DATA-CENTRIC COMPUTING ON DISTRIBUTED RESOURCES

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. D.C. van den Boom
ten overstaan van een door het college voor promoties ingestelde
commissie, in het openbaar te verdedigen in de Agnietenkapel
op woensdag 4 november 2015 te 10:00 uur

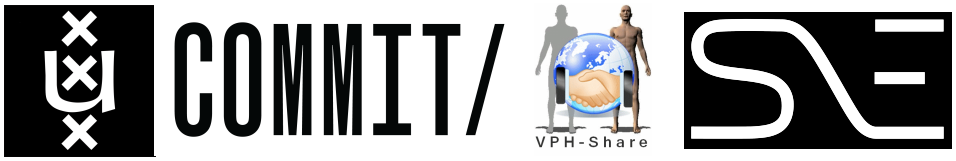
door

Reginald Steven Cushing

geboren te Pieta, Malta

Promotor:	Prof. dr. M.T. Bubak	Universiteit van Amsterdam
Promotor:	Prof. dr. ir. C.T.A.M. de Laat	Universiteit van Amsterdam
Co-promotor:	Dr. A.S.Z. Belloum	Universiteit van Amsterdam
Overige Leden:	Prof. dr. P.W. Adriaans	Universiteit van Amsterdam
	Prof. dr. H. Afsarmanesh	Universiteit van Amsterdam
	Prof. dr. ir. H. Bal	Vrije Universiteit Amsterdam
	Prof. dr. I.T. Foster	University of Chicago
	Dr. P. Grosso	Universiteit van Amsterdam
	Prof. dr. M. de Rijke	Universiteit van Amsterdam
	Prof. dr. P.M.A. Sloot	Universiteit van Amsterdam

Faculteit der Natuurwetenschappen, Wiskunde en Informatica



This research was carried out at the University of Amsterdam within the System and Network Engineering (SNE) group. This work was funded by COMMIT and VPH-Share projects.

CONTENTS

CHAPTER 1

MOTIVATION AND RESEARCH OBJECTIVES

1.1 Vision of Data Science

The deluge of data [1], information explosion and big data are terms used to denote the contemporary phenomena whereby the volumes, velocity and variety of data are outpacing the development in infrastructure [2]. The total volume of data in 2013 was estimated at 4 zettabytes (10^{21} bytes) and it is expected to double every 2 years (at least up till 2020 [3]). In 2014, every minute of Internet saw 1.5 petabytes (10^{15}) of IP data being transferred. These challenges in data are the main driving forces behind our thesis in which we study the future of scientific distributed data processing in the context of the fourth paradigm [1] and recently formulated idea of the 3^{rd} computing platform defined by IT technologies including: cloud and virtualization, big data, mobile devices, and social technologies [4].

Before delving into the intricacies of data processing, it is worthwhile to take a step back and look at the essence of data as a fabric of humanity. The inception of data may be traced back to the earliest humans who recognized that information can be passed down generations through writings and paintings. This human-specific trait is, most probably, one of the cornerstones in our achievement as humans. It might be safe to say that throughout history the volume of data, knowledge and information were always on the increase. We can argue that the constant increase in data meant that data was always somewhat *big* and drove innovation in finding new ways of recording and disseminating information from papyrus to scrolls to books to printing presses and to computers. Some inventions such as the printing press had so much of an influence in information dissemination that it is widely regarded as one of the turning points in human progress. Similarly the 20^{th} century will go down in history as yet another milestone in information technology with the invention of the computer as a data capture, recorder and above all a data processor. We are nowadays capturing so much data that we are constantly overwhelmed with the deluge of incoming data from the zillions of sources. The storing, processing and making sense of it are major challenges. In this context, we are still in the prehistoric era of digital data with centuries ahead of us for innovation in this field.

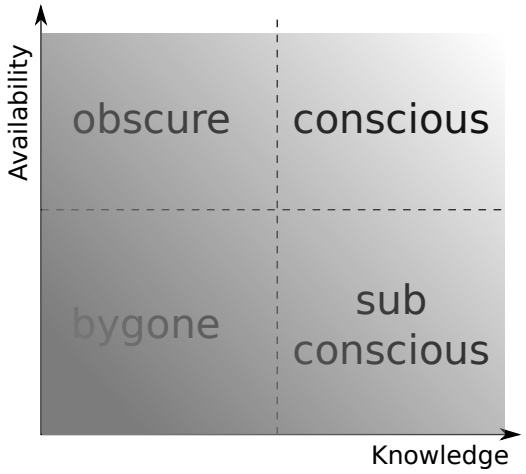


Figure 1.1: High-level taxonomy of data on a knowledge and availability axis. Data such as *trending* data is, typically, highly available and contains a certain level of information. Data for which we have knowledge about but, is not accessible, falls under the subconscious category e.g. Google's past search indexes. Obscure data can be data which is freely accessible but we have no interpretation of it e.g. encrypted information. Bygone data is data that has been lost and irretrievable.

In the greater scheme of data we can categorize data into 4 super categories which we call the taxonomy of data and illustrated in Figure 1.1; conscious data, subconscious data, obscure data, and bygone data. The conscious data is what is currently around us and being used, processed and researched. Such examples would be the accessible Web. The subconscious data is archived data, data that is not readily accessible and not being processed. The main difference between the conscious data and the subconscious is that the latter has a limited impact on future data discoveries due to its limited accessibility and availability. Narrowing the gap between these two categories increases the knowledge base and value of data (e.g. the ability to correlate data sets).

Obscure data is data that has lost its meaning. Ancient Egyptian hieroglyphic alphabet was obscure until the discovery of the Rosetta stone. Similarly, the undeciphered linear **A** writing system[1] is still obscure. The bygone data is data that we know existed but has been lost completely. The taxonomy of data is a starting point in reasoning about big data. The axis illustrate the gaps in data and the challenges we face today; knowledge and availability. Knowledge is, in part, the result of data processing while better availability is the result of data management, storage and infrastructure.

[1] `http://www.britannica.com/EBchecked/topic/342055/Linear-A-and-Linear-B`

The data availability challenge is compounded by the volumes of data we are experiencing. Networking technologies are constantly being challenged to move more data in shorter time spans. The ease and seamless movement of data is paramount for the future of data science since research collaborations necessitates the gathering, processing and redistribution of ever increasing data.

The data infrastructure of the future need to be smart by also using the knowledge in data to their advantage to be able to handle the predicted volumes of data. Data semantics and knowledge representation are still a major challenge in computer science. For example, how can we efficiently represent and search knowledge and information from multi-domain data? We know that the human brain can achieve this goal but as to what data model is used in our brain is still an active area of research [5]. One of the efforts in adding knowledge in the form of semantics to data is the Linked Open Data (LOD) [6] effort where data is structured using semantic tools and published online. The semantic layer allows the data to be linked to other published data and reasoning engines can be used to infer new knowledge from data.

Volumes and velocity of data are not the only data challenges; data variety is also a challenge and is expected to increase especially with the Internet of Things (IoT) [7]. The volumes and variety of data that overlap between research fields is a catalyst for interdisciplinary research. For example, epidemics research involves social behavior sciences and virology. The latter can be combined in computer science to study complex networks and information dissemination through networks [8].

Through infrastructure development, new data processing models and semantic models data increasingly becomes more accessible and networked. Such development is what will shape the future of data science. The high presence of data on the Internet makes the Web as the medium of data where knowledge is accessed, processed and preserved through generations. Our main objective in this thesis is to study the future of data processing and this is tackled from various angles including the infrastructure and the abstract level. These attack angles aim at narrowing the gap in the data taxonomy (Figure 1.1) whereby we focus on increasing knowledge and accessibility of data in the context of data processing.

1.2 Research Objectives

The complexity in scheduling and underlying management routines makes optimal distributed computing a challenging task. The advent of big data increases the dimensionality of the problem whereby data partitionability, processing complexity and locality play a crucial role in the effectiveness of distributed systems. The flexibility and control brought forward by virtualization means that for the first time we control the whole stack from the

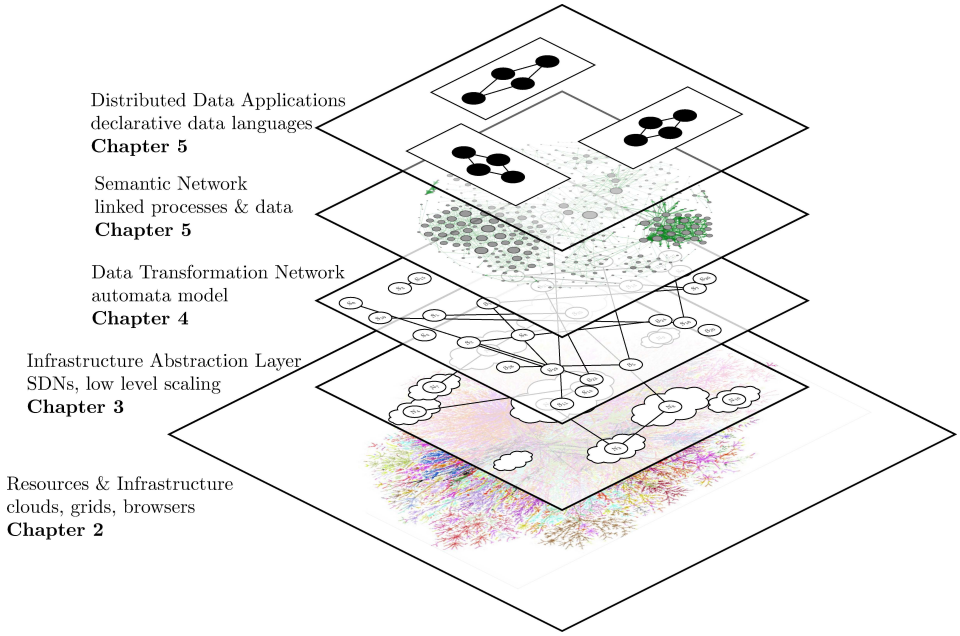


Figure 1.2: Layered system for data processing incorporating contributions from our work. Bottom layer: the connected Internet of resources including clouds, grids and browsers (Internet image as visualized by Barabasi [9]). Layer 2: The software stack directly managing the resources such as SDN controllers, low level application routines such as scaling and resource acquisition. Layer 3: data processing represented as a plane whereby data routing is based on data state transformations which makes this layer as a state machine for crunching data. Layer 4: Relations between data, processes and states allows for reasoning such as inferencing at a higher level. Layer 5: the application layer where distributed networked applications such as distributed data-centric workflows are designed and programmed based on the lower layers.

application down to the network layer but, to a certain extent, the best way to exploit this level of programmability still eludes us.

Our research hypothesis is that given the increasing intricacy and volumes in data processing and the dynamism of infrastructures, data-centric distributed computing should be tackled jointly from both the abstract data processing, semantic models and the infrastructure fronts so as to increase the knowledge and availability of data. As a consequence of the data deluge we hypothesize that the medium of data will outgrow the static devices and encompass also the networks thus making the infrastructure a data medium. Effective modeling of data allows better understanding of what data processing means. In our hypothesis a model should capture the essence of data transformations as abstract transferable knowledge amongst humans and machines.

In the context of the overall data science vision we believe such data models and distributed systems are indispensable for global data science where collaboration of knowledge in the form of data and services play an important role. To tackle our hypothesis from both fronts (Figure 1.2) we demarcate our research into four detailed research objectives:

---

1 Investigating new and emerging computing resources and ways how these resources could be exploited for data processing.

Any distributed system starts from the resources and infrastructure which provide the platform for data making it more accessible (Figure 1.1). Setups of such distributed systems have evolved greatly over the decades starting from Networks of Workstations (NoWs) as the early form of off-the-shelve clusters. Dedicated clusters followed which could churn much more computing power. The computing grid [10] was forged from the many dedicated clusters owned by research institutes which where not necessarily being used at full capacity all the time. This led to resource pooling where clusters are combined together into one huge resource pool and users can get access to a larger pool of resources. The grid, being distributed and made up of many autonomous administrative clusters, needed a complex piece of software (middleware) to manage resources and users. Although the grid is ideal for research institutes for sharing resources, intricate access rights and usage [11] meant that not everyone could make use of this resource. The cloud [12] partly solved this problem by offering infrastructure as a service. The cloud can be considered as a stack where each layer can be offered as a service; from bottom to top: the Infrastructure as a Service (IaaS), Platform as a Service, (PaaS) and Software as a Service (SaaS) [12]. This new paradigm poses new layers of complexity and thus new challenges. As part of this research objective we study new compositions in virtual infras-

tructures and how can we better exploit their full dynamic potential. Volunteer computing such as BOINC [13] also left its mark in distributed computing. The approach to volunteer computing as opposed to grid is a less structured one where desktop computers are used as the main compute power. With the 3^{rd} platform desktop computer are giving away to more mobile devices. As part of this objective we investigate new resource that fits the 3^{rd} platform. We investigate the use of web browsers as a potential resource. This study is motivated from the ubiquity of web browsers across platforms and the ever increasing browser performance. This objective form part of layer 1 in Figure 1.2.

---

2 Investigating various approaches to scaling data processing. We investigate service oriented approach to task farming, prediction-based data processing scaling whereby we look at data processing characteristics.

Organizing resources is one side of the story; the other side is the organization of computing through which data is processed. The processing of data adds knowledge to data which increases the data-knowledge vector in our taxonomy (Figure 1.1). A pronounced difference exits in the usage scenarios of such resources i.e. shared vs private resources. In shared resources the element of fairness plays a crucial role in the distributed system this means that any management of computing must take fairness into account. The simplest form of fairness in such systems is batch processing [14]. In a batch system access to resources is organized through FIFO queues whereby several priority queues may also exist. Simple batch systems assume that tasks are independent of each other and thus do not preserve task ordering. This can be problematic with intra-dependent tasks and thus many systems implement a form of Directed Acyclic Graph (DAG) scheduling where tasks are queued for execution dependent on some ordering described by a DAG. Similar to DAG, workflows are a higher level model of computation. Within Scientific Workflow Management Systems (SWMS), models of computations vary too; a dataflow model of computation will run tasks only when all data is available for that task while a Petri net model will run tasks depending on token transmission as a means of control flow. The shift towards data-centric computing means that data processing needs to be managed alongside the management of compute tasks. As part of our objective we investigate a dataflow model for the organization of data-centric distributed computing. The model allows us to study new approaches to data scaling whereby the management of compute tasks is done in relation to the data processing characteristics. This objective forms part of layer 2 in Figure 1.2

---

3 Elaboration of an abstract data processing model based on automata which aims at describing what data processing means rather then just how data processing is to be

carried out. This is motivated by the need to describe data in task-oriented distributed systems.

Distributed computing programming paradigms, in a way, can be broadly categorized in how declarative they are [15]. Common concurrent programming paradigms such as message passing (e.g. MPI) and concurrent object oriented (Actors) are imperative by nature whereby the distributed execution is planned out step by step as a set of commands which defines the *how* of the processing. More declarative approaches such as dataflow, workflows and MapReduce tend to focus more on the relationship between tasks e.g. dataflow models a data relationship between tasks while workflows model a work dependency between tasks. MapReduce can be considered as a simplified workflow with an implicit relationship between a *map* and *reduce* task. The controllers over the latter approaches can still be imperative. For example a SWMS can be imperative if control structures are part of the workflow planning. Related to the event-based paradigms, is an automata-based programming paradigm. In such a paradigm progress in a system progresses upon events but the event also changes the overall state of the system. The logic of an automata-based controller program is not imperative but reactive i.e. there is no start and stop of an execution but the system reacts on state changes. In this objective we study an automata-based model to, abstractly, describe data processing while also using the same model to build a data processing network. A data processing network acts, simultaneously, on both the knowledge and availability axis in our taxonomy (Figure 1.1). The network itself makes data available while the inbuilt processing adds knowledge to data. As part of the objective we study the data-centric processing network queuing characteristics such as backlog congestion. This objective form part of layer 3 in Figure 1.2

---

4 Reasoning at the semantic level can also find new functionality and resources and therefore it is worth studying the role of semantics in building networks of processes, the implications on data of having such networks and possible scenarios of data enrichment through open process networks.

The crux of data science is to acquire new knowledge and insights from data which increases the data-knowledge vector (Figure 1.1). Data processing allows us to process *old* data and transform it into something new which hopefully takes us a step closer to what we want to our results goal. Data processing is often thought of as a program which reads data and outputs data; most data processing models fall in this category. Post processed data could be valuable to others who can derive different results from such data for example by correlating it to a their own observations. This leads us to higher level of data science where data itself produces new knowledge from making inferences and other reasoning

on data. Semantics is fundamentally important [16] especially for data science and has been gaining momentum for the past decade or so. Semantics strives to give meaning to otherwise unmeaningful digital entities such as data or programs. Efforts such as the Linked Open Data (LOD) [6] use semantics to link datasets together in meaningful way. The latter is the foundation for a new way of data science where knowledge will not only be extracted through traditional processing but also through reasoning over many datasets. In this objective we investigate how semantics can be used to enrich data processing and how new data transformations can be discovered through semantically linking data processors.

The first and second objectives are related to new data processing infrastructures while the third and fourth will contribute to a new approach to development of data-centric applications. The integration of results from these investigation should lead to a new methods of distributed processing. An in-depth study of data processing needs to encompass all the spectrum of the area. The four objectives allows us to perform a detailed research over the spectrum of data processing from the abstract, semantic level to the execution and infrastructure level.

The methodology used throughout this research involves the study and analyses of the problem areas outlined in the above objectives, formulating models of solutions, designing artifacts that reflecting models, implementation of artifacts and testing and validating models through implemented artifacts.

1.3 Structure of Thesis

This thesis consists of 7 chapters and can be subdivided into 4 research strata corresponding to the 4 objectives which make up the layered diagram in Figure 1.2. Chapter 2 addresses the first objective where we study new methods of acquiring resources for distributed data processing. This chapter forms part of layer 1 in our diagram. In chapter 3 we address the second objective where we present methods and models for scaling data processing at a relatively low level. This chapter forms part of layer 2 in the Figure 1.2. In chapter 4 we address the third objective where we present a new data processing model in which data is a first class citizen. We show how the model lends itself well to creating a data processing plane. This objective forms part of layer 3 in our diagram. In chapter 5 we address the fourth objective whereby we study the role of semantics in distributed data processing. This chapter is part of the top two layers in Figure 1.2. In chapter 6 we present the evaluation and results of the methods introduced in the thesis. Finally, in chapter 7 we provide a summary about the research objectives and future work.

