message, actors can perform computation, create other actors, and send additional messages through the system. While the Actor model dates back to the 1970s, Abaco's innovation is to associate each actor with a Linux container image and expose the actor's mailbox as a URI over HTTP. When a message is sent to an actor's mailbox, Abaco launches a container from the actor's image and injects the message contents into the container environment. Actors can be registered as either stateful, meaning they can only process one message at a time, or stateless, meaning they are safe to process multiple messages in parallel. Using internal queues, Abaco is capable of managing tens of thousands of messages at a time for hundreds of actors.

Abaco operations are exposed to the end user as RESTful APIs that speak JSON including: actor registration and management, actor messaging, actor state, execution history and logs, as well as an administrative API for scaling Abaco itself. To register an actor, a user makes a POST request to the actors endpoint, passing a JSON-formatted list of attributes describing the actor. In particular, the description includes the Docker image to associate with the actor. Abaco responds in JSON with metadata it generated about the actor including a UUID and mailbox URL. Once an actor is registered, Abaco will queue a container execution for each POST request to its mailbox URL. Abaco guarantees that each message will result in exactly one container execution and that, for stateful actors, only one container will execute at a time for a given actor.

Because each actor is defined with its own container image, virtually any programming language or technology can be used for its implementation. Moreover, as a rooted file system, a container image necessarily contains all required dependencies, providing a high degree of portability between local development and production execution across whatever compute resources a given Abaco instance might be leveraging. Any server capable of executing the container runtime can be leveraged as a resource for launching actor containers, thus enabling Abaco to utilize a wide range of heterogeneous hardware types. Sending messages over HTTP provide a straight-forward mechanism for integrating loosely coupled components developed by different teams in potentially different technologies.

After becoming an officially funded NSF project in September of 2017, work began to harden the Abaco prototype and

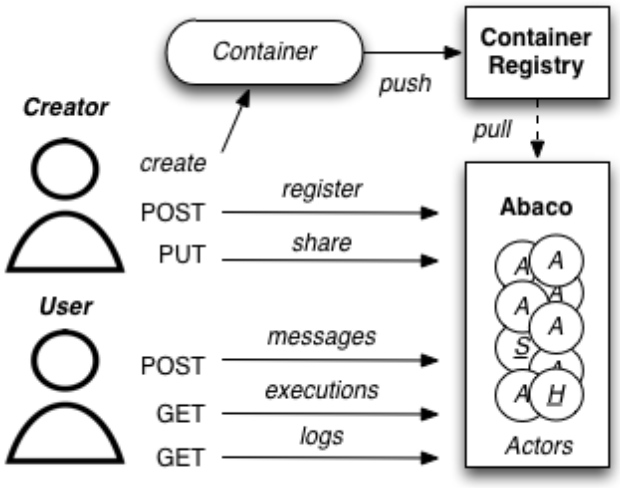

Fig. 1. Using Abaco. Abaco usage workflow: Creators build functions deployed in containers, send them to a public registry, then createand share actors via web service calls. Users execute functions by making web service calls that send a message to a specific actor.

add features necessary for an initial production launch at the Texas Advanced Computing Center in January of 2018. Since then, multiple adoptions have occurred and two additional point releases have been deployed, but a lot more work is planned over the remaining two and a half years of funding. In the remaining sections of this paper, we discuss the primary capabilities being developed and highlight some successes from initial adoption, describe the Abaco architecture in more detail, compare Abaco to other work and discuss areas of future direction.

II. Software Comparison

Abaco draws inspiration from a number of existing software systems.

1) Functions-as-a-service: Commercial offerings such as AWS Lambda [3], Google Cloud Functions [4] and Joyent Manta [5] provide functions-as-a-service similar to Abaco, but control over the runtime environment is far more limited than an arbitrary container image as are the resources available to the function. For instance, a Lambda function has a maximum runtime of 300s over a single core, 3008 MB of RAM and 512 MB of temporary storage, with a deployment package size that must be under 50 MB [6].

Very recently, the open source project OpenFAAS [7] was launched to provide an open source functions-as-a-service based on Docker images. While OpenFAAS is perhaps closest in spirit to Abaco, at present it requires the function container images to run an HTTP server. Function calls are proxied directly back to the web service, so synchronization must be managed by the application. OpenFASS requires its compute servers to be members of a Kubernets or Swarm cluster, the setup and management of which is nontrivial and may have security implications in a multitenant environment. The only requirement of an Abaco compute server is that it be capable of running the Docker daemon. OpenFAAS also lacks several features Abaco provides as part of the Actor model.

