Towards a new paradigm for programming scientific workflows

Reginald Cushing
University of Amsterdam
Amsterdam, The Netherlands
r.s.cushing@uva.nl

Onno Valkering
University of Amsterdam
Amsterdam, The Netherlands
o.a.b.valkering@uva.nl

Adam Belloum
University of Amsterdam
Amsterdam, The Netherlands
a.s.z.belloum@uva.nl

Cees de Laat
University of Amsterdam
Amsterdam, The Netherlands
delaat@uva.nl

***Abstract*—Applications and infrastructures are increasingly becoming more complex. Infrastructures have several layers of virtualisation, programmability and management while scientific applications are diverse in their computing model archetypes. Mapping these two opposing ends of the stack is a non-trivial task. Here we propose a new programming paradigm and architecture that takes into account the different layers of the stack in an effort to create isolated, portable and scalable application micro-infrastructures.**

***Index Terms*—exascale, workflows, data processing, micro-infrastructure**

I. Introduction

Exascale is hard to define, at best it is an agglomerate of notions and terms that together try to capture the next frontier in computing. This scalability step poses a challenge in most computing fields such as data storage, processing, power, memory and networking to name a few. Many countries are funding heavily to build an exascale supercomputer [1] [2] but supercomputers will only solve a small part of the exascale challenge. The remaining challenges can be roughly formulated into questions that are yet to be solved. How can you store, index, share and track the ever increasing amount of distributed data? How can you move an ever increasing amount of data? How can you build scalable applications? How can you port applications on globally distributed heterogeneous stacks? how can minimize application energy consumption? how can globally dispersed research teams collaborate? how can data sovereignty be safeguarded? How can you process data at line speed rate? And finally, how can you orchestrate all the above?

In this article we will propose an approach whereby application program, middleware and infrastructure are flattened into a virtual micro-infrastructure with the aim to scale, isolate and port complex applications better.

II. Challenges

A. Infrastructure

The environment in which scientific applications need to run has changed dramatically over the last decade, from homogeneity at many levels of the stack in the Grid era [4] to extreme heterogeneity with virtualization, containerization and software defined-anything. This infrastructure shift towards a hybrid of physical and virtual hardware creates new challenges for middleware to schedule, run and optimize applications on a myriad of resources. Initial approaches to virtual platforms were to keep the status-quo by extending the job batching systems to the cloud [5]. As more virtualizing technologies and application frameworks evolve, application development becomes a complex jigsaw puzzle of which frameworks and technologies to use which by the time of release would already be outdated.

The main challenge here is how to develop application pipelines that can adapt to a continuous shift in infrastructure?

B. Application

Scientific applications have also evolved drastically, from a more traditional bag of tasks [6] written in one language and needing few dependencies to applications needing complex framework, infrastructure setups and dedicated hardware. The immediate repercussions of this shift meant that traditional shared resources such as HPC sites became to stagnant since only a limited approved frameworks would be installed on site. Furthermore, asking site admins to install every new framework and different versions is surely an unscalable approach.

Workflow management systems for scientific applications have been a main research topic. The aim of many workflow systems are twofold; an intuitive user interface to express the application logic and a coordination system that orchestrates the application on some environment. E.g. Taverna [18] coordinates web services using a pictorial interface that expresses DAG style workloads. Pegasus [19] also models computation s a DAG and separates workflow description and workflow execution thus being able to map to different resources. Other workflow systems model computation in different ways e.g. Pumpkin [17] models data processing as data transformation network while others nature-inspired models such as chemistry-based modeling [16]. The traditional workflow systems tended not to model the underlying infrastructure which we consider a limitation in current environments with many layers of virtualization.

The main challenge here is how to facilitate rapid application development and deployment on the different resources?