CHAPTER 2

EMERGING INFRASTRUCTURES FOR DISTRIBUTED COMPUTING

Distributed data processing is not possible without the backing of resources and infrastructure; by resources and infrastructure we mean the physical and virtual computers, networks and storage and the many-wares that make the resources immediately accessible. Away from the ever increasing need of raw power, a secondary and equally important attribute in resources are the flexibility, dynamism and malleability which enhance the overall usability of resources. In this chapter we focus on the latter attributes by investigating two frontier approaches to expanding the resources for distributed computing: first we study the viability of using web browsers as computing resource which offer a highly flexible way of acquiring resources such as access to GPUs and secondly, we study the dynamism and malleability of virtualization, and SDNs for distributed computing and data processing. The results presented in this chapter formed the bases of the following publications:-

- Reginald Cushing, Ganeshwara Herawan Hananda Putra, Spiros Koulouzis, Adam Belloum, Marian Bubak, and Cees de Laat. Distributed computing on an ensemble of browsers. *Internet Computing, IEEE*, 17(5):54–61, 2013.
- Rudolf Strijkers, Reginald Cushing, Marc X Makkes, Pieter Meulenhoff, Adam Belloum, Cees de Laat, and Robert Meijer. Towards an operating system for intercloud. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 2, pages 63–68. IEEE, 2013.
- Marc X Makkes, Reginald Cushing, Mikolaj Branowski, Adam Belloum, Cees de Laat, and Rob Meijer. Data Intrinsic Networked Computing. Manuscript to be submitted for publication in IEEE Internet Computing, 2015.

2.1 Introduction

The vast amount of available data throughout many science domains poses a challenge to handle and process it. For example, next generation sequencing will increase genome data by 10-fold every 18 months while the computational power is only expected to double every 18 months [20]. Handling and making sense of such data volumes and the rate at which they are being churned out is one of the current major areas of research related to big data. The lack of computing power means that data is increasingly being partitioned into archival data and current or *last mile* [21] data with much of the processing power focused on the latter two. New approaches to managing infrastructures can tap into new ways of harnessing computing power.

A quick look at the evolution of distributed infrastructures will take us back a few decades starting with the first supercomputers such as those from the Cray supercomputer. Networks of workstations (NOWs) followed which where made up with off-the-shelf computers. NOWs evolved into dedicated clusters which paved the way for grids. The complexity of hardware setups most often correlated to the complexity of software to manage them which led to new kind of software named middleware which manages applications, users and data on shared distributed infrastructure. Widely used examples are Globus/gLite [22] and Unicore [23]. With grids, infrastructure spans multiple administrative domain which makes management even more complex and not solely solvable by software. Management of administrative domains lead to consortia such as EGEE (now EGI [24]), foundations and partnerships such as PRACE [25]. With many grids, computing is institutionalized and the average user has a difficulty in gaining access to such resources (by design as a security concern). This changed with the emergence of cloud infrastructure which basically publicized computing.

Volunteer computing has, for a decade, managed to gather an unprecedented computing power. As of July 2012, the aggregated computing power provided by the participants in all projects using BOINC (Berkeley Open Infrastructure for Network Computing) comes to over 6 PetaFlops, with 2.4 million registered users, where 280,000 are active ones[1]. The volunteer computing model is somewhat different than other shared distributed computing platforms such as the grid. One of the major differences is the level of sharing; volunteer computing is a one-way model where users only accept jobs. The grid on the other hand is a two-way model where users are also able to submit jobs. The latter requires a higher level of organization and security.

Most native clients tend to focus on raw computing power by building systems aimed at reducing overhead and optimize throughput; such clients are highly efficient and boast

[1] http://boincstats.com/en/stats/-1/project/detail

features such as cycle stealing during idle CPU time as is done in BOINC and dedicated protocols for large data transports such as GridFTP. The major drawbacks to native clients involve: user participation and software portability. In [26], the authors investigate how resources could be freely acquired, unorthodoxly, through Google's App Engine. Similarly, our approach to harness computing through web browsers is somewhat unorthodox but at the same time it taps into a reservoir of ever increasing power.

2.2 Web Browser as a Resource

The ongoing explosion in mobile device and connected *things* which are giving rise to the Internet of Things (IoT) can be made easily accessible through technologies found in browsers particularly the JavaScript engine can pave the way to harness computing and functionality from such devices. Mobile devices, nowadays, are a bundle of sensors globally distributed with enough computing power to run simple non-trivial computing. Most mobile devices brandish a modern web browser which has access to most of the sensors. The combination of web browser, sensor pack and dispersion makes these devices an interesting form of computing resource.

The Internet browser is ubiquitous on the Web. At the heart of every Internet browser is a JavaScript engine which executes code on the client side. The shift towards Web 2.0 and the Ajax web development techniques drove performance boosts in JavaScript engine. In Google's V8 engine, a set of optimizations referred to as Crankshaft [27], dramatically improve compute-intensive JavaScript. Such optimizations include loop-invariant code movement and register allocation. JavaScript engines also found use outside the scope of browsing with projects such as Node.js [28] which is the JavaScript engine from Chrome and has proven its worth as a stand alone platform in scalable network applications which shows that the JavaScript language is taking a foothold outside the browser context.

Due to W3C standardization, code written in JavaScript is highly portable between browser implementations and therefore portable between different platforms. Standardization also makes JavaScript forward compatible. The advancements surrounding HTML5 also equip the browser with needed functionality for computing. Technologies such as *web workers*, *web sockets*, *local data*, *webRTC* and *webCL* are among the few new enhancements to the browser which aid in many expects of the browser experiencing.

Web workers are threads of JavaScript communicating over message passing as opposed to shared memory and are intended for data intensive computation in the background. The latter is what makes a browser most ideal for computing since threads allow and heavy weight computing to be shifted to the background without influencing the web page responsiveness. Web workers threads also offer an additional level of security since the threads are sandboxed and do not have access to the DOM whereby any thread would be

able to deface the web page. Another addition to the browser tool set is the web socket. This allows bi-directional, raw, full-duplex communication channels between server and client intended for real time communication. The communication is done without HTTP protocol overhead. Communication can be initiated from the server-side which makes it ideal for servers to pull information from the browser instead of waiting for clients to push data.

Another new communication mechanism is webRTC where RTC stands for real time communication. This addition allows real time communication between browsers which is intended for browser based video calls but can be used for communicating any arbitrary data. WebRTC offers functionality which was previously impossible to achieve using browsers and opens new doors for browser based applications. One such application is peerCDN [29] which creates a CDN between a hosting server and all clients surfing the website. Website content can be acquired through peers or server. WebCL is an API for OpenCL to perform computing on the GPU directly from the browser. All these latest features indicate that the web browser and the inbuilt JavaScript engines are on an upward trend towards better performance.

Using a browser as a computing node is not by any means without limitations. For example, whereas a native client can easily detect CPU usage and only use spare CPU cycles, a similar approach is almost impossible with current browser implementations. A solution where the user can control JavaScript through timers is possible by parsing jobs and automatically inserting controllable timeouts into tight loops. Another limitation is the same origin policy whereby browsers block a website from communicating to anything outside the originating domain so as to limit cross-site scripting attacks. This restriction has been somewhat eased with the introduction of the Cross-Origin Resource Sharing (CORS). CORS entail web servers to include an additional HTTP header (Access-Control-Allow-Origin) which is not always set by default. Without CORS any external data has to be proxied through the originating web server. CORS is an important feature as it allows the ability for remote storage access directly from a web browser which helps in distributing large input/output data. Other limitations include the lack of high performance JavaScript libraries such as GMP (for arbitrary-precision arithmetic) for C/C++. These limitations are overshadowed by the prospect of tapping into millions of browsers where each can perform a very small task (a trickle of computing) and together solve a much larger problem.

2.2.1 JavaScript Performance

Distributed computing using JavaScript engines immediately conjures arguments about its performance when compared to other traditional languages and compilers. This section presents some performance analyses of JavaScript when compared to GNU C. Figure 2.1

illustrates the performance ratios of typical algorithms run under Google's V8 JavaScript engine and GNU C. The five algorithms were sourced from [30] and executed under 32 bit architecture on an Intel E5500 processor. GNU GCC version 4.5.2 with optimization flags was used for compiling the C algorithms. The algorithms used for this analysis were: **Regex-DNA** is a string based algorithm which performs multiple regular expression pattern match and replace on DNA sequences; this algorithm uses the inbuilt string replace, match and length functions. **SpectralNorm** calculates the spectral norm of a matrix; the JavaScript algorithm uses *Float64Arrays* and inbuilt square root function. **K-Nucleotide** is a sort, search and counting algorithm for DNA nucleotides. **N-Body** is a simulation algorithm which models the orbits of the Jovian planets. **B-Tree** is a benchmark tailored around binary tree manipulations which include allocating, deallocating, and walking bottom-up binary trees.

For most examples in Figure 2.1 Google's JavaScript engines are slower than GNU C. Although this is not surprising, the results show that the performance ratio differs drastically between different algorithms. The Regex-DNA algorithm performed slightly better than C which is due to Chrome's highly optimized regular expression implementation (Irregexp). All other examples performed worse than GNU C. Most notable in these results is the performance discrepancy between different versions of V8. In most cases, the later versions of V8 perform better than the earliest version 1.3. Some drastic performance gains, as in the SpectralNorm example, are attributed to the introduction of *ArrayBuffer* and *Float64Array* data structures and V8's Crankshaft optimizations in the later versions. These results show the trend in JavaScript engine performance gains. With this upward trend in performance we argue that web browsers can indeed become a middleware for distributed computing in the near future.

In addition to the benchmarks shown in Figure 2.1, we performed benchmark tests with OpenCL and WebCL. With WebCL, browser computation can be accelerated using GPU hardware. Our results for a vector addition algorithm show that both perform equally well. This was to be expected, as the JavaScript acts as a wrapper to the WebCL code which is executed directly on the hardware.

2.2.2 Browser computing with WeevilScout

To demonstrate the principles of distributed browser computing we setup a prototype framework, WeevilScout, where users can donate as well as submit jobs to the browser network. Figure 2.2 depicts an overview of WeevilScout. The job queue database holds a list of jobs on the run-queue. The jobs are written in JavaScript and intended to execute on the collaborating browsers. Jobs submitted to the queue are JavaScript functions which are parsed and transformed by the web service before being put on the job queue. Listing

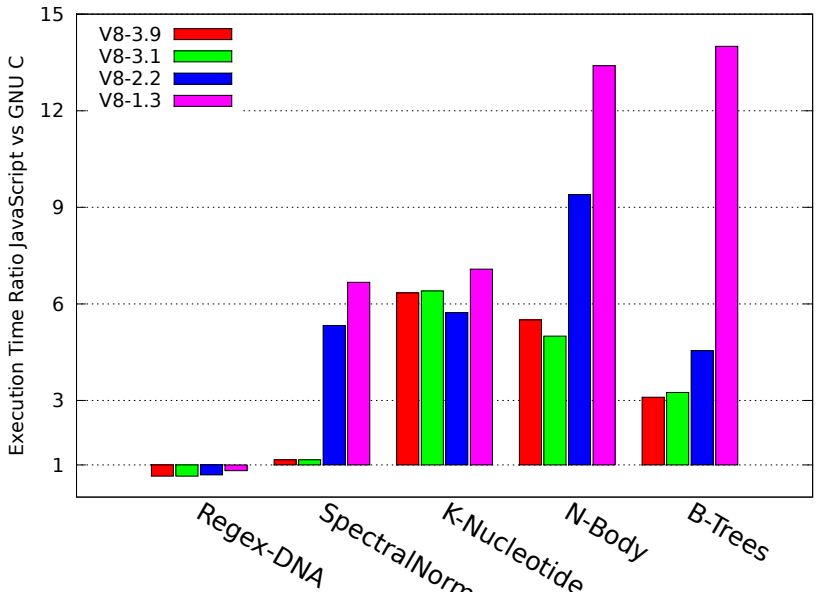


Figure 2.1: Ratios of execution times of a set of representative algorithms compiled and run with 4 versions of Chrome's V8 JavaScript engine to respective execution times with GNU C.

2.1 shows a simple job written in JavaScript which merely performs a matrix multiplication. For this function to be run on remote browsers it must be transformed into a web worker by the server side. The transformed function is shown in Listing 2.2. Web workers run in the background and are isolated from the main thread rendering the webpage thus communication between the web worker and the main thread is only possible through messaging.

When submitting a job, users can specify multiple values for each parameter (matrices for A and B), in which case the server back-end performs a cross product on the input parameters and generates a job for each product set. This allows data partitioning amongst a set of jobs where each job works on one parameter value set. After transforming the JavaScript function into a job, the server saves the job to a web accessible folder along with a XML description of the job which includes the parameters and the script.

The client part of WeevilScout is the website itself which also acts as the GUI (for job creation and submission) and execution platform. The client periodically polls the server for any new jobs needing processing; this is done with the XMLHTTPRequest API. When jobs are available on the queue, the server responds with an XML description of the job which contains the URL of the JavaScript job and the parameter set. The client parses the XML and sets up a new worker with the new job URL. The parameters are passed

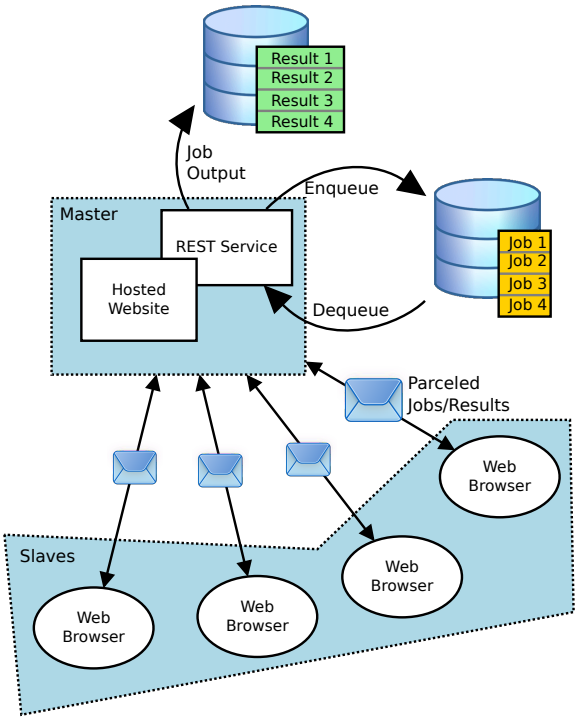


Figure 2.2: Explanation how a cluster of browsers is used to perform computation. The server side components form the master node where the REpresentational State Transfer (REST) service dequeues and enqueues jobs while a website handles user interactivity. Web browsers that load the website pull packaged jobs and send back job outputs to the REST service which in turn stores the results.

as a message to the worker and the start command is issued to start processing. Upon completion, the client returns results to the server and immediately requests a new job. If no jobs are available, the client idles before polling the server again for new jobs. Since most browsers cache downloaded scripts, running multiple identical jobs with different inputs will result in the code only being downloaded once which reduces network traffic especially in parametric study scenarios.

Security in such a distributed system can have many facets: from secure protocols to security in data handling. WeevilScout does not implement security as its sole purpose is to demonstrate how distributed computing can be achieved through web browsers. Nonetheless, all JavaScript jobs are executed in a sandbox within a JavaScript VM in the browser so client side security is as good as the browser implements it. Moreover, web workers provide further security since the JavaScript in a worker has no access to the Document Object Model (DOM) and therefore has no means altering the website.

The techniques used by WeevilScout can be easily used by any other website. This could lead to parasitic computing [31] when users are unaware that their browser is contributing to computing. This reveals a new dimension in future web security where websites can potentially steal computing from visiting users.

```
function weevil_main(A,B){
  var mA = JSON.parse(A);
  var mB = JSON.parse(B);
  var result = [];
  for (i in mA){
    result[i] = [];
    for( var j in mB[0]){
      var sum = 0;
      for(var k in mA[0]){
        sum += mA[i][k] * mB[k][j];
      }
      result[i][j] = sum;
    }
  }
  return(JSON.stringify(result));
}
```

Listing 2.1: A simple JavaScript matrix multiplication function.

Some of the features that have not been addressed in WeevilScout are data management and user-controlled CPU usage. Techniques such as remotestorage (http://remotestorage.io) can give browsers direct access to large remote datasets. This coupled with HTML5 local storage capabilities can make computing persistent so that not all progress is lost when the browser is closed or the Internet connection is lost. Since browsers do not provide any means to throttle CPU usage by JavaScript, a solution is possible by parsing jobs and automatically inserting controllable timeouts into tight loops. The volunteer can then control such timeouts and therefore indirectly control CPU usage by slowing down such loops.

As an example that proves Internet browsers are quite capable of distributed computing, we present a typical scientific study from bio-informatics. This study performs protein sequence alignments using the Needleman-Wunsch algorithm implemented in JavaScript[2]. Sequence alignment is a common method employed in bio-informatics as a way to order sequences of proteins and DNA to identify areas of similarity that could be attributed to

[2]http://opal.przyjaznycms.pl

```
17  self.addEventListener('message', function(e) {
18    var data = e.data;
19    switch (data.cmd) {
20    case 'start':
21        weevil_main();
22        break;
23    case 'stop':
24        self.close();
25        break;
26    }
27    function weevil_main() {
28        var A = e.data.A;
29        var B = e.data.B;
30        var mA = JSON.parse(A);
31        var mB = JSON.parse(B);
32        var result = [];
33        for (i in mA) {
34            result[i] = [];
35            for (var j in mB[0]) {
36                var sum = 0;
37                for (var k in mA[0]) {
38                    sum += mA[i][k] * mB[k][j];
39                }
40                result[i][j] = sum;
41            }
42        }
43        self.postMessage((JSON.stringify(result)));
44    }
45  },
46  false);
```

Listing 2.2: A transformed version of the function in Listing 2.1 into a job that can be distributed to web browsers. The transformed function is wrapped into a web worker with control commands start and stop added to the function in lines 19 and 22. Parameters A and B are remapped from the worker parameters in lines 29 and 30. The return call is remapped to a *postMessage()* in line 42.