2) Containers-as-a-service: Amazon's Elastic Container Service (ECS) [8] and Google's Container Engine [9] are commercial examples of containers-as-a-service offerings. While enabling the use of arbitrary container images, these APIs lack the various Actor-based semantics that distinguish Abaco's design. As a result, these services are better suited for long-running server daemons.

3) Distributed Computing Platforms: Systems like Apache Spark [10], Apache Storm [11], iPython parallels [12] and AWS Kinesis [13] provide various features that resemble Abaco's asynchronous executors capability and even support additional programming paradigms such as IPC with much better performance. Abaco asynchronous executors only attempt to address pleasantly parallel compute jobs with the goal of making them substantially more accessible.

III. Primary Capabilities and Initial Adoption

We describe three primary high-level capabilities of the Abaco platform and some initial adoption to date. These capabilities are in various stages of maturity; we describe additional development efforts needed to fully realize these capabilities in more detail in section six. We propose that the breadth and diversity of these capabilities as well as the initial adoptions is compelling validation of the underlying design.

A. Reactors for event-driven computing

Event-driven architectures utilize software components that execute in response to certain events occurring in the system. The Abaco platform provides specific functionality and tooling for developing what we refer to as *reactors*: actors meant to execute in response to events. Many popular platforms provide event notifications in the form of webhooks: POST requests to a specified URI whenever the event occurs. Examples include code repository hosting platforms such as Github, Bitbucket and Gitlab, continuous integration servers such as Jenkins and Bamboo, cloud storage APIs such as Box and DropBox, and other middleware solutions like the Agave science-as-a-service API platform. By subscribing an actor's mailbox URI as the target for a webhook, the actor becomes an event processor for the subscribed event. Abaco provides features such as its nonce service for generating limited-use, pre-authenticated tokens for a specific actor endpoint to simplify authentication from external systems.

1) Adoption: ETL in the Synergistic Discovery and Design Environment: The Synergistic Discovery and Design Environment (SD2E) [14] is a gateway and computing platform leveraged by a community of investigators spanned across multiple research universities, government and industry labs working on grand challenge problems in synthetic biology and related areas. The labs within the SD2E community generate raw experimental data at a rate of roughly 3 TB per day, and developers have created a series of reactors to implement the complex ETL pipelines that must be performed every

time new files are generated. SD2E ETL reactors implement validation and transformation steps as well as launch longer, asynchronous transfers and compute-intensive batch jobs on HPC resources. SD2E reactors are integrated into multiple external systems including Gitlab, Jenkins and the Agave science-as-a-service API.

2) Adoption: webterm management in IPT on the Web: The Interactive Parallelization Tool (IPT) [15] is an NSF-funded project providing a command-line tool that analyzes a serial source code and generates a parallelized version of the code by asking the developer a series of questions interactively. IPT on the Web is a gateway that provides users with a terminal emulator in the browser (webterm) for running the IPT command line tool as well as web forms for managing files, compiling codes and launching benchmarking runs on HPC systems such as Stampede and Comet. IPT on the Web is comprised of a stateless web application whose architecture is simplified through the use of an Abaco reactor for managing the user webterms. Each webterm runs as a single Docker container on an elastic cluster in the NSF Jetstream cloud. The reactor manages these webterm containers by responding to login and logout events. An upcoming paper will provide more details on the IPT architecture.

3) Adoption: Automatic creation of Singularity Biocontainers: The Biocontainers initiative [16] is an open-source effort aimed at providing portable execution environments for bioinformatics. One of the primary efforts involves creating Docker images from Conda package recipes. Since many of the world's most powerful supercomputer still only offer the Singularity container runtime, The Life Sciences Computing group at the Texas Advanced Computing Center created an Abaco reactor to automatically build a Singularity image from a new or updated Biocontainer Docker image and upload it to the TACC Singularity image repository. The bulk of the reactor development was actually done by an undergraduate research assistant. To date, several thousand Singularity images have been built through this pipeline.

B. Asynchronous Executors for pleasantly parallel workloads

A more nascent capability, whose initial features were only released to production in early March, provides high-level tooling for asynchronously executing functions on the Abaco cluster from directly within a running application; for example, when scaling out pleasantly parallel workloads such as parameter sweeps. To support such a capability, the Abaco Python SDK provides an executor class which wraps calls to the service for creating actors, sending messages, and retrieving results. The executor class implements Python's *concurrent.futures.Executor* interface [17] to provide compatibility with threadpool and processpool executors that may already be in use for parallelizing code on a single machine. The Abaco executor provides *submit()*, *blocking_call()* and *map()* methods for executing a callable against (respectively, mapping a callable over) a set of input data. The technique employed serializes the callable and data and sends them together as a single binary message to an actor registered as part of instantiating the executor.