This research is partly funded by the PROCESS project. https://www.process-project.eu/

978-1-7281-2451-3/19/$31.00 ©2019 IEEE
DOI 10.1109/eScience.2019.00083

C. Middleware

Maybe, one of most apparent evolution in the scientific application area is the disintegration of the traditional middlewares such as gLite [7]. This disintegration is understandable since the traditional middlewares assumed static infrastructures and few application computing models. The traditional middleware architecture model was that of the hour glass [4] whereby applications and resources sat at the opposite sides of the hour glass and the middleware compromised the narrow neck of the hour glass. This meant that a few standards and services would map the many applications to the many resources. With rapid development in both the latter areas, traditional middlewares could not keep up or could not fit in the new environment of computing. Added to this, the line between middleware and application frameworks is, increasingly, being blurred since frameworks, both at application level e.g. TensorFlow [8], Spark [9] , etc and infrastructure level such as Kubernetes [10] include middleware routines such as scheduling. The trend is apparent, that of a shift towards application specific environments. This poses a new set of challenges most notably is the scaling of applications beyond locally tuned clusters for specific applications.

D. Data

The nature of many current scientifically application are dependent on large volumes of dispersed data. From sensor data as in the LOFAR [3] to training sets for ML-based applications. E.g LOFAR project has produced 30PB of data with a steady linear increase over time. Much of this data is yet unprocessed due to lack of infrastructure to move and process this data with enough speed. Such scenarios expose a narrow bottleneck in our current distributed infrastructure i.e. of moving large volumes of data over wide area networks. The lack of hardware capacity is only one limitation, programmability is another. Due to the complexities of application pipelines, staging and caching data becomes an application specific task e.g. protocols used, tape drive staging, retention policies, etc all need to be factored into the application pipeline.

The challenge here is how to smartly move data to make most of the underlying infrastructure e.g. scheduling data transfers in conjunction with application pipelines, finding caching nodes that optimizes transfers.

E. Security and Privacy

The proliferation of data centric applications also means that data privacy and security are a main concern amongst data owners. E.g. medical workflows work in sensitive patient data which, for privacy reasons, may not be able to leave the premises. Such scenarios pose challenges when trying to scale up such applications. HPC sites or cloud resources are not often available on premise thus limiting what can be done with the data.

The challenge here is how to create secure distributed infrastructures that would allow to scale sensitive data processing on dispersed resources.

III. Architecture

In this section we describe our approach to tackle the above challenges through the dynamicity of virtualization and programmability.

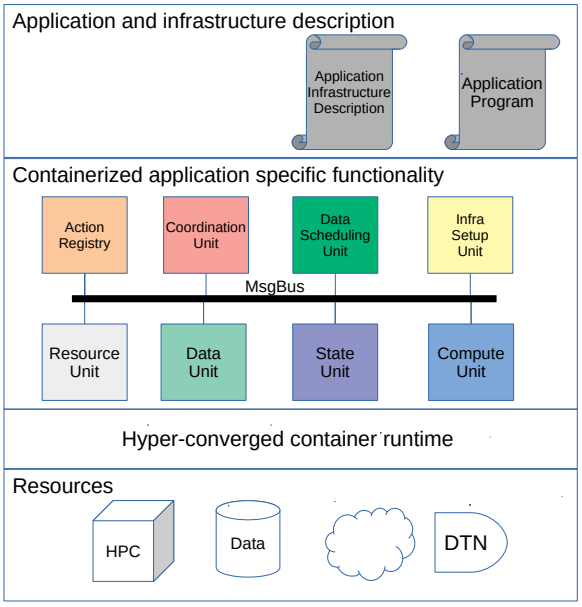

Fig. 1. Architecture components for a container centric middleware. At the top level are the workflow program and an infrastructure description where access points, credentials and settings are defined. The middle part is a virtual infrastructure which combines the application and infrastructure into one logical layer. A hyper-converged container runtime can virtualize compute, network and data thus providing maximum programmability to setup and host application containers. The lower level are the actual distributed resources for compute and data.

A. Application program

The application program is a set of instructions that are meant to coordinate the application workflow. Our proposed programming model is based on dataflow [13], reactive [14], and event [15] processing models which together allow for a model that combines and coordinates loosely coupled components together. The main constructs of our model are *events, guards,* and *actions*. Events and guards trigger actions which intern generate more events which trigger more actions.