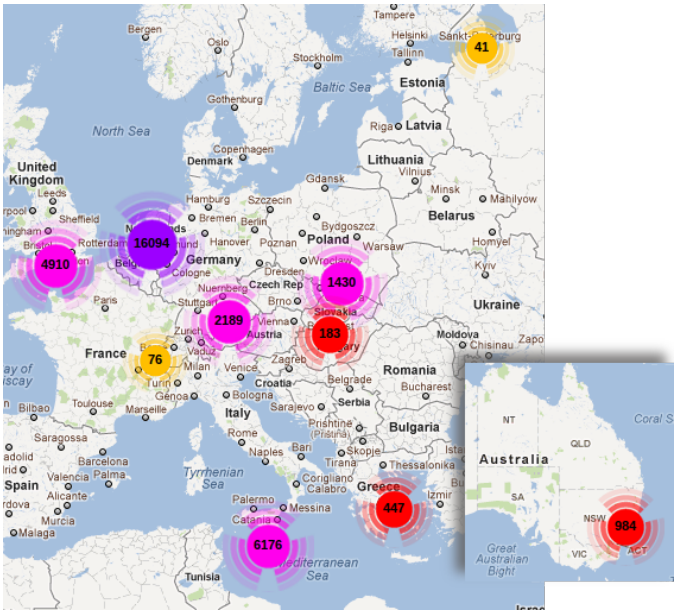


Figure 2.3: The distribution of over 30,000 bio-informatics tasks on the global cluster of browsers.

some relationship between the sequences. The data for the alignments was obtained from the UniProtKB[3]. We set out to perform more than 30,000 alignments. Each alignment formed a single task and each connected browser performed many of these tasks over a 5 hour time window. The cluster of browsers was assembled simply by providing the URL[4] to colleagues and friends through social media. The simplicity of joining the network meant that we could build a sizable cluster (45 simultaneous browsers) in a short time. This cluster was enough to perform the required computation in the 5 hour time window. The total processing time for completed tasks within this time window amounted to, approximately, 135 hours thus the Web provided 135 CPU hours for computing. Figure 2.3 illustrates how the tasks were globally distributed. This indeed shows that the framework allows a truly distributed computing platform in a very simple way. The heterogeneous composition of the cluster in this experiment (Linux 44.4%, Windows 40.7%, Mac 14.8%, Chrome 58%, Firefox 37%, and Safari 5%) shows the intrinsic portability of JavaScript between different operating systems and browser implementations. Figure 2.4 depicts the power of the cluster in combination to the number of completed tasks. Each browser that connects is benchmarked using a large matrix multiplications. The aggregated power of the cluster is the result of the summation of the estimated performance for each browser in

[3] `http://www.uniprot.org/`
[4] `http://elab.lab.uvalight.net/weevilscout/`

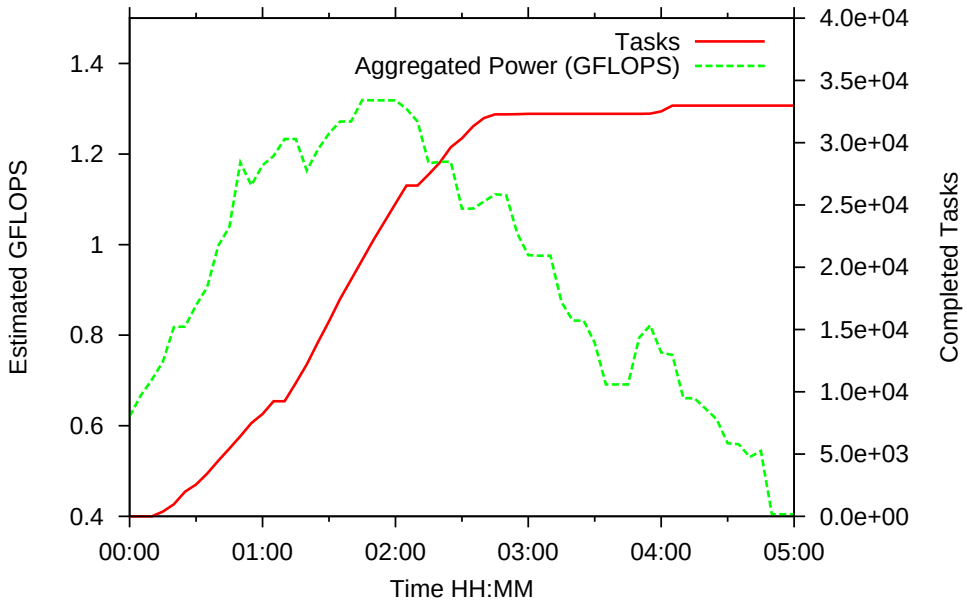


Figure 2.4: In our example a FLOP is calculated as 1 multiplication and 1 addition. The FLOPS metric is an estimate taken when the browser is connected. The estimate is subject to many variability since the browser and the web page are not necessarily the only running entities on the node. The purpose of this is to demonstrate a relative power metric for browsers and is not meant as a comparative to other benchmarks. Aggregated power (the result of summation of all connected browser computing power) on the left y-axis shows the cluster power during the experiment time window. Tasks on the right y-axis shows the number of completed tasks. As the cluster grows in power, the rate of completed tasks increases. As the cluster power diminishes (around 3hrs) the rate of completed jobs grinds to a halt.

FLOPS. This gives us an idea of the processing power of the cluster. One can immediately identify a correlation between the cluster power and the number of tasks completed. This is attributed to the malleable nature of the bio-informatics application which is adjusting to the increasing resource pool.

The above example is based on 100% JavaScript. WebGL and WebCL allow the browser to access the underlying GPU hardware. Although this is not yet an out-of-the-box working standard, our experiments in a controlled environment with AMD OpenCL driver and Nokia WebCL plug-in for Firefox 17 using a simple FLOP metric constantly resulted in a $3\times$ performance gain using WebCL as opposed of using pure JavaScript.

2.2.3 Browsers for Scientific Computing

What browsers have to offer better than traditional compute infrastructures is the flexibility, ease and mobility of its user/compute nodes. With WebCL and access to the plethora of sensors on mobile devices, browsers might as well find a place on the fringes of distributed data-processing.

A cluster of Internet browsers is inherently dynamic in nature. The size varies drastically, the network connectivity also varies between browsers and server. These characteristics define the class of applications which are applicable to such a framework; malleable applications are these that can adopt to the changing computing environment, particularity the changing cluster size. An important aspect of malleable applications is that, given a dataset input, the malleable task can operate on any subset of the dataset thus data can be unequally partitioned between many instances of the same application running inside the browsers. Malleable applications include: Monte Carlo simulations, parametric studies, algorithms based on directed acyclic graphs, data-parallel analysis algorithms and others.

In eScience, tasks are commonly expressed in worklfows. As part of our study [32] we showed how workflows can be expressed in browsers. A dataflow model was implemented for browsers where data dependent tasks are organized in a directed acyclic graph (DAG) and tasks are executed on browsers when inputs are available. This workflow construct allows for more complex execution on browsers since simple tasks can be organized in one bigger distributed program.

Recently, scientific computing is becoming more and more data intensive while browsers do not have means to deal with large data sets, so large files require appropriate partitioning [33]. This allows a browser to only handle a small part of the data at a time and also allow the computing to scale well with a very large cluster of browsers.

With the current explosion of small devices connected to the Internet (Internet of Things (IoT)) [7], web browser technology can offer a highly portable platform for such devices. We hypothesize that the Internet browser is a future computing platform with the potential

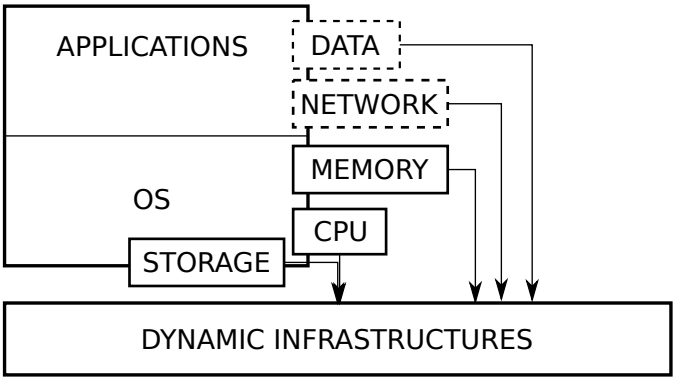


Figure 2.5: VMs expose the traditional OS metrics such as CPU load, memory and storage. Virtual infrastructures can better optimize networks by using additional telemetric from applications such as data processing complexity and communication application peers.

of transforming the Internet with its unused, free computing power, into a true distributed computer.

2.3 Intercloud as a Computing Infrastructure

Infrastructure as a Service cloud (IaaS) computing enables users to dynamically create virtual application-specific computing environments. Such virtual environments can be globally distributed as would be the case when acquiring resources from different providers. The dynamic and fluctuating characteristics of cloud resources entails that we need a different approach to manage data processing on clouds than other traditional resources such as clusters.

Cloud providers manage their resources with a so-called cloud operating system. These operating systems, e.g., OpenStack [34], provide a management layer to allocate virtual machines on computer clusters on behalf of users. Public cloud providers expose their APIs to users with the same goal, but use a cost model to constrain resource usage. By utilizing the APIs of public clouds users can gain access to more computing resources than available in a single cloud. Furthermore, with access to multiple public clouds, users can choose which provider(s) is best suited for the application.

2.3.1 New Generation of Applications

The advent of dynamic infrastructures presents us with new challenges and new opportunities of computing. Programmability allows the infrastructure to adopt to the application. Adaptation means that the infrastructure can scale, migrate and provision networking de-

pending on the input from VMs such CPU and memory load. The common approach to monitoring cloud applications is to treat them as either transaction applications such as web applications or batch applications where jobs are queued [35]. Adaptation by the infrastructure is thus based on such gathered metrics where scaling techniques would load balance network sessions or database threads in transactional applications or load balance job queues for batch applications [36]. To fully exploit the dynamism offered by virtual infrastructures such as Software Defined Networks (SDNs), more metrics are needed about the applications running on a virtual network. Applications are becoming increasingly networked i.e. no application is an island and the correct provisioning of an infrastructure would need to grasp the extent of the networked applications. Such metrics would give an insight into how data is communicated between cloud applications thus paving the way for data/compute temporal and spacial optimization. A further insight into networked applications is their performance predictability on incoming data. Big data processing scenarios often entails data queuing (queuing data for processing); performance of such data processing can vary considerably among seemingly identical VMs (different hosts, different virtual networks, etc). Profiling an application in combination with its VM and the input data the underlying infrastructure can deduce the application/data/VM combo complexity (liner, exponential, etc) which gives the infrastructure a head start on provisioning.

2.3.2 Data Defined Networking

The dynamism in both infrastructure and application management quickly made us realize that having one entity such as an application manager to control everything from distributed application scheduling to infrastructure optimization was not a sustainable and scalable solution. Our believe is that a separation of concern is needed between application managers or middlewares and infrastructure controllers. As described in Section , through adequate exposure of metrics from the application layer, the infrastructure can snoop for this information an optimize itself. What, in effect, we are creating is an interface between application an infrastructure where a symbiotic relationship is created. The infrastructure is in continuously mold to the application. The end result is a distributed system networked in a way the mirrors the data needs to the application. The formation on this network is what we dub the Data Defined Network (DDN) as it is one level up from an Software Defined Network (SDN). In SDN software control loops have the ability to create networks, the application can provide the *data map* thus allowing the SDN controller build a DDN as in Figure 2.6.

The two layers of a DDN capable system are, as described in [19], the application middleware and the infrastructure controller. The infrastructure controller is based on Internet factories [37] where controllers can create any kind of network topology on top of

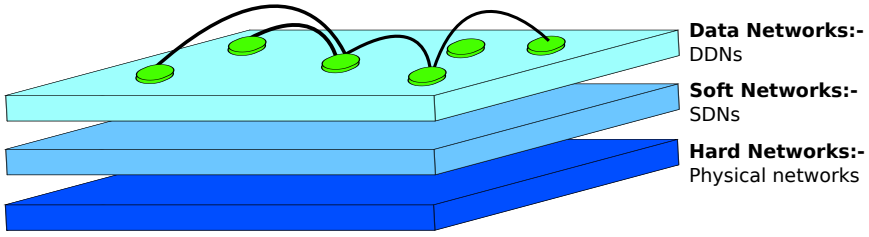


Figure 2.6: Exploiting SDNs coupled with exposing application internals such as application peers a DDN can be created which is a network defined specifically for a particular data processing scenario.

public/private cloud providers. In addition, other control loops are able to interface into the application middleware layer. The middleware layer provides data processing performance metrics, applications currently running and data routing information (i.e. where is data coming from and where is it going). The middleware exposes some control interfaces which allows external controllers to have finer control on the application. These includes deploying, starting and stopping applications. Through these primitives a controller can make noninvasive optimization of the distributed application by controlling, solely, the network or can be invasive and dictate which application is hosted where.

2.3.3 Application Managers as Middlewares

Early approaches to exploit the then emerging networked applications on cloud resources was the integration of a workflow manager to a cloud resource manager. The workflow manager is a client/server run-time system developed to support dynamic use of computing resources. It hides the details of execution and management of applications from users. In the client, a user composes a workflow of applications with a data flow Model of Computation (MoC). The server uses the workflow description to manage application execution. The workflow manager is made up of a task scheduler and a task mapper.

The *task scheduler* determines the execution order of tasks with regard to data dependencies and supports farming (run the same computation on partitioned data) and parameter sweeps (run computations with different parameters on the same data), which require separate data dependency processing.

The *task mapper* takes the results of the task scheduler and matches the tasks to available resources represented via *resource handler* objects, which are adapters for computing services. Resource handlers are tightly coupled to the interface of the computing service. Resource handlers include: a Grid resource handler and Transient Grid (TGrid) resource handler, which is a Grid abstraction for a cluster of virtual machines. Different from the

Grid resource handler, TGrids are temporary. So, the TGrid resource handler implements additional mechanisms to register and deregister TGrid resources in the task mapper.

The task mapper executes the workflows as fast as the budget allows. It can also implement other scheduling algorithms including a round-robin, least-used resource, and budget-aware scheduling for graceful and efficient (de-)allocation of resources. Depending on the prioritization of handlers, the task mapper will send a request to the TCM to create more resources for submitting tasks. The task mapper also maintains two budgets: a main budget and a reserve budget. The main budget decreases by a pre-defined rate for every resource handler. Once the main budget is over, no more tasks are mapped to the resources. The second budget is a reserve budget which is used to allow tasks to terminate after the main budget has be fully consumed avoiding restarting of unfinished tasks when cloud resources become unavailable due to budgeting.

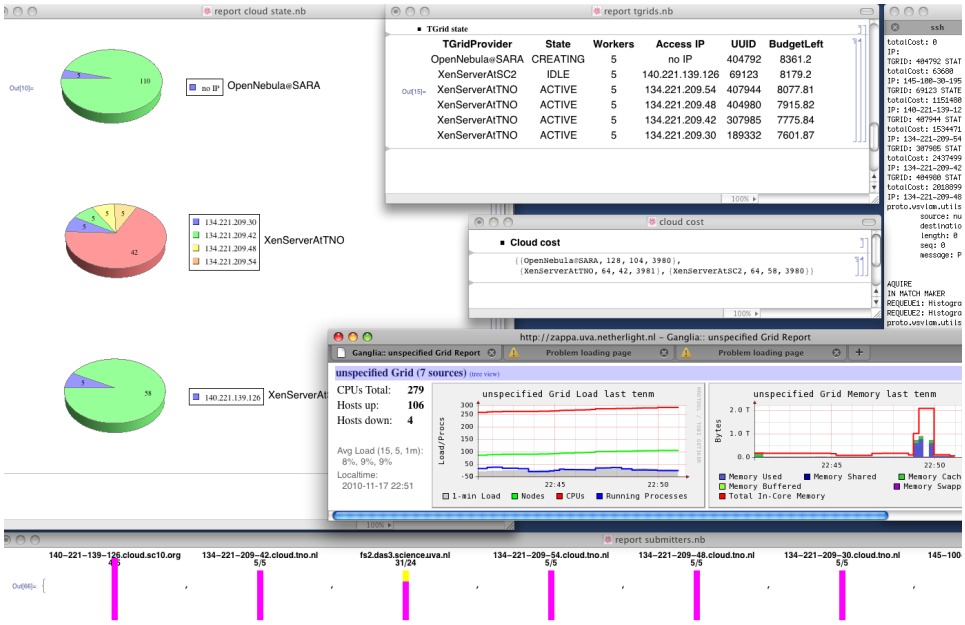


Figure 2.7: Monitoring the execution of the image processing workflow on the virtual infrastructure. Virtual clusters are being created on-demand initiated in two phases: a request from the workflow manager and the actual creation from the infrastructure controller. Further details about the workflow manager part are described in chapter 3.

This concept was presented at Scale 2012 challenge [38]. As a driving application we created an image processing workflow (described in detail in chapter 3). The workflow is meant to exploit the scaling techniques employed by the workflow engine for dealing with farming and parameter sweeps (see chapter 3). The initiated on-demand cloud clusters can

be seen in Figure 2.7, As the workflow engine starts scaling the workflow modules, more resources are needed and thus the infrastructure initiates on demand clusters and networks.

The test bed included three clouds: a 128 core OpenNebula cloud at SARA high performance computing center, Amsterdam, The Netherlands, a 64 core Intel XenServer cloud at TNO, Groningen, The Netherlands, and a 24 core XenServer cloud in New Orleans, US. The OpenNebula cloud used KVM virtualization technology, while the XenServer clouds used Xen virtualization technology). We also used the 41 node (164 core total) Amsterdam site of the DAS-3 distributed compute cluster as a grid resource. The system software ran on two machines: a workflow application server and a server running the infrastructure controller.

2.4 Summary

In this chapter we have demonstrated how distributed resources can be flexible. We showed the ease of setting up browsers as a thin middleware that can easily access resources and how access to GPUs from WebCL in browsers offers the potential of unlocking performance. Computing is simply amassed through URL sharing. We believe a resource such as a browser is not something that will replace traditional resources but is a resource that can extended the user base and locality of computing. An area worth investigating is mobile distributed computing. At face value, WeevilScout works on latest mobile operating systems but mobile devices pose several challenges for computing; lack of power supply being one of the major limitations. On the other hand, mobile devices are packed with sensors thus, the applicability of WeevilScout to mobile devices merits further investigation whether such a framework can be used to build sensor networks where JavaScript jobs collect and transform sensory data with minimum impact on the device.

Malleability and dynamism have also been discussed in the context of interclouds. We illustrated the need of a new generation of applications that can work in symbiosis with the underlying infrastructure. The programmability in the infrastructure such as SDNs paves the way for shaping the infrastructure to the application as opposed to the traditional approach of fitting the application to the resources. We described a middleware for interclouds based on a workflow manager. Workflow models are suitable for capturing application communication thus can model the application on an intercloud. The network programmability in the infrastructure can model the mirror the data-centric distributed application communication. We defined these types of networks as a DDNs which are a layer on top of SDNs. A DDN is an SDN setup specifically to model a data-centric distributed communicating application.