The actor associated with the executor is assigned a special image developed by the Abaco team that knows how to deserialize a properly formatted message back into the original callable and data, execute the callable against the data, and register the result to Abaco's results endpoint. Somewhat surprisingly, this simple approach was able to achieve 1 Teraflop of average performance on an 8 node Abaco compute cluster by running matrix-matrix multiplication of large, square numpy arrays. A key point is that the actor's container image needs all software dependencies preinstalled in order to minimize the code needed to be serialized at execution time. The Abaco development team plans to integrate this capability into TACC's custom JupyterHub offering where user notebook servers run out of Docker containers. The Abaco executor will be able to automatically detect the JupyterHub context and register an actor with a matching image, thereby guaranteeing the software dependency requirement without the user ever being aware that container images are in use.

1) Use case: Exploratory runs of Opensees from Jupyter: The Abaco team developed the initial implementation of the asynchronous executors capability in close collaboration with researchers at the University of California, Berkeley who needed a way to launch a number of short, exploratory runs of the Opensees application [18] for earthquake modeling. Resources required for these runs were on the order of a few dozen cores over a span of 10-15 minutes. While such runs might collectively take a few hours on an individual workstation, leveraging a traditional HPC system may not improve time to solution if it is experiencing long queue times.

C. Data adapters for rationalized interfaces to heterogeneous data sources

Still in the design and early prototyping stages, *data adapters* are specialized actors that create high-quality API access to data from disparate external sources such as files, databases, HTML tables and third-party APIs. Through base images and SDKs, Abaco will enable a developer to write a short function to return results from the data source based on a query, and Abaco will automatically format the response to common serialization standards such as JSON. Additional benefits enjoyed by all actors such as authentication and access control, fault tolerance and scaling are inherited by data adapters. Once the initial data adapters feature becomes available, the Abaco team will develop additional features such as global search and caching across adapters and a provenance endpoint compliant with the W3C PROV specification [19].

IV. Additional Features

The primary capabilities are enhanced by a number of additional features, many of which can be tuned through the main Abaco configuration file. We briefly describe a few highlights below.

1) Multitenancy and JWT Authentication: The Abaco platform is multi-tenant, meaning that different projects can leverage the platform and enjoy logically isolated views into the API. Abaco provides a configurable authentication mechanism including support for the standard JSON Web Tokens (JWT) [20]. Tenants can be configured in the Abaco configuration file and resolved at runtime using the JWT.

2) Integration into the greater TACC ecosystem: An Abaco instance can be configured to leverage a number of additional services within the TACC ecosystem, and these integrations will find analogs in other HPC centers. First, Abaco integrates with the TACC identity and access management system so that actor containers automatically start with a valid OAuth token representing the user who registered the actor. This enables actors to make authenticated calls to other TACC APIs, including Abaco itself, as part of their execution. Second, on a per-tenant basis, Abaco can be configured to create mounts from various external file servers directly into the actor containers. At present, NFS mounts to various collections on TACC's Corral storage system as well as native Lustre mounts to TACC's global parallel file system, Stockyard, are available. Thus, actors can leverage files and directories from these storage systems via a POSIX interface. Finally, actor containers are started with the uid and gid associated with the TACC account that registered the actor. This means that the POSIX permissions on the underlying file systems are strictly adhered to.

3) Default actor environments and custom container environments: Environment variables are a standard way of parameterizing container images. When an actor is registered, the operator may choose to provide a dictionary of name-value pairs to serve as the actor's default environment. Then, query parameters sent to the messages API may be used to override or extend the default environment with values for a specific actor execution.

4) Basic Permissions System with RBAC: Abaco provides a basic permission system recognizing the READ, EXECUTE and UPDATE permission levels. Actors can be shared (or unshared) with other users at the different levels by making requests to the actor's permissions endpoint. Additionally, when configured with JWT authentication, Abaco will parse the JWT for specific roles to support different levels of access. Presently, four roles are recognized: *admin*, *privileged*, *user* and *limited*, though it is easy to extend the system to support additional roles.

5) State API: In general, stateful actors are responsible for managing their own state, but to ease that burden, a state endpoint is available. The state endpoint provides a store, private to each actor and accessible during execution time, for storing and retrieving arbitrary JSON-serializable objects across different actor executions.