An event object is 5 tuple object consisting of an id, type, value array, change function, and state.

$$event := \{id, type, [values], change(), state\}$$

Id is a unique identifier of the event which is used to track the event in the runtime and allow to bind the event with actions. Type defines the type of values stored for the event e.g. boolean or integer. Values are an array of current and past events. State object records state information about the event such as the progress of the last event. Events can take time e.g. an event the includes downloading data would start when an action fires the event that starts downloading and resolved when the data is downloaded. For this reason, values

are set through *futures* and *promises* asynchronous paradigms. A change function is triggered whenever a value has been added to the value array. The purpose of this function is to fire the event if 'enough' change has occurred. E.g. a change function on temperature values might take the rate of change over a time window to decide if the event should fire or not. A simple event is a boolean event e.g. *Raining* which has a type boolean and value true or false and state showing heuristic of past values. Another event can be *Temperature* which is of type float and state that records the past temperatures.

Guards are functions applied to a list of event state object that return true or false. The guard function combines multiple events with combinatorial logic to decide if the action should be fired or not. Every time an event is fired, the dependant guard functions are evaluated on all input events.

Actions are the compute unit functions III-C that do the actual work. Actions return events as output. The program flow is event based where reactions to state change of an event value will fire actions. Thus, concurrent events can trigger parallel actions. Program statements are of the form:

$$[events] : guard \rightarrow action \rightarrow [events]$$

The simple statement construct is powerful enough to express several programming models e.g. Algorithm 1 shows a typical dataflow programming model where functions are fired on the availability of data instead of instructions being evaluated sequentially as is done in imperative model. Algorithm 2 shows communicating actor functions using channel events. The event is a channel and the condition is a to check for a new message. Algorithm 3 shows a repetition construct where the counter is an event of integer type and a function increments the event value. The two statements show an infinite loop since the *equals* condition on the counter event will reset the counter thus restarting from 0.

Algorithm 1 Dataflow

$data1 : available() \rightarrow process(data1) \rightarrow data2$
$data2 : available() \rightarrow process(data2) \rightarrow data3$
$data3 : available() \rightarrow print(data3) \rightarrow null$

Algorithm 2 Actor

$channel1 : newMsg() \rightarrow actor1(channel1) \rightarrow channel2$
$channel2 : newMsg() \rightarrow actor2(channel2) \rightarrow channel1$

Algorithm 3 Iteration

$counter : lessThan(100) \rightarrow inc(counter) \rightarrow counter$
$counter : equals(100) \rightarrow reset(counter) \rightarrow counter$

B. Infrastructure description

The infrastructure description is a set of container contextualization that infrastructure unit will use to instantiate containers. Some of the important contextualization is the credentials for resource and data adapters. The adapters are the bridge to the 'physical' world and need to be contextualized enough to have access the the resources. For this reason care must be taken to protect these descriptions through encryption.

C. Compute Units

The minimum compute unit in our architecture is a containerized function. Containerized functions are any compute routine that is encapsulated in a container such as Singulartiy [11] or Docker [12] and invoked through a message bus system. A message bus mechanism allows functions to be invoked in a pull fashion. This is important since running containers will, often, have no public facing interface (e.g. containers running on HPC nodes). The ability to encapsulate functions with all their intricate dependencies allows us to port the function to many different, heterogeneous resources and dynamically scale the function. The properties of these functions are:

- Stateless: Synchronizing state between distributed functions will negatively impact scalability. For this reason container functions should be made stateless. A separate state unit will be responsible for communicating state between functions.
- Referentially transparent: The output of the function does not depend on the ordering of other functions. This makes the function independent and thus able to scale better.

D. Routine registry

Two or more compute functions can be combined to form a new action. This is done by writing coordinating scripts that are hosted in the action registry. In these scripts, developers define actions by composing different compute containers. The published actions are what is used in the application program.

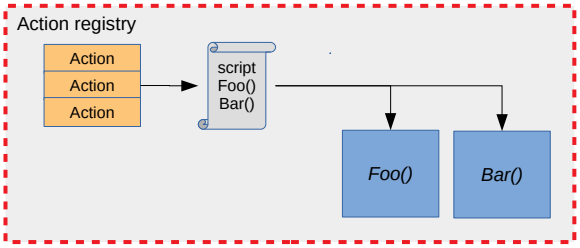

Fig. 2. Actions registry: Composing multiple containers into one actions using activity scripts.