Although browser as resources paradigms and cloud computing paradigms are somewhat different they still be part of a wider data driven network. While virtual infrastruc-

tures can offer the raw computing power, browsers and JavaScript engines can form the sensor network around the core cloud computing. Mobile devices are becoming exceptionally good as mobile computers for example NASA's PhoneSats[5] project launched such devices into space and successfully beamed data back to Earth. It is not just in space that things are connected but everywhere around us, a phenomena known as IoT. We believe that technologies such as browsers are a step towards connecting such devices with the possibility to perform client side computing and increase the availability dimension of data in our taxonomy. In chapter 4 we introduce a data processing model and protocol which can integrate different heterogeneous resources into one compute platform.

[5] http://www.space.com/20772-nasa-phonesats-smartphone-satellites.html

CHAPTER 3

SCALING DATA CENTRIC COMPUTING

A core challenge in data-centric, communicating distributed computing is the scaling of data processing. CPU and memory load are not always indicative of the need for scaling in data-centric, communicating I/O intensive computing. Queued data can give us a better insight into scaling decisions. Simply put, the problem is how can we achieve speedup in distributed data processing workflows while keeping resource usage efficiency in check? This problem is the focus of this chapter. We tackle three distinct scenarios of distributed workflow execution. A service oriented approach to task farming which aims at better resource usage efficiency in task farming scenarios. A prediction-based scaling technique used in data-centric workflows aims at using data processing prediction metrics as an indicator to scale up resources. A fuzzy logic based approach which uses resource load metrics in combination to data processing prediction to create resource usage fairness between competing but collaborating tasks. The results in this chapter formed the bases of the following publications:-

- Reginald Cushing, Spiros Koulouzis, Adam Belloum, and Marian Bubak. Applying workflow as a service paradigm to application farming. *Concurrency and Computation: Practice and Experience*, 26(6):1297–1312, 2014.
- Reginald Cushing, Spiros Koulouzis, Adam Belloum, and Marian Bubak. Dynamic handling for cooperating scientific web services. In *E-Science (e-Science), 2011 IEEE 7th International Conference on*, pages 232–239. IEEE, 2011.
- Reginald Cushing, Spiros Koulouzis, Adam Belloum, and Marian Bubak. Prediction-based auto-scaling of scientific workflows. In *Proceedings of the 9th International Workshop on Middleware for Grids, Clouds and e-Science*, page 1. ACM, 2011.
- Reginald Cushing, Adam Belloum, Vladimir Korkhov, Dmitry Vasyunin, Marian Bubak, and Carole Leguy. Workflow as a service: an approach to workflow farming. In *Proceedings of the 3rd international workshop on Emerging computational methods for the life sciences*, pages 23–31. ACM, 2012.

3.1 Introduction

The shift towards data in eScience has given rise to the fourth paradigm in scientific discoveries [43] where it is envisioned that data analyses will play a central role in future discoveries. In data-centric distributed applications, managing data in its various forms plays a leading role in the system. A simple data processing cycle entails the data acquisition, processing, tracking and storage. Data can come from various sources which can be distributed beyond global (e.g. satellite data) and reside on access restricted infrastructures. Any start of data processing needs to start with data acquisition which means the efficient means of dealing with such dynamic sources. During processing data is in a *transient* state meaning that data from intermediate data processing is also available and needs to be managed for adequate processing (e.g. moving data closer to computing, partitioning data). A simplistic distributed data processing would entail manually partitioning data among a preknown number of distributed nodes. In workflows this approach is not always possible due the complexity of the workflow where data is processed in several stages thus the output of one stage needs to reshuffled to other nodes. Workflow execution creates a scenario where multiple datasets (the inputs for different stages of the pipelines) and different tasks (tasks at different stages of the pipeline) need to be managed on common resources.

Tools such as Scientific Workflow Management Systems (SWMSs) can play a vital role in accelerating discoveries by providing means for coordination of data-centric workflows where computation can automatically scale to meet the data demands. Workflows are commonly described as Directed Acyclic Graphs (DAG) where vertices represent computation tasks while edges represent dependencies between tasks. In SWMSs such as WS-VLAM [44], tasks also include a list of input and output data ports which, apart from the data dependency, also describe data channels between tasks. Tasks within a data-centric scientific workflow are often data dependent on each other (when edges represent data dependencies the graph is usually referred to as a dataflow) where each task can, potentially, be a data intensive task. Managing multiple data-intensive tasks in SWMS poses a coordination challenge since the progress of the whole workflow is easily hampered by the slowest task. Data-centric tasks follow a pattern of consuming data chunks, processing the data and output results. This cycle is repetitive and the amount of data chunks needing processing directly influences the execution time of the task.

A number of tools, such as Nimrod/G [45], NetSolve [46], Ninf [47], AppLeS [48], have been developed and offer scalable computing while hiding the low system level details. The way these tools expose such facilities vary from one tool to another.

Nimord/G [45] is a system which aims at scheduling parameter sweep studies on grid architectures (Globus). Nimrod provides a declarative language for describing parametrized experiments. The core part of the architecture is a parameter engine which is responsi-

ble for parameterizing the experiment, creating jobs and mapping tasks to the resources through a schedule advisor. The scheduling approach in Nimrod/G is based on grid economy with deadlines. This tries to achieve trade-offs between performance and cost [49]. The GridSolve system works in a similar fashion, exploiting an agent to maintain details about available servers and then selecting resource on behalf of the user. NetSolve has several specialized execution mechanisms which support common computing models. Another similar application is AppLeS [48] which focuses on the scheduling problem and provides various solutions such as self-scheduled work-queue and adaptive scheduling with heuristics.

BOINC [13] is a task farming, CPU scavenging framework which has been popularized by SETI@home. BOINC architecture is centralized and clients log into servers asking for work. The BOINC system harnesses a wider distributed system through volunteer computing whereby any user on the Internet can take part in the system by donating computation and storage to be used by BOINC. Applications running on the BOINC system are largely independent and hence can scale quite well on such architectures.

Tasks need not just be traditional jobs such as scripts or executables. A data task can also be a web service which means we need techniques for data computing using services. Most of the common workflow system within the scientific community such as Taverna [50], Triana [51], Kepler [52], Pegasus [53], WS-VLAM [44] and GWES [54] focus on orchestrating service-based workflows by contacting the statically located services and marshaling the output of one as the input of the successor. This technique usually involves data being passed through the central coordinator which can easily result in a bottleneck for large web service workflows.

Circulate [55] is a web service choreographic and orchestration system which decentralizes web service choreography through a system of proxies which aid the web services to directly talk to each other without going through a central coordinator. Orchestration is still centralized and is only used to control the overall execution.

DynaSched [56] provides a framework for dynamic WSRF service deployment on Grid resources. A central orchestration engine overlooks the whole workflow execution. A scheduler is responsible for deploying services into WS-containers. The WSRF services communicate with files over GridFTP or RFT servers. With dynamic deployment DynaSched achieves service mobility.

ServiceGlobe [57] only aims at dynamic web service deployment with replication and load balancing. ServiceGlobe differentiates between dynamic and static services. The latter being those services which can not be moved around due to some dependency. The architecture relies on a dispatcher which is described as a software-based layer 7 switch. The dispatcher balances the load on a set of replicated web services and can initiate replicas

on-demand when the load increases. This system can reallocate web services and also scale services using classic metrics such as CPU load.

3.2 Service-based Approach to Farming Workflows

The concept of workflow as a service (WFaaS) has been elaborated to increase the performance and minimize the overheads of workflow farming. In the initial scenario a workflow is submitted to computational resources to process a particular set of data and input parameters; after the processing is finished and the results are collected, the workflow is gracefully terminated. When the next set of data and parameters is to be processed, the workflow is started again which means that all the workflow components have to be re-scheduled, most likely, on a different set of grid resources.

In grid and other shared environments these activities form a significant overhead due to queue waiting times for resource acquisition and staging-in overheads. One way to avoid such overheads is to keep the workflow running on the resources even after a particular data set has been processed. The next data set can be assigned to the workflow that is already instantiated; the parameters for the next execution can also be changed at runtime. This approach helps to save the time of executing the whole workflow each time. Such a concept of user-level preliminary allocation of resources has been employed for user-level scheduling and execution of multitude of short-running jobs on grid resources [58]. Because workflows are kept running, waiting to process new data or parameter sets, they behave as services hence the Workflow-as-a-Service or WFaaS paradigm. In this way, for workflow farming, only a limited set of workflows or workflow tasks have to be executed and kept running on the resources, see Figure 3.1.

Another approach is to reuse computing resources. In many workflows some tasks are short lived and other are long lived. This creates a situation where short lived tasks spend more time stuck in queues than actually spend time executing. A better solution would be to reuse the resource acquired by short lived jobs to run other workflow tasks thus reducing waiting times. Because harnesses (task container) are kept running, waiting to start new tasks, they also behave as services and adds to the WFaaS paradigm.

3.2.1 Data-centric Workflows

Our dataflow model is represented as Directed Acyclic Graphs (DAGs). Vertices in the graphs represent computation as tasks while edges represent data communication and dependency. Each task has a number of typed input and output ports. These ports represent data channels between tasks in the workflow. The links between data channels represent

data dependencies. This allows the enactment engine to model the workflow as a dataflow graph.

We model a workflow W as a set of interdependent tasks $\{t_1, t_2, ..., t_n\}$ which are matched to the set of resources R. Tasks are represented as tuples

$$< id, st, IP, OP, PT, DT, IC, OC >, \tag{3.1}$$

where id is the task id, st is the allocated computing slot time for a given task, IP is the set of input ports, OP is the set of output ports, PT is the set of tasks that precede task t_{id} where $PT \subset W$, DT is the set of dependent tasks that follow task t_{id} where $DT \subset W$, IC is the set of input data channels between output ports of tasks in set PT to input ports for task t_{id}. Similarly, OC is the set of output data channels between output ports for task t_{id} to input ports of tasks in DT. Ports consume and produce a set of messages $\{m_1, m_2, ..., m_n\}$, messages are delivered and produced sequentially. The dataflow model dictates that a task t_k will only be matched to a resource in R when, for each input port IP_{t_k}, the first message m_1 is delivered.

WFaaS paradigm is described at two levels: data-level; and task-level. To better describe the distinction between these levels we extend the definitions introduced in section 3.2.1 by adding a new set, H, which is the set of harnesses (task containers) hosting the set of tasks $\{t_1, t_2, ..., t_n\}$ pertaining to workflow W on the set of resources R. The traditional approach of mapping the set of tasks, W to the resources, R is done directly through a scheduler, $shed : W \rightarrow R$. In task-level WFaaS we use an intermediate mapping such that a scheduler maps the set of harnesses to the resources, $shed\_harness : H \rightarrow R$, and $shed\_tasks : t_{i...j} \mapsto h_n$ maps an arbitrary number of tasks to a single harness instance, h_n. This allows a harness to process multiple tasks thus achieving task-level WFaaS. Furthermore, data-level WFaaS is achieved by invoking tasks with multiple data. In our module the unit of data is a message, thus during the lifetime of a task, the set of processed data equates to the set of messages, M, consumed by the task. Similarly to task-level WFaaS, $shed\_data : m_{i...j} \mapsto t_n$, an arbitrary number of messages can be mapped to a single task thus we do not need to create a separate task for every data message.

3.2.2 Task Farming with Data Partitioning

A common pattern in scientific applications for achieving higher throughput is using a master/slave model and partition the data amongst the identical slave tasks [59]. In such a model, a master coordinator is responsible for disseminating data chunks to all slaves. This approach does not usually consider auto-scaling the amount of slaves and most often relies on over provisioning resources by greedily initiating as many salves as possible. Such an approach is not well suited for scientific workflows since each task can possibly hog all the

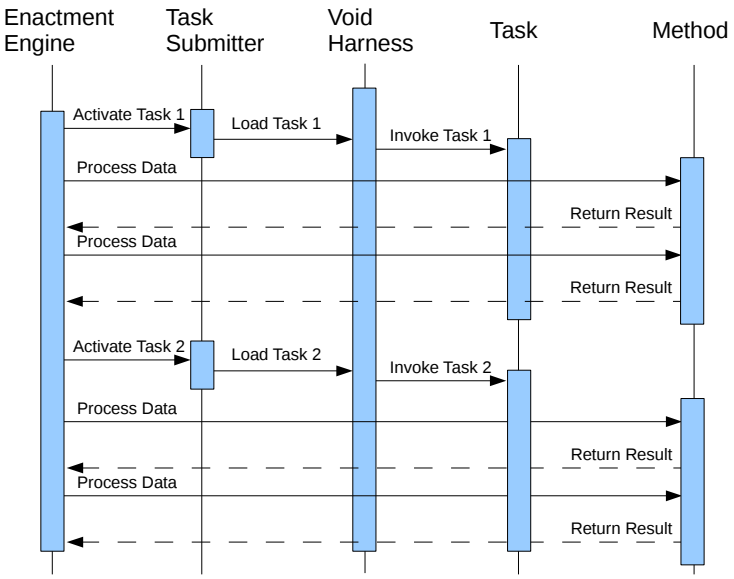


Figure 3.1: Sequence of events following the WFaaS paradigm. The enactment engine activates a task which in turn is passed onto the submitter. Having already a harness running on some resource, the submitter invokes the harness to load the task. Once loaded, the methods within the task can start processing data. The task methods can be invoked over multiple data chunks thus the WFaaS paradigm; furthermore, the void harness can be re-invoked with a new task thus acting as a service at the task-level also.

available resources by initiating too many slaves and hence starve the rest of the workflow which will impede its progression.

Many simulation-based scientific workflows involve setting appropriate initial parameters which are often determined from a large number of smaller scale test runs and automatically staged to appropriate resources [60]. The task-farming paradigm is a technique well suited to distributed architectures such as grid or clouds where multiple heterogeneous resources can be used for the concurrent task execution.

Task farming is applicable to embarrassingly parallel applications where non communicating tasks can be executed efficiently across geographically distributed computing resources. The basic operations are: spawning tasks, finding appropriate computing and storage resources and assigning work to each task. One approach to task farming is the master-worker paradigm where a master process spawns multiple worker tasks or processes; once these initial tasks are executed, the master process creates the next tasks until the task-bundle is finished. A number of potential applications follow this paradigm, among them Parameter Sweep Applications (PSAs) [60] in which a potentially huge parameter space is divided into regions and worker tasks search the specific region for optimal values. A subclass of such applications is data independent parameter sweeps where the input dataset is shared amongst workers.

Farming tasks deals with splitting data or parameter space n-wise amongst n identical tasks. This technique speeds up data processing especially when dealing with independent tasks. Farming can make better use of the resource by elastically replicating tasks to reduce empty resource slots while reducing the workflow makespan. If we consider data D to take time T_1 to process on one node and T_n on n nodes when dividing D amongst the n nodes, the ideal speedup is n and is defined as $\frac{T_1}{T_n}$. A close to ideal speedup can be achieved when assuming independent tasks with negligible overhead.

Farming allows the execution of large number of experiments each of which may have a different input data set or a different parameter set. A number of the preparation steps involved in these experiments can be automated: staging in and out the appropriate input and result datasets, finding available and appropriate computing and storage resources etc. Another important aspect is the reproducibility of results. When a scientist has performed thousands of simulations and a few of them fail, he is interested to know which ones failed and the context in which they have been executed (computing and storage resources, libraries, input data sets, parameter values etc). This requires appropriate monitoring and provenance systems.

Parameter sweeps are a special kind of task farming with the difference that the data being split amongst the task pool is the set of parameters. PSAs are characterized by an embarrassingly parallel application which is an application that can be decomposed into many independent tasks with little or no synchronization or data dependencies. Parameter

sweep model is a simple yet powerful concept used by many scientific application such as those in: computational fluid dynamics, bio-informatics, particle physics, discrete event simulation and computer graphics [49]. In a PSA, data is replicated to all collaborating tasks while each task is given a set of different parameters where each task in a PSA works on identical data. Since PSAs are intrinsically independent they can tolerate network latency and therefore scale to large distributed architecture. Additionally, they are amenable to simple fault tolerance mechanisms such as retries [49].

Having a single workflow instance processing all the parameter space might not always be optimal as shown in Figure 3.4 especially when resources are under utilized thus the workflow management system has to leverage the number of workflows for better resource utilization by taking into consideration the size of parameter space and the available resources. This is done through two main scaling strategies: fixed-scaling, and auto-scaling. In fixed-scaling the WS-VLAM initiates a fixed number of workflow instances whilst with auto-scaling, WS-VLAM automatically replicates the workflow by looking at the parameter space and the time taken for each parameter to be processed. Auto-scaling is based on estimating the predicted task execution time as is described in Section 3.3.

In chapter 6 we demonstrate the WFaaS approach by comparing the resource usage footprint using two identical workflows: one using the WFaaS approach and the other without WFaaS.

3.3 Predication-based Scaling Dataflows

The dataflow model described in Section 3.2.1 allows us to implement auto scaling routines by reasoning about data messages being passed around tasks.

A prediction engine for each task can infer the data load on a task by monitoring the queued data chunks for a given task and computing the estimated execution time using heuristics from previous data processing times. Having a high load on a task, a coordinator can decide to replicate the task and partition the data chunks amongst them thus reducing the apparent load since data is consumed at a higher rate. Within scientific workflows, where each task is subject to this prediction engine, each task can independently scale to accelerate its consumption rate and match its predecessor's data production rate thus maintaining a steady flow of data through the workflow.