V. Architecture

Inspired by the Actor model, the Abaco architecture consists of several narrowly focused agents communicating via messages. The web service API layer is divided into several agents

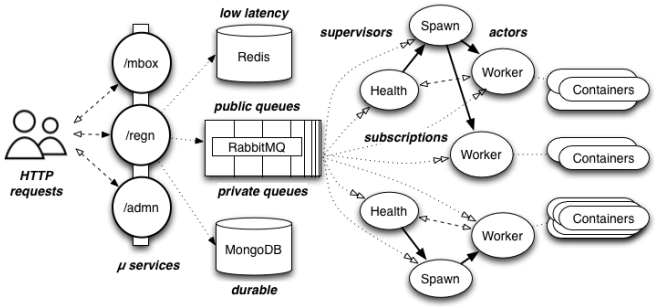

Fig. 2. The Abaco Architecture. Abaco employs an architecture inspired by the Actor model, with independent agents organized into a supervisor hierarchy. Messages are passed between agents through channels brokered by RabbitMQ, while Redis and MongoDB provide persistence. Actors execute functions within containers in response to messages.

to handle different endpoints, including actor registration and management, actor messages and results, actor execution history, details and logs, and an administration API for internal scaling and reporting. The web services layer communicates with various backend agents by sending messages over a flexible channel mechanism. All channels currently leverage AMQP brokered by RabbitMQ, though it is easy to swap RabbitMQ for another message broker such as Kafka or even a peer to peer transport like zeromq. The Abaco development team is considering such options for the data adapters feature where it will be crucial to minimize latency.

Abaco's backend agents include one or more spawner processes responsible for starting and stopping worker agents associated with a specific actor. Workers subscribe to an actor's mailbox queue and start actor containers when new messages arrive. The workers supervise the execution of the actor containers, collecting results, execution data such as total runtime and resources (cpu, memory and i/o) used, and container logs. Health agents periodically check the health of all workers and spawners.

The Abaco agents are stateless, and thus multiple instances of any given agent can be started and stopped at any time for upgrades and to achieve a desired scalability with minimal impact to the overall health of the system. For persistence, two databases are employed. The state of the currently running system, including all registered actor data, is stored in Redis while accounting and historical data such as execution history and logs are stored in MongoDB. Redis was chosen for its low latency and transaction semantics while MongoDB, through clustering and sharding, enables performant access to very large datasets.

VI. Future Work

While already in production supporting several use cases, many additional development efforts are planned for the Abaco system over the next two and a half years. In this section we provide an overview of the features planned.

1) Autoscaling: For actors registered with the stateless property, Abaco is developing a capability to automatically

scale the number of workers associated with an actor according to the size of the actor's mailbox. An initial prototype is currently being developed on top of Prometheus, an open-source time series database and monitoring server. Via a new Abaco metrics endpoint, mailbox sizes as a function of time are scraped by Prometheus. When mailbox size thresholds are crossed, a Prometheus alert triggers a webhook to an administrative scaling endpoint within Abaco, and appropriate action is taken to add or remove workers. The Abaco team is exploring different algorithms for triggering the scaling alert, including computing rates of change of mailbox sizes over different intervals of time. An interesting research problem is to automatically learn efficient scaling behaviors for specific actors across different times of day based on historical data.

2) Data Adapters: The data adapters capability described in section two is still in early development stages, and the Abaco team is working with multiple groups interested in the functionality to define precise requirements for the initial feature release. The goal is to enable a data adapter to expose a source of data easily, perhaps with a function written in only a few lines of code. Registration of data adapters will extend the base actor registration to allow provenance to be captured in accordance with the W3C PROV standard. Once the initial release is made, development will turn to more advanced features such as data adapter response caching and global search. Caching will be optional, and specific aspects of it behavior, such as time to live, will be customizable on an individual adapter basis. A key aspect of the search design will be the determination of global ontology for the query language. An interesting long-term research question is to what extent an effective and efficient ontology can be automatically learned from historical data. Open source projects such as Elasticsearch are being considered for the implementation.

3) Events API and Actor Subscriptions: A formal mechanism enabling external systems to publish events to Abaco will be developed together with a subscription capability so that actor executions can be triggered from external events without requiring the external system to be capable of making webhooks. This event system can also be consumed internally to provide additional features such as actor linking where a given actor automatically executes in response to another actor completing an execution.

4) High Performance Private Container Image Registry: Currently, actor registrations require an image attribute that references a Docker image available on the public Docker registry. This precludes leveraging private images for actors and has the potential to introduce performance variance since images must be fetched from hub.docker.com to any Abaco compute node from which actor workers will reside. In a future release, Abaco will integrate a dedicated private image registry on high-performance storage hardware across its production compute cluster.

5) Secrets API: In addition to providing support for private images via the private image registry, Abaco plans to offer a secrets API for centrally managing sensitive information such as database passwords and API credentials.