Actions are imported as libraries. The compilation of the application program does the necessary action definition checking thus abstracting away from the fact that it is a containerized routine. These scripts, dubbed *activities*, are written in an existing language and can make use of variables and the typical conditional and iteration constructs.

E. Data Units

In our model data is geographically dispersed, replicated, under different jurisdictions, accessed with different protocols and have different access policies. For these reasons, any

middleware needs to cater for all these variations. Having a monolith system that encompass all combinations and variations is not a scalable solution. Our approach is to reuse the containerized functions to implement the different staging methods. For example, custom protocol to move data such as UDT can be containerized in a send/receive function while a data catalogue container acts as a user's data registry.

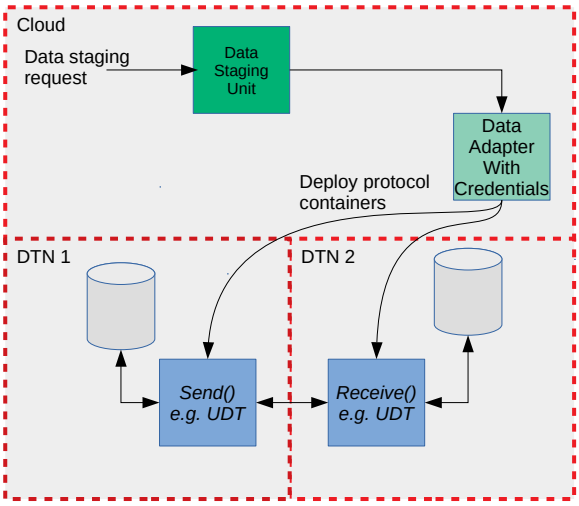

Fig. 3. Data staging components.

A recent effort in making data transfers more efficient is the notion of Data Transfer Nodes (DTN). These nodes are nodes with dedicated hardware such high throughput network cards, fast disks and good network links. The network between geographically distributed DTNs is often optimized which means collaboration between network engineers on both sides of the network. The outcome is to push data as fast as possible over wide area networks. Here, containers are also applicable since DTNs can be user programmed with custom protocol containers such as the UDT example mentioned earlier. Figure 3 illustrates staging between two DTNs using containerized infrastructure. A staging service acts as a queue for data staging requests. Resource access is abstracted through a data adapter container. Depending on the DTN setup, the adapter will facilitate the copying. In this example the DTN servers accept Singularity containers and have SSH access thus the adapter deploys a UDT server in a Singularity container and a UDT client in a container on the client side. The containers are thorn down after the transfer.

This approach allows the customization of any two node communication. It also means that for every two nodes, a data adapter needs to be in place that has the required credentials for both nodes.

F. State Units

Since function containers are stateless, state is managed separately. This can be done through persistent containers that retrieve and update state objects being used by the functions trough the message bus. A state unit can be considered as a generic key/value store which any component can use to maintain some form of state.

Event objects also maintain state. The primary state of the event is a heuristic of past event values which is important for guard functions to to decide whether to trigger actions or not. Such event objects can be maintained by different state units thus allowing for a distributed state management system.

G. Resource Units

Compute happens on resources which can vary from HPC sites, cloud, IoT devices and private hardware. For this reason an abstraction is needed for different types of resources. This is also done through container adaptors. For example, user access to an HPC site can be containerized to use SSH and interface with the job queue system on site such as SLURM. Adaptors abstract the underlying infrastructure while exposing a common interface to the coordination runtime. In our case the adaptors have types where each type has its own message bus protocol. E.g. a container submission adaptor would expose methods such as *start, stop* container on the message bus irrelevant of how the actual adaptor will implement it. Similarly, a data transfer adaptor will expose methods such as *send, receive* while the adaptor itself might use gridFTP or UDT for the actual data transfer.