Auto-scaling is achieved by monitoring messages between tasks, the auto-scaling component can deduce which tasks are overloaded by predicting the completion time. Given a task t_k, replication can be applied by monitoring a designated port $ip_{t_k} \in IP_{t_k}$. The first step in auto-scaling is to keep track of the data processing rate of t_k on a node $r_j \in R$. $M_{ip.t_k}$ is the set of messages for the designated port ip_{t_k}. $C_{ip.t_k} \subset M_{ip.t_k}$ is the set of messages already consumed by ip_{t_k} where the current message being processed is also

part of this set. Similarly $Q_{ip.t_k} \subset M_{ip.t_k}$ is the set of messages yet to be consumed by ip_{t_k}. The function $time_{ip.t_k}(m_j)$ records the time a message has been delivered to input port ip_{t_k}. The processing time of a message is defined as the interval time recorded between successive messages thus $mit_{ip.t_k}(m_{l-1}, m_l) = time_{ip.t_k}(m_l) - time_{ip.t_k}(m_{l-1})$. $size(m_k)$ represents the data size of m_k thus the actual data size of the set M, $size(M) = \sum_{j=1}^{n} size(m_j)$. Since messages can have different data sizes, auto-scaling component needs to calculate the data processing rate. This is done on every message being consumed by a port and is defined as:

$$proc(ip_{t_k}) = \frac{size(C_{ip.t_k})}{\sum_{j=2}^{n} mit_{ip.t_k}(m_{j-1}, m_j)},\ n = |C_{ip.t_k}|. \tag{3.2}$$

Equation 3.2 calculates the data processing rate in bytes per second and this includes also the overhead of delivering a message, the overhead to start processing the next message, and depending on the implementation of the task it would typically include the time for producing messages on output ports as a response to processing input messages. Equation 3.3[1] calculates the expected time for completion

$$pred(ip_{t_k}) = \frac{size(Q_{ip.t_k})}{proc(ip_{t_k})}. \tag{3.3}$$

The prediction is calculated on every consumed message and is averaged out with the last calculated prediction to smooth out large spikes in the prediction graph. Task t_k is said to be overloaded if for the designated port ip_{t_k}, $pred(ip_{t_k}) > t_k^{st}$ alternatively, t_k^{st} could be substituted by a user-defined threshold. A simple calculation of the needed clones to reduce the completion time below t^{st} is

$$repl(t_k) = \frac{pred(ip_{t_k})}{t_k^{st}}. \tag{3.4}$$

Equation 3.4 assumes that all messages have approximately the same size which may not always be the case thus another solution is to use a replication coefficient that influences the replication count. One way to calculate the coefficient[2] is to use the standard deviation of the normalized message sizes (between 0 and 1) such that

$$corr(Q_{ip.t_k}) = 1 - stdd(||Q_{ip.t_k}||). \tag{3.5}$$

The new number of clone to be initiated can then be calculated as $repl(t_k) \times corr(Q_{ip.t_k})$. The greater the message size standard deviation the less clones are initiated. Large standard deviation results in messages having drastic variation in their data sizes and can lead to over-provisioning resources through inaccurate clone number calculation.

[1] equation is a corrected version presented in publication [41]
[2] modified from the original paper [41]

Another source of variation in the calculation is the fact that the prediction is based on some resource with its own characteristics CPU power and memory capacity. Since resources in distributed systems are intrinsically heterogeneous, the time to process messages on one resources might not be the same as on other resources. This may lead to over-provisioning if the first estimation is done on a relatively slow resource while the clones are scheduled on faster resources and vice-versa leads to under-provisioning.

To cater for over-provisioning, clones are scheduled in bursts. When a burst of clones is scheduled, auto-scaling continues predicting the estimated completion time which would now include the data processing of all instances of the replicated task. If the task is still overloaded more bursts are scheduled until the load is reduced within the acceptable limit.

3.3.1 Dataflow Architecture

Figure 3.2 depicts a high-level overview of the Dataflow architecture. The system is composed of a set of loosely coupled components bound by a central messaging back end. The Dataflow architecture implements a two-step scheduling system. The enactment engine represents the top-level scheduler which models the workflow task data dependencies. The enactment engine deals with tasks at an abstract level and merely marks tasks as runnable when their dependencies are met. The bottom-level scheduler deals with scheduling tasks on a set of resources thus its main role is matchmaking. The central component in the whole architecture is the messaging back end which binds all the components together.

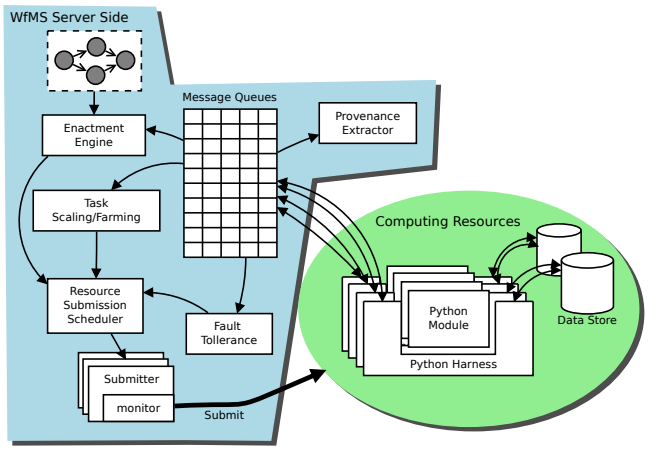


Figure 3.2: Loosely coupled Dataflow architecture components revolving around a core messaging component.

The Dataflow enactment engine is the entry point into the system. It accepts a WS-VLAM [61] DAG workflow. At this stage the workflow is interpreted and a dataflow object

representation is generated. When a task is made runnable (i.e. all input ports have data) it is passed to the bottom-level scheduler for matching to a resource. The architecture allows for different schedulers to be implemented. Default schedulers are *round-robin* scheduler which circularly matches tasks to resources and therefore achieves load balancing between resources, *bucket* scheduler which orders resources by the amount of slots available and fills up the resources consecutively starting from the largest resource thus achieving locality between tasks, *cloud* scheduler which takes into consideration a budget for running a workflow and elastically expands the resource pool R by calling an interface to cloud resources [62] for on-demand cluster creation to accommodate more tasks.

3.3.2 Dataflow Data Queueing

The message queues play a pivotal role in the whole architecture. Most importantly message queues allows inter-task communication over inter-cluster domains which, most often, have restricted internet access. Intermediate messages allow tasks to exploit fine concurrency between dependent tasks. As with streams the granularity of concurrency depends on the task logic and how often messages are produced and consumed. A one-to-one mapping exists between the set of task input/output ports (IP, OP) and message queues. The message queues provide a persistent means of communication between tasks which in turn decouples task execution in time and hence eliminates the need to co-allocate resources. Co-allocations is known to degrade the system due to increased task wait times [63, 64]. As depicted in figure 3.2, the message queuing also allows for components such as the message router and the Dataflow engine to spoof on the messages being transmitted. Based on the message routing, the Dataflow engine knows which tasks have data on their ports and thus can make tasks runnable.

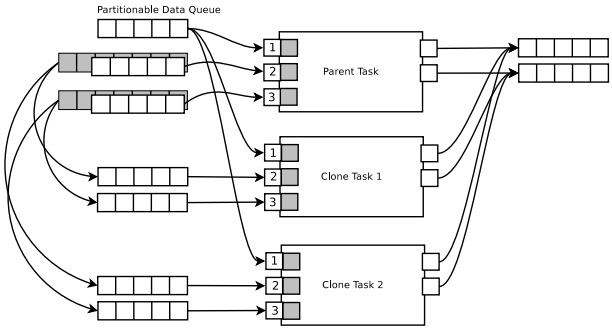


Figure 3.3: Queue setup strategy for achieving auto-scaling. Parent non partition-able data queues have associated shadow queues which enable clones to retrieve the whole input message set any time. Input ports also have a reserve port which is used by the fault tolerance subsystem to replay the last message in case the task is re-submitted.

Figure 3.3 depicts the queuing strategies that supports cloning tasks. Cloning is the procedure of replicating a task by the scaling subsystem. The parent task is responsible for managing its own clone farm which means that the Dataflow engine has no knowledge of replication taking place. This preserves the original workflow semantics. Port 1 of the parent task is the designated port for which auto-scaling will perform prediction and replication. All instances of the same task will share the designated port and thus the data is partitioned amongst all clones. Ports 2 and 3 are not partitioned amongst clones. This is due to the complexity and ambiguity of how to partition multiple ports amongst a set of clones. The system guarantees that for any non-designated input ports, all clones have the same input message set therefore in figure 3.3 the input message sets M_{ip_2} and M_{ip_3} are identical to the parent and clones. This is achieved through shadow queues on the central message exchange. A shadow queue acts as a buffer for the set of messages consumed by the parent. Clone input ports are attached to the shadow queues instead of the standard queues. All clones share the same output ports with the parent. By default no ordering is done on the message output queues hence messages are delivered out of order. Ordering is an expensive routine and can be achieved through the message sequence numbers.

Each task in the workflow polls a message server for new inputs, be it parameters or data. This polling mechanism circumvents common network restriction on computing nodes which tend to block listening ports. With many polling tasks, a server can easily be inundated with polling requests which is a common problem known as a thundering herd problem. This occurs when many tasks decide to poll the server at the same time. To limit this problem with implemented an exponential back-off where tasks exponentially increase their own polling interval when no new messages are retrieved. Each port on a task is associated to a message queue on the message server. Task communication is achieved by routing output messages from one task to the input message queues of the next. Tasks act as services by continuously consuming new parameters and terminate once all input queues are exhausted of data or the allotted time on the computing node has expired. Replicas of the same task are attached to the same queues hence data is automatically partitioned amongst instances of the same task. This system of message queues depicted in Figure 3.5 allows tasks to scale so that each task processes a subset of the parameter space instead of just processing one parameter and immediately terminate. Data scheduling is also done through the message queues attached to the ports on the messaging system. Data is partitioned through shared queues where multiple tasks access the same queue and in turn retrieve data messages. Data that is not meant for partitioning is fanned out to all replicated task input ports thus each task has a copy or reference to the same data.

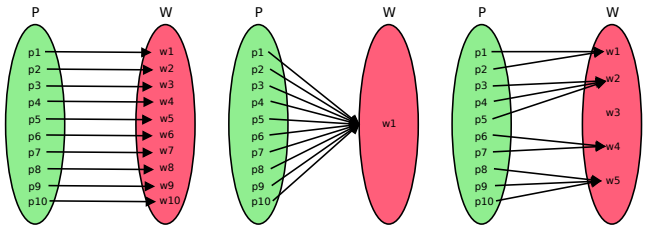


Figure 3.4: Parameter to task mapping. Left: traditional mapping where each parameter is mapped to a single task. Center: the whole parameter space is mapped to a single task instance. Here the task is working as a service but this solution may not be optimal for large parameter spaces. Right: Subsets of the parameter space are assigned to replicas of the different task. Each task is working as a service processing part of the parameter space.

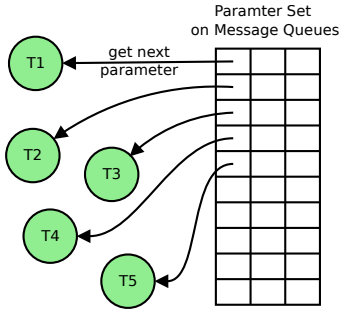


Figure 3.5: Tasks participating in a parametric sweep study. Each replica task reads parameters as data from their input ports which are bound to message queues. Since the input message queue is shared between all replicas, parameters and data are automatically partitioned amongst a farm of tasks.

3.3.3 Dataflow Task Harnessing

The unit of submission is a task harness. The main goals of the task harness are two fold; it allows task late binding by dynamically plugging tasks, and abstracts the underlying data management and communication from the core scientific logic. The task harness is responsible for retrieving messages from the queues, interpret the protocol used in the reference, load the necessary communication libraries, retrieve the actual data from reliable storages, and push the data up to the task. On data output, the harness locates the closest data store from a list of stores, puts the data on the server and queues a message indicating the reference to the stored data.

The task harness, Figure 3.6, architecture is based on a plug-in model whereby the task and communication are dynamic loadable modules into a harness. The core of the harness is the data management fabric which binds loadable communication libraries to

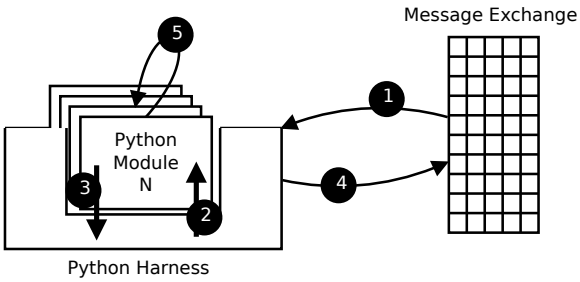


Figure 3.6: A task harness is submitted to a computing node. The actual task is dynamically loaded into the harness. The harness handles the runtime execution of the task by keeping the task alive an realizing the WFaaS paradigm. In (1) the harness gets data from the message queue which is bound to the input port of the task. (2) the data is moved to the task. At this stage the task performs the necessary processing on the data and writes the output to one of its output ports (3). The harness then gets the data and sends it to message exchange (4) to be consumed by other tasks in the workflow. Steps (1) to (4) repeat themselves until all data on the message queue is consumed. After terminating the task, the harness does not quit the resource but loads the next task in the job queue (5) and repeats steps (1) to (4) again.

the task input/output ports through a system of queues. These internal queues decouple the scientific logic from the underlying communication mechanisms. On starting the harness, the configuration is loaded which allows task late binding since it is only at this point that a task is assigned to a harness. On loading a task t_k, the harness sets up internal data queues for each task port in $IP_{tk} \cup OP_{t_k}$. Messages containing referenced data are handled by the harness by dynamically loading the appropriate protocol library, such as GridFtp, for retrieving the actual data. The communication library responds by pushing the data onto its internal queue. The harness will then route the data from the communication queue to the relevant task input queue. Data output by the task happens in reverse order. When data is available on the tasks' internal output queue, the harness picks up the data, it then locates the closest data server from a list of servers, loads the required communication libraries and pushes the data to communication queue. The communication module picks up the data and sends it to the data server. The harness will then construct a message with the endpoint reference of the newly created data and sends it to the central message queue which is then routed to other tasks in the workflow.

Since communication in distributed environments can be quite restrictive due to security policies, the architecture relies on a pull model whereby tasks initiate all communication. A pull model is the best guarantee that tasks can establish communication. The task harnesses poll the server for new messages. Polling implements an exponential back-off when no new

messages are retrieved. When no messages are retrieved the polling interval is increased up until a fixed threshold or until a new message is retrieved. This reduces the load on the messaging back end by not overwhelming the server with too many unnecessary requests.

A simple task module is listed in Listing 3.1. The harness is responsible for executing the scientific task and abstract the task communication. Each input and output port on a task is bound to a message queue on the exchange server. The harness is responsible of subscribing to such queues and retrieve data. This data is then moved to the relevant ports on the task. Large data is not passed through the message exchange as this would overload the server. Therefore data is referenced while the actual data is resident on some dedicated storage such as Webdav, GridFTP, etc. The harness is also responsible for downloading the actual data from the reference in the message. Large output data is automatically copied to a dedicated storage and the reference to it sent to message server. The output data server can be dynamically chosen from a set of configured servers by choosing the closest one. Listing 3.1 shows a typical workflow module; Line 1 names the task/module. This is the name given to the task in the workflow. Line 2,6,15 define three function which are implemented by the scientific programmer. Upon loading the module, the harness calls *on_load()* where initialization can take place. After *on_load()*, the harness calls *run()* on a separate thread. *run()* is the main routine where most scientific logic takes place. The harness offers easy ways how to read data, line 10 reads a data chunk from the tasks input port. After processing the data, output results can be simply written to the output port in line 13. The harness also allows for in-application provenance gathering whereby the programmer can write events, line 12, which are collected by the system. When no more data is available on the input ports, *on_unload()* is called which is intended to run cleanup routines. The harness will then proceed to unload the module and load a new one. The lower example in Listing 3.1 shows the same module implemented using callback functions to process data. Line 22 registers the callback function for the input port. For every data chunk (parameter *body* in Line 28) the callback *data_processor()* at Line 28 is called to process the data.

In chapter 6 we demonstrate an streaming image processing workflow and how the prediction-based scaling autonomously scale parts of the workflow independently from the other tasks in an effort to limit workflow bottlenecks.

3.4 Fuzzy-based Scaling Web Services

The same dataflow model described in Section 3.2.1 can be applied to various types of tasks. In this section we describe how the approach was applied to traditional web services while at the same time extending the auto scaling to include the state of the resources and thus dynamically scale the data processing dynamically depending on the state of the


```
0 import plugin
1 class TestModule(plugin.PluginBase):
2     def on_load(self):
3         #this is called when the module is
4         #loaded and before run()
5         pass
6     def run(self):
7         #this is the mian function for performing
8         #scientific logic.
9         while true:
10            data = self._read_from_port(self._get_port("input_port"))
11            #process data
12            self._write_event("provenance data")
13            self._write_to_port(self._get_port("output_port"), data)
14        pass
15    def on_unload(self):
16        #this is called when no more data is available
17        #for processing
18        pass
```

```
19 import plugin
20 class TestModule(plugin.PluginBase):
21     def on_load(self):
22         self._register_callback(self.data_processor, self._get_port("
               input_port"))
23         pass
24     def run(self):
25         pass
26     def on_unload(self):
27         pass
28     def data_processor(self, body):
29         data = body
30         #process data
31         self._write_event("provenance data")
32         self._write_to_port(self._get_port("output_port"), data)
33         pass
```

Listing 3.1: A task module where scientific logic is implemented. This module gets loaded dynamically into a harness which in turn is running on a resource. The harness abstracts communication and simplifies scientific logic implementation. (Upper) an implementation of the module using explicit reads from port. (Lower) the same module implemented using callback functions instead of port reads.

resources. In doing so, we also present new methods making web services more tailored for data processing.

eScience applications are becoming increasingly data-centric and service oriented. Web services support interoperable machine-to-machine interaction [65] and give rise to the Service Oriented Architecture (SOA) paradigm. The nature of many web service based eScience application such as those from bio-informatics rely on statically located web services. The static characteristic of such services makes it difficult to choreograph a set of cooperating web services in such a way that they can be optimized to meet the demands of the increasing scientific data. On the other hand distributed resource middlewares such as the Grid are not well equipped to host such services in a dynamic way. Common middlewares expose low level interfaces for accessing the resources. This usually results in scientists writing custom software frameworks for accessing the distributed resources. The consequence of middlewares lacking mechanisms for provisioning scientific web services means that a huge body of scientific logic is trapped within static web services and have no means to exploit distributed resources. The combination of multiple services or tasks that should act on data leads to workflows of execution whereby the structure of the workflow defines some interdependency such as data or control.

Web services are passive program objects which are hosted in service containers such as Apache Axis2 [66]. Containers are responsible for managing the service life-cycle including starting, stopping and invoking methods. WSDL is a descriptive language that abstractly describes the operations a web service exposes without any knowledge of the implementation. Web services are referenced through an End Point Reference (EPR) which is a location-based addressing scheme using URLs. Common web service method invoking is through SOAP over HTTP. SOAP is an XML-based protocol that describes which method to invoke and the list of parameters for the particular method. Common service containers listen for incoming SOAP messages over HTTP and after the method has been called it returns a SOAP response to the client.

The above exposes the first two challenges for realizing our architecture. (1) The EPR system of addressing a web service binds a service to a location. This hinders the web service mobility as clients have no easy way locate a service that is roaming about at different locations thus mobile service have to be addressed in a location-agnostic manner. (2) The passive mode of communication means that web services residing behind firewalls, as is the case with the majority of distributed shared resources, have no way of being access from outside the network as inbound connections are usually blocked.