6) Additional Tooling and Sample Images Library: Currently, an official command line interface (CLI) and Python SDK are available, but the development team plans to make SDKs available for additional languages. Based on demand, JavaScript and Ruby are likely to be the next SDKs developed. In addition to SDKs, a growing library of sample images is being produced to cover common tasks. Throughout the course of the project, numerous utility images and examples will continue to be developed by both the core Abaco development team as well as the open source community.

7) Interactive Web Environment: The initial release of Abaco provided a simple web interface to some of the basic endpoints as well as an admin tab for administrators to get a quick view into the overall usage and health of the system. In a future release, the interface will be overhauled and extended to provide a comprehensive, interactive environment for working with actors. A possible integration with JupyterLab being designed at present would enable the development, registration, execution and live debugging of actors all from within the browser.

VII. Conclusion

Abaco is a distributed computing platform providing functions-as-a-service though a novel unification of the Actor model and Linux containers. Using Abaco, components of a computational science experiment can be built with different technologies and integrated together through message passing over HTTP. Containers offer a significant degree of portability which simplifies the development life-cycle, and the Actor model enables the platform to provide scaling and fault tolerance. The platform is general enough to enable event-driven programming, scaling out of pleasantly parallel functions from within running applications, and data federation and rationalization from disparate data sources. In its first year of funding, Abaco has been adopted by a broad range of users and applications, including leading scientists working on multi-institutional projects as well as undergraduate students on individual REUs.

Acknowledgment

This material is based upon work supported by the National Science Foundation Division of Advanced CyberInfrastructure (1740288 and 1127210), the National Science Foundation Plant Cyberinfrastructure Program (DBI-0735191), the National Science Foundation Plant GenomeResearch Program (IOS-1237931 and IOS-1237931), the National Science Foundation Division of Biological Infrastructure(DBI-1262414), and the National Institute of Allergy and Infectious Diseases (1R01A1097403).

References

[1] (2014) Abaco. Last access: 2015-11-11. [Online]. Available: https://github.com/TACC/abaco

[2] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *International Joint Conference on Artificial Intelligence*, 1973.

[3] (2018) Aws lambda: Serverless compute. Last access: 2018-03-17. [Online]. Available: https://aws.amazon.com/lambda/

[4] (2018) Google cloud functions: Event-driver serverless compute platform. Last access: 2018-03-17. [Online]. Available: https://cloud.google.com/functions/
[5] (2018) Manta, tritons object storage and converged analytics solutions. Last access: 2018-03-17. [Online]. Available: https://github.com/joyent/manta
[6] (2018) Aws lambda limits. Last access: 2018-03-17. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/limits.html
[7] (2018) Openfaas: Serverless functions made simple for docker and kubernetes. Last access: 2018-03-17. [Online]. Available: https://github.com/openfaas/faas
[8] (2018) Amazon elastic container service. Last access: 2018-03-17. [Online]. Available: https://aws.amazon.com/ecs/
[9] (2018) Google container engine (gke) for docker containers. Last access: 2018-03-17. [Online]. Available: https://cloud.google.com/container-engine/
[10] (2018) Apache spark: Lightning fast cluster computing. Last access: 2018-03-17. [Online]. Available: https://spark.apache.org/
[11] (2018) Apache storm. Last access: 2018-03-17. [Online]. Available: http://storm.apache.org/
[12] (2018) Using ipython parallel for parallel computing. Last access: 2018-03-17. [Online]. Available: https://ipython.org/ipython-doc/3/parallel/
[13] (2018) Amazon kinesis. Last access: 2018-03-17. [Online]. Available: https://aws.amazon.com/kinesis/
[14] (2018) Synergistic discovery and design environment (sd2e). Last access: 2018-03-18. [Online]. Available: https://sd2e.org/
[15] R. Arora, J. Olaya, and M. Gupta, "A tool for interactive parallelization." ACM. Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment (XSEDE), 2014.
[16] (2018) Biocontainers. Last access: 2018-03-17. [Online]. Available: https://biocontainers.pro/
[17] (2018) 17.4. concurrent.futures: Launching parallel tasks, python 3.6.5rc1 documentation. Last access: 2018-03-17. [Online]. Available: https://docs.python.org/3/library/concurrent.futures.html
[18] F. McKenna, "Opensees: A framework for earthquake engineering simulation," *Computing in Science and Engineering*, vol. 13, 2011.
[19] (2018) Prov-overview. Last access: 2018-03-17. [Online]. Available: https://www.w3.org/TR/prov-overview/
[20] (2018) Json web token (jwt). Last access: 2018-03-17. [Online]. Available: https://tools.ietf.org/html/rfc7519