H. Coordination Unit

Tying everything together is a coordination unit. The responsibility of the coordination unit is the execute program definitions from section III-A. The program describes the application flow through events and actions whereby actions are performed by containerized functions. Combining events with dataflow model allows compute containers to be scheduled only when data is ready to be processed. The core of the coordination unit is an event loop which tracks events. In our model, the coordination unit issues requests on the message bus for functions to execute and process data. The runtime is responsible to schedule containers. For this reason an infrastructure unit keeps track of running units and deploys containers on demand. The infrastructure unit is also responsible to setup the initial infrastructure i.e. the adaptor containers that allow access to different resources. It also discovers compute units and listens on the message bus for function requests and decides if the request can be satisfied with current running compute containers or if new instances of the compute container should be initiated. The infrastructure unit is elastic; the coordination unit can submit many function requests creating a backlog of requests. One compute function can only process so many requests. The infrastructure unit detects the backlog and instantiates more compute replicas to reduce the backlog.

IV. Discussion

In this paper we outlined the challenges faced by modern scientific applications. the challenges stem from all layers of the stack. Resource pools are dispersed and heterogeneous on many fronts including access methods, software stacks,

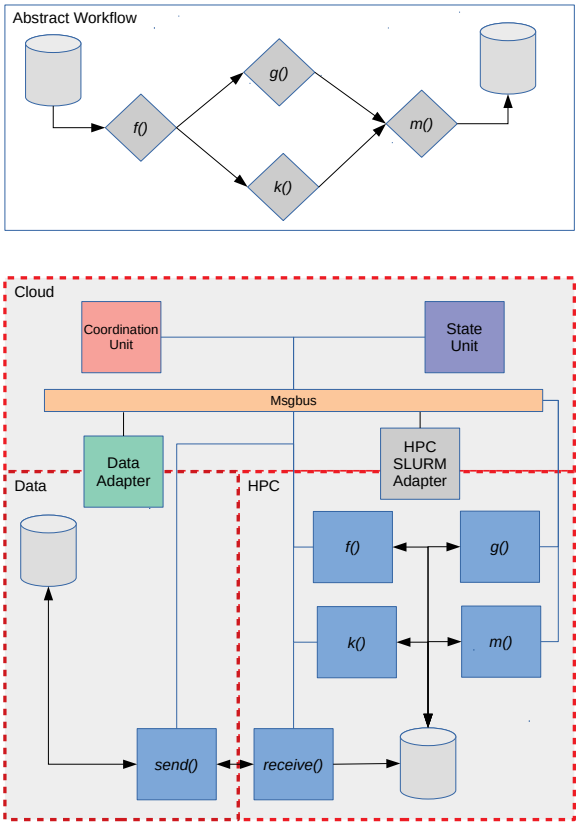

Fig. 4. Mapping of a workflow to a micro-infrastructure which is composed of cloud resources for coordination, HPC for compute using Singularity containers and DTNs for data transfer and staging.

hardware, network and performance. Furthermore, the complexity of the infrastructure is compounded by the different virtualization levels, from virtual machines to containers with each layer also compromising of a management and scheduling routines. At the apex of the infrastructure layer is a hyper-converged layer. In such a layer compute, storage and network are virtual and dynamically programmed. This dynamicity, although powerful, adds more complexity into mapping and coordinating applications. At the same time applications are also more diverse in their computing model archetype. AAN training-based applications are conceptually different than simple batch-based applications and therefore scaling mechanisms and coordination would be specific to each application.

All this points to the way applications use the underlying layers of infrastructures differs between application archetypes and therefore applications need their own micro-infrastructure which encapsulates the complexity of the full stack. Our approach in this paper outlines the basic components that are needed to create such isolated, portable and scalable micro-infrastructures.

References

[1] EuroHPC, https://eurohpc-ju.europa.eu/

[2] Exascale computing project, https://www.exascaleproject.org/

[3] Low-Frequency Array, http://www.lofar.org

[4] Ian T. Foster. 2001. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. In Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing (Euro-Par '01), Rizos Sakellariou, John Keane, John R. Gurd, and Len Freeman (Eds.). Springer-Verlag, Berlin, Heidelberg, 1-4.