Both these challenges are solved using the same basic idea of message queueing. For the communication problem, message queues allow containers to poll and pull SOAP messages off a queue which itself is accessible outside the network. By systematically setting up different queues for each deployable web service, the queue id becomes the service

EPR. This de-localizes web services and allows them the migrate to different resources without clients being aware of it. In light of web service mobility we distinguish between two types of services; *fixed* services which can not be re-allocated due to some dependency such as accessing a local file systems, and *pure* web services that are self-encapsulated. The architecture targets the latter.

The pull communication model and the location-agnostic service addressing are the basic foundation for our architecture. With these two characteristics, services can be dynamically deployed anywhere on the Internet having at least outbound communication capabilities. Based on the same notion of message queueing, the system is further capable of achieving web service back-to-back communication and elastic scaling.

The architecture depicted in Figure 3.7 revolves around a message brokering system (similar to the architecture in Figure 3.2) which loosely couples all other sub components. The message queue system exposes two types of queues, those intended for web service communication and other queues for orchestration coordination. Coordination queues include; a global run-queue where services awaiting execution are queued, an events queue for gathering events from running services, connection queues for describing the workflow topology, and a command queue for every service where commands such as `kill` can be sent.

Figure 3.7 illustrates the steps in which a web service based workflow is orchestrated. In **1**, a workflow is bootstrapped. In bootstrapping, the first web service is put on the run queue and the web service connections are also made available on the messaging system. These connections allows web services to autonomously know to whom they are connected which in turn allows back-to-back communication. In **2** the service submission picks up the bootstrapped service and submits it to one of the configured resource submitters **3**. Submitters abstract the actual resources and are responsible for monitoring the available free slots on the resources.

In **4** a submitted service container lands on a worker node. The service container is initially void of a web service to host. The latter is referred to as late binding which binds a service to a resource only if the container successfully loads. The first step for the container is to check the message queue for any available web service to host. If so it will load it from the service library **5**. Upon deploying a web service, the container register itself as a consumer for the service input and command queues. The container then starts consuming SOAP messages from the designated queues on the message system and pushes them up to the web service. At this point two threads of logic are being executed on the worker node; that of the web service itself ant the other of the container which apart of pulling and pushing SOAP messages is also responsible for orchestrating its neighboring workflow services and handle its own scaling routines.

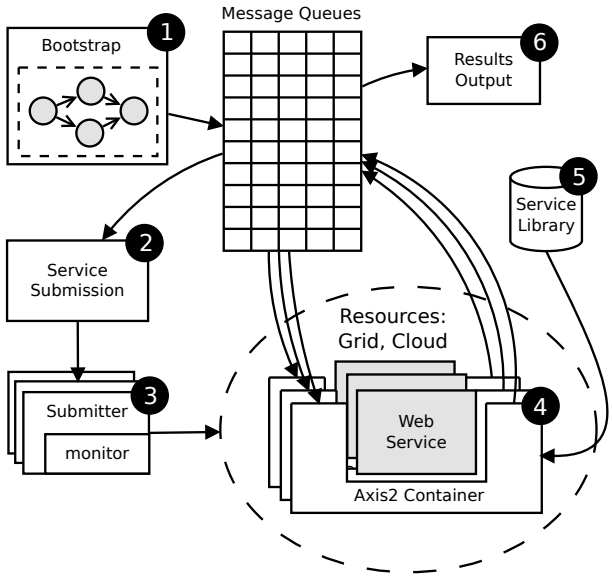


Figure 3.7: Loosely coupled core components revolving around the message broker.

The hosting container learns about its hosted web service neighborhood workflow topology through the web service connections queue. This queue is created and populated during the bootstrapping stage. With the knowledge of its successors, the container transforms outgoing messages directly to input for the next web services and pushes the messages directly onto the input queue of the next services. This allows web service back-to-back communication since communication is done directly through queues without any other central entity marshaling outputs to inputs.

Web service orchestration is modeled on a dataflow approach. This model dictates that only those web services having input data to consume may become active. The advantages of such a model is that resources are not waisted by idling services. Furthermore, combining dataflow models with messaging back-ends, services are said to be decoupled in time. With time decoupling, cooperating web services need not be active simultaneously. This reduces the need for co-allocating resource which have been shown to degrade a system due to increased run-queue waiting times [63, 64].

Due to the dataflow model, services will not be active at the time their predecessor outputs the first data messages. The predecessor service container is responsible for orchestrating the next services in line. This is done at the same time messages are being transformed from outputs to inputs. The predecessor service container queries the input message queues for it successors to deduce whether any instances are active. If not the predecessor will orchestrate the services by pushing an instance of the successor on the global run-queue. The queued instance is then picked up by service submission in **3**.

If the running web service is a terminal service the output is written to the standard service output queue. A client in **6** will then read the workflow results from the last web service/s output queues.

The central messaging system is an Apache ActiveMQ [67]. ActiveMQ is an enterprise messaging system with many features that can be used to tune the performance of the architecture. Noticeable features include; fail-over setup where web services can connect to different brokers in the event that one fails, and networks of brokers where messages travel from one broker to the next until it reaches a consumer. This would facilitate hierarchical web service orchestration where different groups of services are attached to different brokers and communication between services is done through the network of brokers.

The web service container used in this architecture is the Apache Axis2 [66]. Axis2 is a light weight extensible container perfect for submitting container-level jobs. The architecture relies heavily on the modifications done to the default Axis2 container. Most of the architecture is implemented within the container. Axis2 container offers an easy way how to extend its functionality through hooks in the system.

The web service library is a simple HTTP server where web service bundles are kept. For the rest of the architecture including bootstrapping, submission, and results client, Java was used as the programming language of choice although ActiveMQ has numerous APIs for different languages.

3.4.1 Web Service Container Architecture

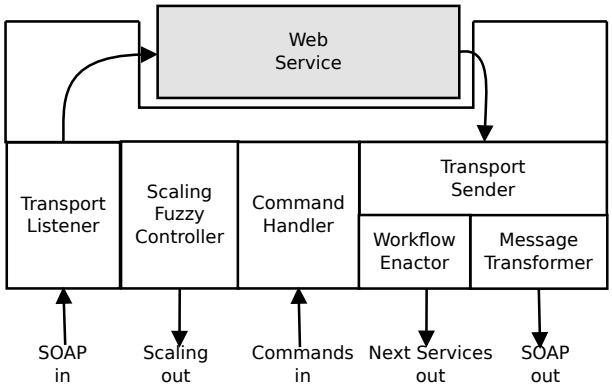


Figure 3.8: Modified Axis2 container including transport handlers for pulling messages, autonomous workflow enactor, fuzzy controlled scaling, message transformer, and a command handler.

Most of the management routines reside in the Axis2 container which executes alongside the web service on worker nodes. The modified Axis2 container transforms a tra-

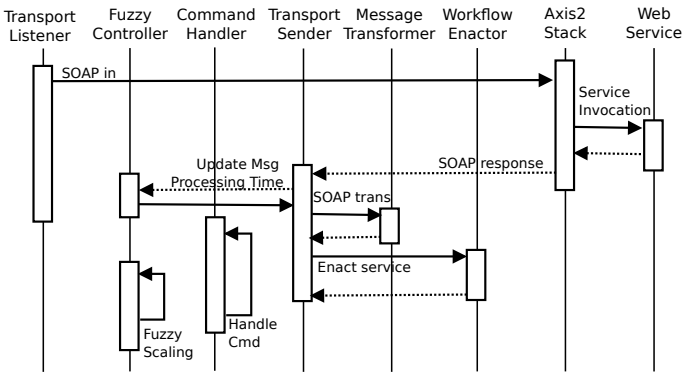


Figure 3.9: Sequence for message reception and delivery to the web service by the modified Axis2 container. Although the system uses SOAP in this scenario, the same method can be used for RESTful services.

ditional web service into a mobile object with smart orchestration and scaling routines. Figure 3.8 highlights the main components added to a standard Axis2 container. The customized transport handlers are the main entry and exit points for the web service. The workflow enactor component implements the autonomous orchestration (Section 3.4.3). The message transformer implements back-to-back communication (Section 3.4.2) and the fuzzy controller implements autonomous scaling (Section 3.4.4). The command handler consumes command messages from the service command queue and acts upon them.

Figure 3.9 illustrates the round trip path of events for a SOAP message through the container. On reception of a SOAP message the transport listener processes the message such as adding time stamp information and moves it up the to the Axis2 stack. The container unmarshals the SOAP message and invokes the web service method with the parameters extracted from the message. The container stack returns the SOAP response to the transport sender. The latter updates the message round trip time. Message round trip times are used by the service replication routine to deduce the load on the service. The message transformer transforms response messages to input SOAP messages for successor services. The workflow enactor checks if any successor services need to be initiated. Finally, the transport sender sends out the transformed messages or the default response message if no transformation took place to the designated queues. The fuzzy controller and the command handler execute loops on separate threads. The former elastically scales the service instances while the latter listens for commands.

3.4.2 Web Service Back-to-back Communication

In cooperating web services such as those in pipelines or workflows it is often advantageous to allow web services to directly talk to each other without the need for a client to coordinate the communication. This is especially the case for complex workflows where it is not feasible to manage all the inter-service communication. For this reason the modified Axis2 container allows web services to directly talk to each other through the message broker.

At the bootstrapping stage, the topology of the workflow is known. The topology is synthesized into messages on dedicated connection queues thus, in the pipeline topology depicted in figure 3.10, there exists a connection between A.method1() and B.method1(). The connection would translate into a message on A.method1.connections queue. This designated queue is used by service A to deduce to whom it is connected hence giving A the knowledge of its neighbors. The messages on the connections queue describe the SOAP template expected by the successor (in this case B.method1()).

When A.method1() returns a SOAP message it is picked up by the message transformer inside the Axis2 container (see Figure 3.8). The SOAP template present on the connections queue is used to transform the response message from A.method1() to the input of B.method1(). This transformed message is then written directly to B.method1.input queue by the transport sender for A.method1().

Since B.method1() has no successor connections, any output by this method is written to the method's default output queue B.method1.output which can then be consumed by a client waiting for output from the pipeline.

In the scenario of a fan-in topology, multiple services connected to B, write their messages to the same input queue for B. Similarly in a fan-out approach where A is connected to multiple services, the message transformer transforms the SOAP response for each successor. In both cases message ordering is not guaranteed but can be accomplished through message sequence ids on the container though this is very expensive operation and can lead to memory exhaustion due to buffering messages in order to re-sequence them.

3.4.3 Web Service Autonomous Orchestration

The connections queues described for back-to-back communication are also used to enable autonomous orchestration. Connections between services represent a data dependency thus from figure 3.10 service B is data dependant on service A. The dataflow model approach dictates that service B should only become active when it has data to process. This model is enforced autonomously by the individual containers.

From figure 3.10 A.method1() produces data for B.method1(). This satisfies the dataflow model that B.method1() should become active since data is now available. The workflow

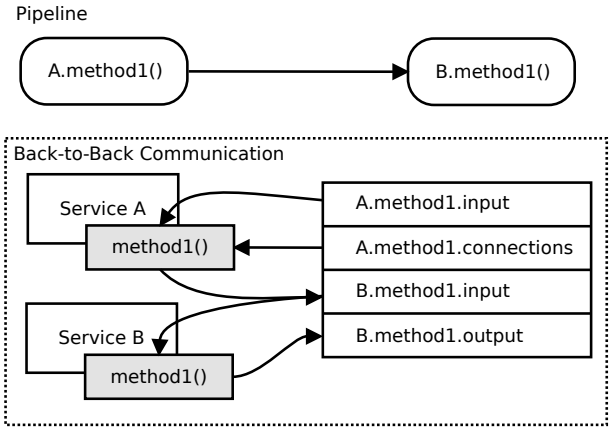


Figure 3.10: Back-to-back communication for a two service pipeline. Service **A** knows about the connection between A.method1() and B.method1() through the connections queue. Any output from A.method1() is sent directly to B.method1().

enactor component in the container for service A can deduce if any instance of B is running. This is done by checking the number of consumers on B.method1.input. If no consumer is active on the queue, the workflow enactor submits and instance of B to the global run-queue. The instance is picked up by the service submission (figure 3.7) and submitter to a resource. In the case that service A has multiple successors the procedure is repeated for every successor.

This approach differs from the common scenario of having a central SWMS which has to orchestrate the whole workflow. Typical SWMS are far-sighted i.e. they have knowledge of the whole workflow and hence have to maintain the whole workflow which could be a limitation for large complex workflows. With autonomous orchestration, services are myopic as they only have knowledge of their immediate successors thus no central entity is coordinating the whole workflow execution.

3.4.4 Web Service Fuzzy Controlled Elastic Scaling

A characteristic of many e-Science applications is that they are embarrassingly parallel and therefore can be easily scaled up with simple data partitioning techniques. The main goal of partitioning an embarrassingly parallel application is to achieve better throughput and hence reduce the makespan. This is usually done in a greedy manner where the application consumes as many resources as possible to reduce overall execution time. This premise is not always an ideal solution when dealing with cooperating tasks since one's greed to consume as many resources as possible will result in starvation for other tasks in the workflow. Starving tasks can degrade the whole system since progress is hampered and

data gets piled up waiting to be processed. For this reason we propose a fuzzy controlled elastic scaling mechanism for individual services taking part in a workflow. The fuzzy controllers can autonomously scale up and down the service depending on the predicted service load and the resource load.

Through the messaging system web services can be replicated as many times as needed. Every instance of the same web service is attached to the same queues. From figure 3.10, if multiple instances of service A are initiated then all instances read data from A.method1.input thus the input data is said to be partitioned amongst all instances of the same service. This implements data parallelism. The assumption here is that there is no casual dependency between data messages on the input queue as this would impede data partitioning. Similarly all instances of the same service write data to the same output queues. Services where data parallelism is not possible such as services that need the whole data set to accomplish their task can not exploit such replication and would have their fuzzy controller disabled.

Within a single workflow, cooperating services are competing for resources. This is especially evident when the resource pool is pseudo finite as would be the case in many distributed shared resources. Thus to achieve adequate workflow progress, services must not replicate themselves greedily when not enough resources are available. Conversely, service scaling must take an abstemious approach to resource consumption so as to guarantee whole workflow progress. Such an approach is implemented by means of a fuzzy controller whereby each Axis2 container runs a fuzzy controller for each hosted service to scale up or down the replicated instances of the same service. The bases of the controller is that a web service should be able to aggressively replicate itself when its load is high and resources are free but scale down when its load diminishes and the resource are quite occupied. The latter is intended to make space for other services to take hold of the resources. The decision of *when a task is overloaded* or *enough resource are available* is difficult to simplify using a simple thresholds since service load and resource load are very dynamic especially when cooperating service are influencing each others view of the load. For this reason, calculating the scaling factor of a service such that it does not overuse the resources but at the same time does not under utilize them is a problem well suited for fuzzy logic. In fuzzy logic, terms like *high load* do not represent a single threshold but a range of thresholds with varying membership probabilities.

Figure 3.11 illustrates the inputs (taskLoad, resourceLoad) and output (replication) for the fuzzy controller. The taskLoad input defines a set of fuzzy membership function for the terms *very_low, low, ideal, high,* and *very_high*. Similarly the same terms are defined for the resourceLoad. The output from the fuzzy controller is the scaling count which ranges from -15 to 15 so if the output is -10 then the number of instances for a particular service should be scaled down by 10. These adjustment are done at timed intervals hence the

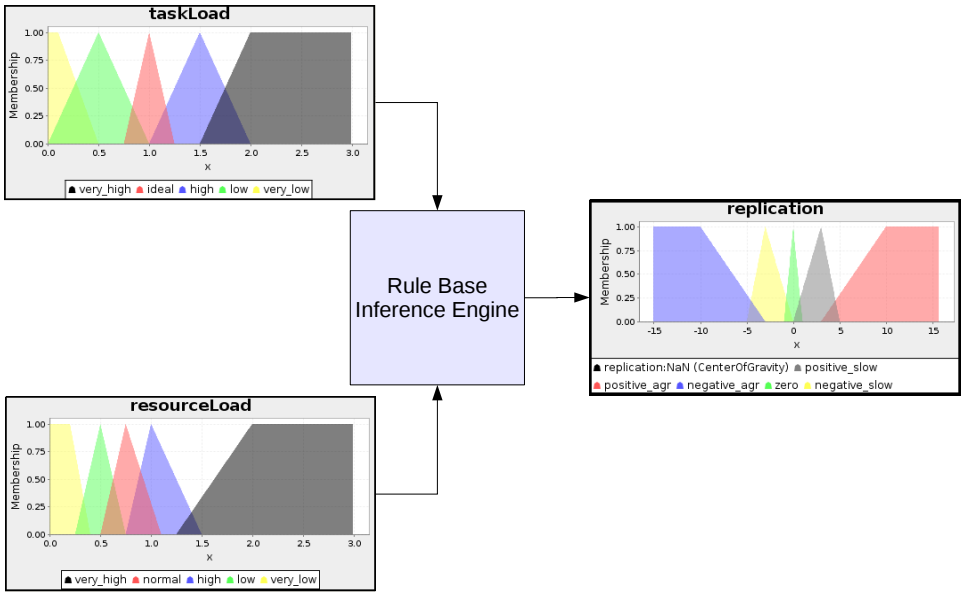


Figure 3.11: Fuzzy controller triangular and trapezoidal membership functions for inputs (resourceLoad and taskLoad) and output (replication). The rule engine implements the rules listed in Listing 3.2.

controlling is progressive. The fuzzy output defines membership functions for controlling how aggressive scaling should be done hence terms like *positive_aggressive, positive_slow, negative_aggressive,* and *negative_slow* are defined.

The taskLoad defines the web service load and is a prediction-based load calculation. Given that at any point in time we know input queue size and the average message processing time, we try to predict the total processing time for the whole input data queue. For every message that leaves the container, the average message processing time is update. A message processing time T_i is defined as the round trip time from when the message enters the container up till it leaves the container hence $T_i = (t_i^{out} - t_i^{in})$. The mean message processing T_i^{avg} is defined as the weighted mean of the current and last message processing time hence $T_i^{avg} = (T_i^{current} w_k + T_{i-1}^{avg} w_p)$ where $i > 0$, $T_1^{avg} = 0$.and $w_k + w_p = 1$. The weights w_k and w_p are always set to favor the highest load thus if $T_i^{current} > T_{i-1}^{avg}$ then $T_i^{current}$ has a higher weighting and vice-versa. This smooths out *flip-flop* scenarios when the message processing time continuously fluctuates between a high and a low. Favoring the highest message processing time in the weighted mean ensures that an increase in load is rapidly evident while a decrease in load is gradual. Having calculated T_i^{avg}, the predicted processing time P_i for the whole message queue is then calculated as $P_i = (T_i^{avg} \times S_i)$ where S_i is the input queue size at the moment of calculation.

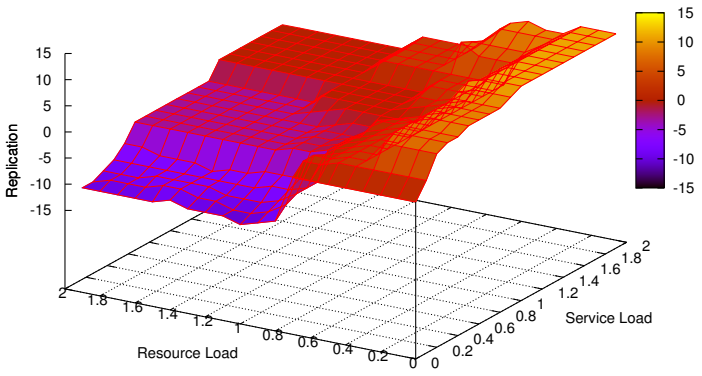


Figure 3.12: Scaling fuzzy controller surface plot. A visualization of the fuzzy rules listed in Listing 3.2.

Given a time quantum Q for a web service which could either be derived from a budget to use a resource or an alloted time quantum by a resource manager, the web service load can be calculated as $L_i = P_i/(Q - E_i)$ where E_i is the elapsed time since the web service initiated. When $L_i \approx 1$ the service is in an ideal load since it should manage to process all the data within the alloted time quantum. A load much lower than 1 indicates the web service is under-loaded while a load much greater than 1 indicates the web service is overloaded. L_i is the input value for the taskLoad in the fuzzy controller.

The resourceLoad R_i is defined as a ratio $R_i = (U_i + W_i)/A_i$ where U_i is the amount of used resources, W_i is amount of queued tasks waiting for a free resource and A_i is total available slots. A resource pool is fully used when $R_i = 1$. When $R_i > 1$, the resources are overbooked since a number of tasks are queued waiting for a free slot. R_i is the second input to the fuzzy controller.

We refer to the set of all replicated instances of the same service as the service farm. Service replication routine is restricted to one per service farm. The designated service instance which is currently responsible for running the fuzzy controller is referred to as the master of the service farm. Since the size of the service farm starts out as one, the service is automatically elected to a master. The service knows its the only instance running by querying its own input queue and deduce the number of consumers on the queue. Subsequent instances created by the master do not, themselves, become masters since they deduce that they are not the only consumers on their queue. Before a master terminates it relinquishes its own mastership by putting a master token on its own command queue.

Since all instances are consumers on the same command queue and the message broker guarantees that only one instance will consume the message, the instances that gets hold of the master token elects itself as the new master. If a master abruptly dies without relinquishing its mastership then the only way a new master is elected is when the service farm is reduced back to one. A better solution, although not implemented, is for the master to elect a secondary master who will periodically challenge the mastership by sending a command to all instances asking who is the master. If no master replies then it takes over the mastership and relinquishes the secondary master.

Figure 3.12 shows illustrates the surface plot for taskLoad L and resourceLoad R. The output, replication indicates how to scale the number of services. The plot is derived from a set of 15 fuzzy rules (see Listing 3.2).

In chapter 6 we demonstrate the detailed evaluation of this method using a bio-informatics workflow implemented as web services. We show how the modifications to Axis2 allows the web service to be submitted as a job and how the independent fuzzy controllers of various workflow tasks collaborate to achieve a fair resource usage and communication.

3.5 Summary

Scalability and resource usage efficiency are corner stone attributes in distributed computing. We have presented several approaches to tackle these challenges. We have presented prediction-based models for scaling data processing where estimations for processing are calculated on queued data. These estimations allow informed decisions on scaling data processing. The ability of scaling tasks independently enables replication of tasks to match the data production rate.

Auto-scaling is an attractive approach especially within the context of scientific workflows where single tasks can independently scale themselves. Applications that can immediately benefit from this model belong to the class of data-centric applications where data can be decomposed into atomic records and partitioned. A model for autonomous scaling was presented using fuzzy controllers that balance the scaling with the fairness of resource usage. WFaaS was also presented as a way to achieve better resource usage by reusing tasks and resources.

Through task harnessing we showed how scientific logic can be separated from underlying communication and data transport intricacies. This introduces dynamism in task scheduling. The same concept was extended to the dynamic web service architecture where the Axis2 container acted as the harness. Autonomous orchestration was also presented for web services where Axis2 containers have a myopic view on the workflow and can schedule their immediate neighbors. Through dynamic handling of web services, services have been made mobile by using queue ids as their EPR instead of the URL based EPR. A pull

```
1 IF taskLoad IS very_high AND resourceLoad IS very_low THEN replication
     IS positive_aggressive
2 IF taskLoad IS very_high AND resourceLoad IS low THEN replication IS
     positive_aggressive
3 IF taskLoad IS high AND resourceLoad IS very_low THEN replication IS
     positive_aggressive
4 IF taskLoad IS high AND resourceLoad IS low  THEN replication IS
     positive_slow
5 IF taskLoad IS very_high AND resourceLoad IS normal THEN replication IS
      positive_slow
6 IF taskLoad IS ideal AND resourceLoad IS normal THEN replication IS
     zero
7 IF taskLoad IS very_low AND resourceLoad IS high THEN replication IS
     negative_aggressive
8 IF taskLoad IS very_low AND resourceLoad IS very_high THEN replication
     IS negative_aggressive
9 IF taskLoad IS low AND resourceLoad IS high THEN replication IS
     negative_slow
10 IF taskLoad IS low AND resourceLoad IS normal THEN replication IS zero
11 IF taskLoad IS very_low AND resourceLoad IS high THEN replication IS
     negative_aggressive
12 IF taskLoad IS ideal AND resourceLoad IS very_high  THEN replication IS
      negative_slow
13 IF taskLoad IS low AND resourceLoad IS very_high THEN replication IS
     negative_slow
14 IF resourceLoad IS very_low AND taskLoad IS NOT very_low THEN
     replication IS positive_aggressive
15 IF taskLoad IS low AND (resourceLoad IS very_low OR resourceLoad IS low
     ) THEN replication IS positive_slow
```

Listing 3.2: Fuzzy rules for balancing data processing scaling, resource usage and fairness between tasks.

model allows web service to be deployed deep within a network. Back-to-back communication has been achieved through a system of message brokering.

The evaluations of these models and paradigms are described in detail in chapter 6.

CHAPTER 4

AUTOMATA-BASED DISTRIBUTED DATA PROCESSING

Distributed data process modeling is often task-oriented i.e. data processing is accomplished by modeling task ordering. The task ordering model does not always captures the essence of data processing especially when the model is designed to fit a specific distributed system. Complex data processing necessitates effective modeling which allows the understanding and reasoning of the fluidity of data processing. In this chapter we propose a new distributed data processing paradigm that describes units of data transformations as automata. Though the model can be considered as an abstract schema for data processing it also lends itself well to runtime where it acts as a data routing information allowing the creation of data processing overlay network and data processing protocol. The results of this chapter formed the bases of the following publications:-

- Reginald Cushing, Adam Belloum, Marian Bubak, and Cees de Laat. Automata-based dynamic data processing for clouds. In *Euro-Par 2014: Parallel Processing Workshops*, volume 8805 of *Lecture Notes in Computer Science*, pages 93–104. Springer International Publishing, 2014.
- Reginald Cushing, Adam Belloum, Marian Bubak, and Cees de Laat. Towards Computing Without Borders: Data Processing Plane. Manuscript submitted for publication in Future Generation of Computer Systems, 2015.

4.1 Introduction

Data processing complexity, partitionability, locality and provenance play a crucial role in the effectiveness of distributed data processing. Dynamics in data processing necessitates effective modeling which allows the understanding and reasoning of the fluidity of data processing. Through virtualization, resources have become scattered, heterogeneous, and dynamic in performance and networking. In this paper, we propose a new distributed data processing model based on automata where data processing is modeled as state transformations. This approach falls within a category of declarative concurrent paradigms which are fundamentally different than imperative approaches in that communication and function order are not explicitly modeled. This allows an abstraction of concurrency and thus suited for distributed systems. Automata gives us a way to formally describe data processing independent from underlying processes while also providing routing information to route data based on its current state in a P2P fashion around networks of distributed processing nodes.

The proliferation of cloud-based processing means that data processing is increasingly becoming service oriented, specifically each single task is an Application-as-a-Service (AaaS). The potential scale of inter-cloud systems coupled with the dynamism of the infrastructure makes it increasingly difficult to coordinate a cohort of applications in a traditional central scientific workflow systems. Furthermore, visualization opens new possibilities for compute collaboration adding to the already existing web services.

The increased variety of data means we need dynamic and adaptable systems that can easily fit new data where new data types are potentially generated on the fly and new processing paths are created dynamically from the new data. This level of reasoning about data implies that we need models to describe data as processable objects.

4.2 Paradigms of Distributed Data Processing

Distributed computing paradigms vary considerably and target specific application scenarios; process oriented such as Actor model [70] and KPNs (Kahn Process Networks) [71], large scale data oriented such as MapReduce and dataflows. MapReduce [72] is based on a *map()*, *reduce()* functions. These functions are common procedures in many data analyses problems. Many frameworks have evolved around this central concept which aim at ameliorating certain aspects such as programmability and communication. Another breed of post-Hadoop distributed frameworks address different application scenarios where the batch oriented Hadoop [73] is not ideal. Frameworks such as Storm [74] and Spark [75] aim at streaming data while others such as Pregel [76] are for large-scale graph processing. The aim of such systems is to achieve high-processing throughput on dedi-

cated clusters whereby the software stack is tuned for the specific resources. A lower level of distributed computing paradigm deals with fine grained control and communication of concurrent functions. Such a paradigm is the actor model whereby functions known as actors communicate asynchronously using messages [70]. Actors perform a certain task in response to a message, actors are autonomous and are able to create new actors creating a hierarchy of actors.

Scientific Workflow Management Systems (SWMS) come in many shapes and sizes and vary in the computational model, types of processes used, and types of resources used. Many base their model on process flow [77] and also include a form of control flow [78], others implement data flow models [79] or communication models as used in coordination languages [80] and [81]. and some propose eccentric models such as based on chemical reactions [82]. A common denominator in most workflow systems is that the unit of reason is the process i.e. the abstract workflow describes a topology of tasks configured in a certain way. Coordination languages are another form of coordinating multiple tasks.

Distributed computing programming paradigms, in a way, can be broadly categorized in how declarative they are [15]. Common concurrent programming paradigms such as message passing (e.g. MPI) and concurrent object oriented (Actors) are imperative by nature whereby the distributed execution is planned out step by step as a set of commands which defines the *how* of the processing. More declarative approaches such as dataflow, workflows, MapReduce and event-based tend to focus more on the relationship between tasks e.g. dataflow models a data relationship between tasks while workflows model a work dependency between tasks. MapReduce can be consider as a simplified workflow with an implicit relationship between a *map* and *reduce* task. In event-based paradigms the distributed entities are loosely coupled and only activated upon an event of some sort.

Related to the event-based paradigms is an automata-based programming paradigm. In such a paradigm, relationship between tasks is a state change. Progress in an automata-based system depends upon state change events. The logic of an automata-based controller program is reactive i.e. there is no start and stop of an execution but the system reacts on state changes. Our described model in the following sections follows this principle for distributed data processing.

A missing concept in many distributed computing frameworks is the separation of concern between data process model and the compute model. Most of the systems focus on the compute model thus fitting the data to the architecture. In our opinion, a data processing model should be a schema of how data can be processed. This notion would be analogous to relational databases; the data schema shows relations between data while the underlying system maps such relations to files, memory, clusters, etc. Our approach introduces this concept to data processing whereby automata is used for data processing schema while a distributed back-end can interpret the automata schema to process the data.

4.3 Provenance in Distributed Data Processing

Complexity in data processing does not only revolve around the actual computing on data but also the provenance of data. Provenance helps to track back the execution of the workflow and provides information which can help either in reproducing successful workflow execution or discovering problems that led to faulty execution. Provenance may also provide means to create links between publications and data sets, allowing to repeat published experiments; it helps in managing data-lineage, and solving the questions of trust and reputation. In case of workflow applications, provenance data has to be collected at each phase of the workflow lifecycle starting from workflow design, going through the prototyping and calibration phases and ending by the execution and result analysis.

In many applications data provenance is a way to reconstruct the data processing model from execution logs but this is a *post-mortem* approach to reason about data whereby we build data processing graphs such as Open Provenance Model (OPM) [83] after execution. Processes are often ordered to exploit the underlying infrastructure thus the same data processing workflow might look different for using grids, clouds or services. It is just to say that data-centric applications have an implicit data transition map which can be used to aid data processing, querying and data provenance. For example, coupling data with a state map in a workflow will provide a wider context for the data processing as, at any point in time, the previous, current and possible future states of the data are known. State diagrams also aid in illustrating the logical reasoning in data processing which helps finding logical faults. Markov chains have as their foundation state diagrams with added probability applied to state transitions. This could give us further insight about the state of data at some future time.

System level provenance keeps track of a context in which a simulation experiment has been performed: when the workflow was executed, on which machines, which libraries and data have been used, been produced etc. This information is used for debugging by workflow developers. Users of the workflows can require this information for validation and reproducibility. Application level provenance concerns information which had a direct impact on a successful workflow execution or a faulty one. This kind of provenance allows the scientific programmer to record important events regarding the data processing such as at which iteration a simulation is converging.

4.4 Automata as a Data Model

Automata data processing paradigm is related to the notions of automata-based programming [84] in which programs are organized in blocks of code that are triggered by state change. A clear distinction between automata-based programming and traditional imper-

ative is that program execution is separated into steps which are not ordered sequentially but ordering comes from the automata model of the program where progress adheres to the automata state transitions. This separation of steps make automata-based style as an ideal candidate to describe data processing as a set of steps where such steps can potentially be concurrent and therefore distributed.

Our unit of computing is a *data object*. We describe a data object as being an arbitrary type and size of data such as a file, row in file word in a text, binary data, etc. A data object can be a collection of data objects and vice versa a data object can be divisible into data objects. The smallest unit of a data object is called the atomic data object and this is application dependent. For example the atomic data object in a table can be a row, a column or an entry.

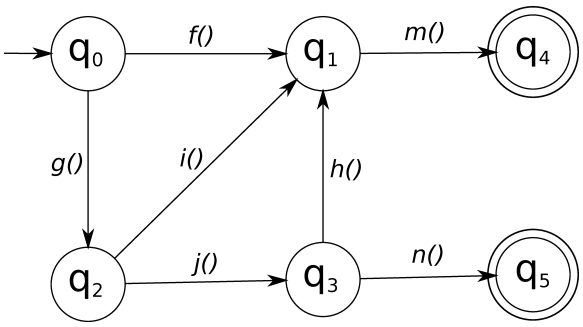


Figure 4.1: A NFA for describing 6 states of a data object, d. The set of state $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$. The alphabet set, $\Sigma = \{f(), g(), h(), i(), j(), m(), n()\}$. The start sate is q_0 while the set of final state is $F = \{q_4, q_5\}$. $\Delta(q_0, g(d), j(d), n(d))$ is a program on a data object d which processes it from state q_0 and ending with the data object in state q_5.

In our approach we employ the formal definitions of non-deterministic finite state automata (NFA) for describing data processing as an automaton. A NFA is defined as a 5 tuple $(Q, \Sigma, q_0, F, \Delta)$ where Q is a finite set of state, Σ is the input alphabet, $q_0 \in Q$ is a start state, $F \subseteq Q$ is the set of final states, Δ is the transition relation which is a relation on $(Q \times \Sigma) \times Q$. The characteristic of NFAs implies that the new state after reading symbol $\sigma \subset \Sigma$ is non-deterministic. This means the possible output state after a transition from state q, $\Delta(q, \sigma)$, is a collection of states. We extend the standard model in a way that our input alphabet is the set of operations on data *d-op*, $\sigma(d, q)$, where q is the input state, $d \in D$ is the input data object to be processed. The automaton transition function is the function which takes in a *d-op* and a data object and will transition the automaton into new states. The transition and selection function can be considered as nested functions.

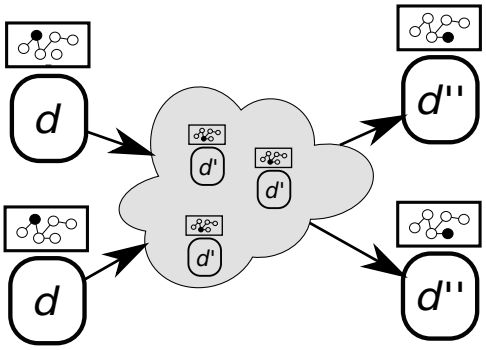


Figure 4.2: Data objects d are transitioned from one state to the next using an automaton to guide the transitions which are done using *d-op*. Data is processed along the way thus $d \rightarrow d' \rightarrow d'' \rightarrow d^k$ where d^k is the k^{th} derivate of the data object.

d-op are the functions that perform the actual state transition through processing. *d-op* accept input data set, D and set of input s-tags, T, and output transformed data set, D' and its set of s-tags, T', thus a *d-op* provides a mapping $(D, T) \rightarrow (D', T')$.

In our extended model we refer to states describing data as *s-tags* (short for state-tags). s-tags label data with processing context thus giving different forms of data different identifiers. These identifiers are pivotal in our data processing model as they allow abstract description of data transformation and its concrete processing.

Given a data object and a set of *d-op*, the data automaton describes the sequence of acceptable operations on data. In our model every data object has an associated automaton (Figure 4.2) which describes the possible final s-tag that the data object can transit to. The transition function, $\Delta(q, \sigma(d, q))$ defines the set of states that are reachable from state q with selection function σ.

Given a data object at s-tag q, only a subset of *d-op* in Σ can act on the data and produce state transitions. We define this set of *d-op* as $H_q \subseteq \Sigma$. The transition function can then be extended to multiple *d-op* such that $\Delta(q, H_q)$ defines all the states that are reachable from state q and *d-op* H_q. The latter shows that having a data object in a particular state, we can determine all its possible next states.

Since in our model the alphabet is composed of *d-op* which are operations on data thus one can consider that languages accepted by an automaton M are in fact programs such that $L(M) = \{p \in \Sigma^* \mid p \text{ is accepted by } M\}$ where Σ^* is the power set of the alphabet (*d-op*). These programs on data objects can be considered as data adapters tailor made for every data object. This gives us a powerful way of describing data processing at the fine granularity of a data object (e.g. row in a file, word in a text).