[5] Linton Abraham, Michael A. Murphy, Michael Fenn, and Sebastien Goasguen. 2010. Self-provisioned hybrid clouds. In Proceedings of the 7th international conference on Autonomic computing (ICAC '10). ACM, New York, NY, USA, 161-168. DOI: https://doi.org/10.1145/1809049.1809075

[6] Ngoc Minh Tran and Lex Wolters. 2011. Towards a profound analysis of bags-of-tasks in parallel systems and their performance impact. In Proceedings of the 20th international symposium on High performance distributed computing (HPDC '11). ACM, New York, NY, USA, 111-122. DOI=http://dx.doi.org/10.1145/1996130.1996148

[7] Rdiger Berlich, Marcel Kunze, and Kilian Schwarz. 2005. Grid computing in Europe: from research to deployment. In Proceedings of the 2005 Australasian workshop on Grid computing and e-research - Volume 44 (ACSW Frontiers '05), Rajkumar Buyya, Paul Coddington, Paul Montague, Rei Safavi-Naini, Nicholas Sheppard, and Andrew Wendelborn (Eds.), Vol. 44. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 21-27.

[8] Martn Abadi, Michael Isard, and Derek G. Murray. 2017. A computational model for TensorFlow: an introduction. In Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2017). ACM, New York, NY, USA, 1-7. DOI: https://doi.org/10.1145/3088525.3088527

[9] Zaharia, Matei and Chowdhury, Mosharaf and J. Franklin, Michael and Shenker, Scott and Stoica, Ion. (2010). Spark: Cluster Computing with Working Sets. Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. 10. 10-10.

[10] Eric A. Brewer. 2015. Kubernetes and the path to cloud native. In Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15). ACM, New York, NY, USA, 167-167. DOI: https://doi.org/10.1145/2806777.2809955

[11] Gannon, Dennis and Sochat, Vanessa. (2017). Singularity: A Container System for HPC Applications.

[12] Dirk Merkel. 2014. Docker: lightweight Linux containers for consistent development and deployment. Linux J. 2014, 239, pages.

[13] Cushing, Reginald and Belloum, A and Bubak, Marian and de Laat, Cees. (2015). Towards a data processing plane: An automata-based distributed dynamic data processing model. Future Generation Computer Systems. 59. 10.1016/j.future.2015.11.016.

[14] Aleksandar Prokopec, Philipp Haller, and Martin Odersky. 2014. Containers and aggregates, mutators and isolates for reactive programming. In Proceedings of the Fifth Annual Scala Workshop (SCALA '14). ACM, New York, NY, USA, 51-61. DOI=http://dx.doi.org/10.1145/2637647.2637656

[15] Desai, Ankush and Gupta, Vivek and Jackson, Ethan and Qadeer, Shaz and Rajamani, Sriram and Zufferey, Damien. (2013). P: Safe Asynchronous Event-Driven Programming. ACM SIGPLAN Notices. 48. 10.1145/2499370.2462184.

[16] H. Fernndez, C. Tedeschi and T. Priol, "A Chemistry-Inspired Workflow Management System for Decentralizing Workflow Execution," in IEEE Transactions on Services Computing, vol. 9, no. 2, pp. 213-226, March-April 2016. doi: 10.1109/TSC.2013.27

[17] Reginald Cushing, Adam Belloum, Marian Bubak, Cees de Laat, Towards a data processing plane: An automata-based distributed dynamic data processing model, Future Generation Computer Systems, Volume 59, 2016, Pages 21-32, ISSN 0167-739X, https://doi.org/10.1016/j.future.2015.11.016.

[18] D. Turi, P. Missier, C. Goble, D. D. Roure and T. Oinn, "Taverna Workflows: Syntax and Semantics," Third IEEE International Conference on e-Science and Grid Computing (e-Science 2007), Bangalore, 2007, pp. 441-448. doi: 10.1109/E-SCIENCE.2007.71

[19] E. Deelman et al., "The Evolution of the Pegasus Workflow Management Software," in Computing in Science & Engineering, vol. 21, no. 4, pp. 22-36, 1 July-Aug. 2019. doi: 10.1109/MCSE.2019.2919690