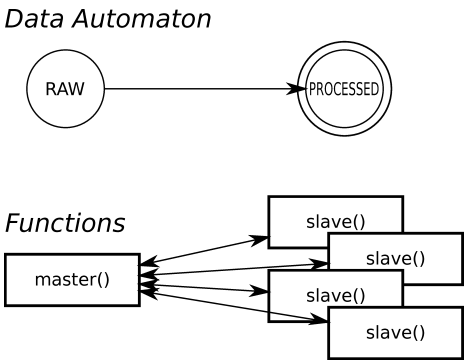


Figure 4.3: Common master-slave approach to data processing represented as an automaton. The set of states, Q is $\{RAW, PROCESSED\}$ while Σ is $\{master(), slave()\}$.

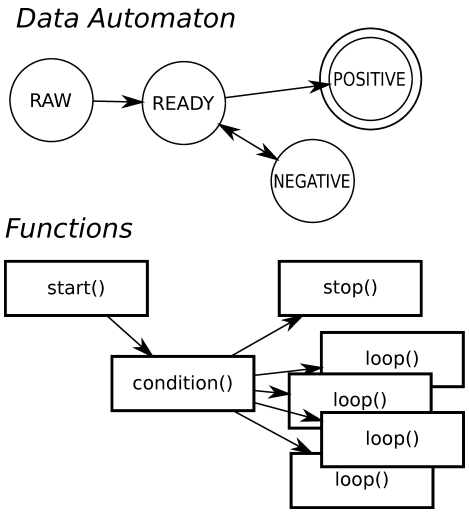


Figure 4.4: An iterative data processing scheme represented as an automaton. The set of states, Q is $\{READY, POSITIVE, NEGATIVE\}$ while Σ is $\{start(), stop(), condition(), loop()\}$.

Figure 4.2 illustrates the how the model is applied to data processing; atomic data objects such as a record or a file are encapsulated together with an automaton. The automaton defines the data processing as state transitions. A system capable of interpreting the automaton can process and transfer the data to other nodes for further transitions.

Figures 4.3 and 4.4 illustrate two simple automata to describe classic distributed data processing which suffice to introduce our model. Figure 4.3 illustrates a typical master-slave approach. The automaton is a trivial 2-state graph which describes data in its unprocessed state as RAW and processed state $PROCESSED$. The execution of the automaton includes two functions *master()* and *slave()* which are *d-op* according to our model. The *master()* does an implicit state transition from the empty s-tag to RAW while *slave()* *d-op* does the transition from RAW to $PROCESSED$. As a means to speed up execution a scheduler may run many *slave()* functions. The latter is not represented in the data automaton which is solely intended to describe the data transformations independent of the way the execution takes place. This abstraction provides us with a means of describing data processing at an abstract level.

Figure 4.4 illustrates a second typical example which includes processing data in loops until a condition is met. A common approach in parallelizing such a loop is to unroll the

loop and distribute it. The simple automaton captures the three states in which data can be; the $READY$, $POSITIVE$ and $NEGATIVE$ states. The functions *start()*, *stop()*, *condition()* and *loop()* provide the *d-op* for the automaton. *start()* and *stop* provide transition to/from the empty state, the *condition() d-op* provides both transitions from $READY$ while the *loop() d-op* provides the cyclic transition back from $NEGATIVE$ to $READY$. As with the previous example the execution of many *loop()* instances does not change the description of the data processing.

Apart form a state description of data processing the automata also describes the allowable programs (function permutation) for processing data in a certain way e.g. in Figure 4.3 we know that data can only be considered acceptable if *master()* and *slave()* are executed in certain order and only once while from Figure 4.4 *condition()* needs to be executed at least once and *loop()* any number of times so {*start(), condition(), stop()*}, {*start(), condition(), loop(), condition(), loop(), condition(), stop()*} are both acceptable programs on the data. A data automaton can recognize multiple programs so given a data processing automaton we can produce the set of all programs $P \subseteq \Sigma^*$ that are acceptable.

As we described in our model, *d-op* are the functions performing operations on data and transit the data to new states. *d-op* can transform multiple input s-tags into multiple output s-tags thus internally in a *d-op* an $n \times n$ mapping exists. The multiple input s-tags are combined together in boolean logic and similarly is done for the output s-tags. In Figure 4.4 the *d-op condition()* transits to 2 s-tags $POSITIVE$ **or** $NEGATIVE$ similarly it can produce both simultaneously or any boolean combination of any number of s-tags. This allows *d-op* to multiple data objects in different states from one data object.

4.5 Data Packet as a Unit of Computing

Communication between *d-op* hosted on nodes is done with data packets. Data packets are first-class citizens in our proposed model. A data packet is a self-routable encapsulation of a data object and optional code with meta-data to facilitate state transitions described in our automata model. Packets are stateful (i.e. the packet carries much of the data processing state in it) which allows nodes to be stateless to a certain extent. Figure 4.5 presents the basic packet construct; the *packet id* is a concatenation of 3 separate header fields: a *ship id*, a *container id*, and a *box id* which together form a hierarchical naming scheme analogous to cargo shipping. The id scheme allows for related packets to be given the same *ship id* which acts as an execution context id. The data packet is modeled as a container into which arbitrary data can be placed, extracted, modified, and replaced.

The *d-op* code section is an optional section which carries a list of codes to be deployed at the receiving nodes. This feature allows the unrolling of the automaton *d-op* on the network. Codes can also be deployed in quantities which allows multiple instances of the

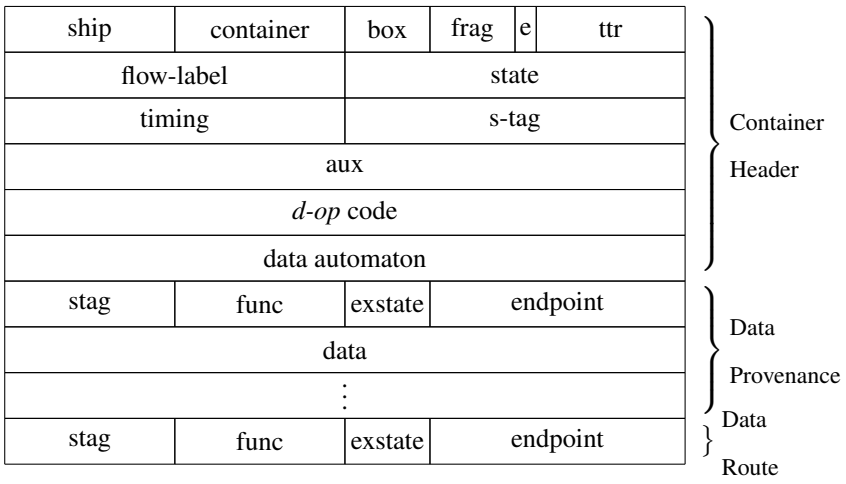


Figure 4.5: High level structure of a packet encapsulating a data object, *d-op*, automaton, provenance and routing information. Together *ship, container* and *box* represent the packet *id*. The naming is hierarchical i.e. a ship has many continaers and a countainer has many boxes. These ids are not associated with source and destination as would be with traditional network protocols but the id is given to data objects. Fields such as *ttr* (time-to-resend), *aux* and *flow-label* aid data routing. Some other fields aid specific scenarios such as *timing* is used when benchmarking. A record of where the data object was and the results from the last state transition are kept (optionally) in the data provenance section while the next hop (state transition) is appended at the end of the packet.

same code to be deployed on different nodes. The auxiliary (aux) section is a bit field enabling certain packet features such as acknowledgments. multi-packet and timing. The data automaton section includes the automaton for the data processing. This describes the allowable states the data packet can be transitioned too. The field can be null in which case the data packet discovers its possible state transitions from the data routing table on each node. This transcends data routing to exploring possible data processing paths.

A data packet has the option to carry data from every *d-op* output. This feature allows the direct provenance capture within the data packet itself. Although being a powerful feature it also has a snowball effect on the packet size and thus might not be a feasible feature for all applications. The data route entry is the last entry in the packet. This describes the next node hop where the packet should be sent.

4.6 Computing Flow Control

Since the unit of processing is a packet and packets are communicated between nodes in a P2P fashion, packet flow control becomes an essential part of the framework for maintaining coherent and stable network. A mechanism for reliability can be enabled, albeit with a performance penalty. With reliability enabled nodes share responsibility of a data packet. The mechanism works as follows: Upon sending a data packet a node *A* retains the packet in a buffer until the peer node *B* sends back a Processed ACKnowledgment (PACK) back to *A*. A PACK is only sent after node *B* finished processing the packet and has dispatched it forward at which stage *B* becomes responsible for the packet. Upon reception of a PACK at *A*, the latter is relieved of its responsibility. If a PACK is not received in a timely fashion, node *A* will activate the TTR field in the data packet and resend the packet upon time expiration. Each node is equipped to detect duplicates, thus if *B* where to receive the duplicate packet it will reject it immediately. The TTR field can be tuned for different packets so that process hungry packets can have higher TTRs. As can be imagined this mechanism will put extra pressure on the system especially on packet buffers which are awaiting acknowledgments and thus the whole mechanism can be disabled in scenarios where it is not needed e.g. a streaming application.

An other flow control mechanism is packet coalescing; the premise for this feature is that packet overheads can be partially amortized by packing in more data into one data packet. Although this can be user defined in the *d-op* implementation, we try to achieve this automatically by dynamically calculating packet efficiency and ameliorate the efficiency by grouping packets together. The efficiency formula is based on:

$$ceff(p) = \frac{exec(p)}{exec(p) + overhead(p)}, \tag{4.1}$$

where $exec(p)$ is the execution time of packet p, $overhead(p)$ is the overhead for sending and unpacking the packet. Although the overhead is influenced by the size of the packet, a fraction of the overhead remains a constant such as communication latency and call stack latency for every packet. In very small packet loads the overhead can be much more than the actual packet processing which lead to inefficient communication. A way to increase this efficiency is to increase packet sizes which will increase $exec(p)$.

Although this coalescing method has the effect of increasing efficiency it can also impede parallelism. The rational behind this is that any number of packets coalesced in a single packet are destined to be executed in serial on some node thus in a distributed system some nodes might lay idle while others are busy processing huge packets. This phenomena leads to what we describe as potential parallelism efficiency.

Lets assume we have an arbitrary size of data D and this data can be split into it atomic form d such that $d \in D$ for example a line or a word in a file. The cardinality (the number of atomic data objects in D) of D is given by $|D|$. Now lets assume we have some parallel program which is executed on every atomic record d. To scale up the processing it is common to initiate multiple processes and partition D into chunks of data objects, C_i, such that $D = \bigcup_{i=1}^{n} C_i$. Every process gets a chunk of data, C_i, thus allowing D to be processed in parallel. A matter of tuning the system revolves around the number of atomic data objects per chunk. Having a number of resources N, a straight forward split is $p = \lceil \frac{|D|}{N} \rceil$ which means that with enough resources to match the number of data objects, $N \geq |D|$, $p = 1$ since p can not be less than 1 as atomic data objects can not be further subdivided. If we increased p to 2 meaning that we split our finite data into chunks of 2, at most only half of our infinite resources will get a data chunk. The latter shows that with $p = 2$ and $N \geq |D|$ we get a parallel efficiency of 0.5. Parallel efficiency is generalized by the formula

$$peff(p) = \frac{|D|}{p \times N}, p \geq 1, \tag{4.2}$$

where $|D|$ is the data window size, p is the number of coalesced packets and N is the number of nodes with *d-op* that can process the data. $peff(p)$ is a graph of the form $\frac{1}{x}$.

Another packet control flow that we investigated has to do with networks of queues. Each node receives and sends packets thus a node can be modeled as having two buffer queues, an inbound *rx* and outbound *tx* queue. These buffers are serviced asynchronously and independent of the running *d-op*. In any queue system backlog is considered detrimental to the whole system thus some level of queue control is needed to minimize such backlogs. Queue backlogs are problematic in two main ways; the first is buffer flooding where backlogs flood memory which decreases node performance to the extent that nodes can crash; the second is input queue load imbalance which happens when data is partitioned unfairly (an unfair data partition is such that slower nodes get more *work* than faster nodes) between processing nodes. In our model, backlog is defined as the predicted data object compute time remaining in the queue and not just the number of packets in the queue.

A way to deal with backlogs in a queuing network is to minimize the Lyapunov drift [85]. The Lyapunov quadratic function defined by $L(t) = \frac{1}{2} \sum_{i=1}^{n} Q_i(t)^2$, where $Q_i(t)$ is the queue backlog at Q_i at time t. The drift, $\Delta(t) = L(t+1) - L(t)$ is defined as the change in the backlogs between time intervals t. Minimizing $L(t)$ means reducing overall backlog of the whole network which in turn reduces strain on nodes and allows for better scaling by removing the need to have rx to rx packet migration.

Since our backlog is based on predicted data object compute time remaining in the queue, a method needs to be in place to gather such information. Timing bits in the packet header allow every *d-op* to time packet execution. From packet processing, histograms are generated where packet payload sizes are sorted in bins of execution times. Every packet ameliorates the running average of its bin. *d-op* implement regression algorithms to fit predictive models on the histogram data. By default every *d-op* has a linear regression implementation which can be overridden since linear regression assumes data processing time is linear to the data size which is not the case for all types of data processing.

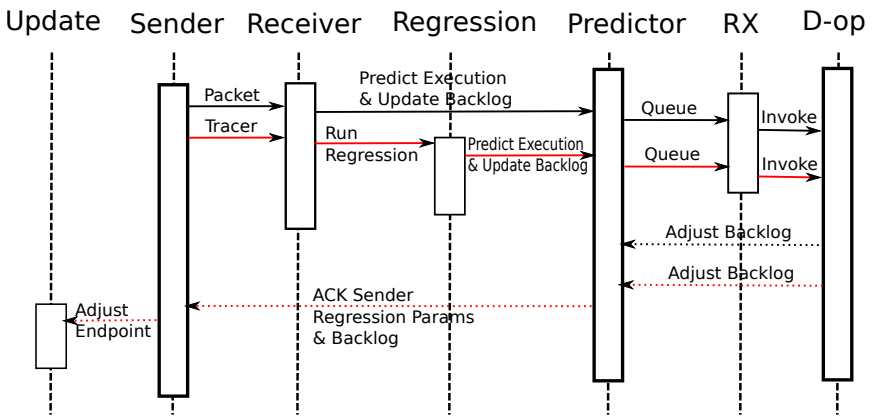


Figure 4.6: Sending tracer packets to calculate compute backlog and update sender with regression parameters to adjust the packet rate limiter.

On every incoming packet and before being queued on the rx queue, the regression parameters are used to predict the compute time of the new packet. This prediction information is attached to the packet header. The backlog for a particular node is the total predicted compute time of these packets. The method is intended for decentralized systems therefore a method of using special packets, referred to as tracer packets, was devised to collect prediction information to be used by the source as the bases of the packet control limiter. This is done through *tracer* packets as show in Figure 4.6. A tracer bit can be activated in a packet. This bit instructs the receiver to run the regression algorithm on the *d-op*'s histogram data, update the regression parameters, calculate the compute backlog and send an acknowledgment back to the sender with the backlog and regression parameters.

Tracer packets are sent at intervals Figure 4.6 which minimizes the need to run regression and send acknowledgments on every packet. The sender updates the endpoint with the regression parameters and backlog. The sender will then start limiting packet dispatch by predicting the execution time using the receiver information, the limiter sends the packet

and disables the endpoint for the predicted execution time since the receiver should, in theory, be busy computing and any other packet sent will be queued up on the rx buffer. Since there will always certainly be a deviation from the predicted time and the actual execution time, packets are still bound to be backlogged. To adjust this deviation the backlog compute time is included in the tracer acknowledgment and this allows the sender to correct the deviation by initially disable the endpoint until the backlog has been cleared.

This rate limiter is set per endpoint thus a sender with multiple receivers will manage multiple rate limiters. In a network each receiver is also a potential sender which also implements the rate limiters for its forwarding traffic. The overall outcome of the distributed rate limiters is the continuous minimization of backlogs.

4.7 Data Transition Functions: *d-op*

d-op implement the data processing and state transition in our proposed automata model (Section 4.4). These functions are loaded dynamically at startup of every node or deployed during runtime. As described in the model, *d-op* map input data and s-tags to output data and output s-tags, $(D, T) \rightarrow (D', T')$. As part of the meta-data describing each *d-op* is the input s-tags and output s-tags.

d-op acts in a service oriented way [86] where functions are invoked upon a request through data packets and an explicit *dispatch()* is called instead of a function return. A function is essentially an implementation of an interface class that overrides a *run()*, and optionally *pre_run(), split(), merge(), on_load(), on_unload()* methods. *d-op* are annotated with the list of input s-tags and output s-tags. These s-tags are combined in boolean logic which allows data splits and merges.

A function is invoked as follows: upon reception of a data packet, the relevant meta-information from the packet is extracted and a lookup for the appropriate *d-op* is done. After a staging sequence where data files might be downloaded from the previous node or other data sources, the *pre_run()* and *run()* methods are called in sequence. The *pre_run()* allows the function to do pre-checks before accepting the packet such as checking if all dependencies are met. If the pre-run fails then the packet is re-dispatched to another node. Each *d-op* can optionally define a *split()* and *merge()* methods. This scenario is useful when processing data packets in parallel would improve execution time. The *split()* fragments the packet into many packets which enables the data to be partitioning. The newly generated packets will be distributed to multiple instance of the same class. The results from the worker peers will be returned to the original peer where the *merge()* method is called on every packet thus the splitter node acts as a temporary master node. The *merge()* method implements a data specific merging routine. Section 4.7.1 shows a typical code implementation of a *d-op*.

4.7.1 Pumpkin Data State Network Implementation

The model described in Section 4.4 is implemented as the PUMPKIN framework[1]. The aim of the implementation is to exploit the automata-based data processing model as a decentralized distributed framework. The model lends itself well to distributed computing since *d-op* can be easily distributed while the s-tags provide the necessary connectivity information.

PUMPKIN treats data processing as a network plane whereby data is encapsulated in packets which are routed on the data processing plane. The automata provides the routing information thus a packet of data having an automaton as part of the header can find its way in the network. This is achieved through a P2P distributed system for routing and processing data based on s-tags. The distributed characteristic of the system removes control centrality and the P2P characteristic allows for data being processed to pass directly between nodes thus minimizing third party data stores for intermediate data. The architecture is designed with the emerging cloud computing paradigm and virtual infrastructures in mind therefore one of the goals of PUMPKIN is to dynamically adapt to the varying network and resource performances in globally distributed virtual resources.

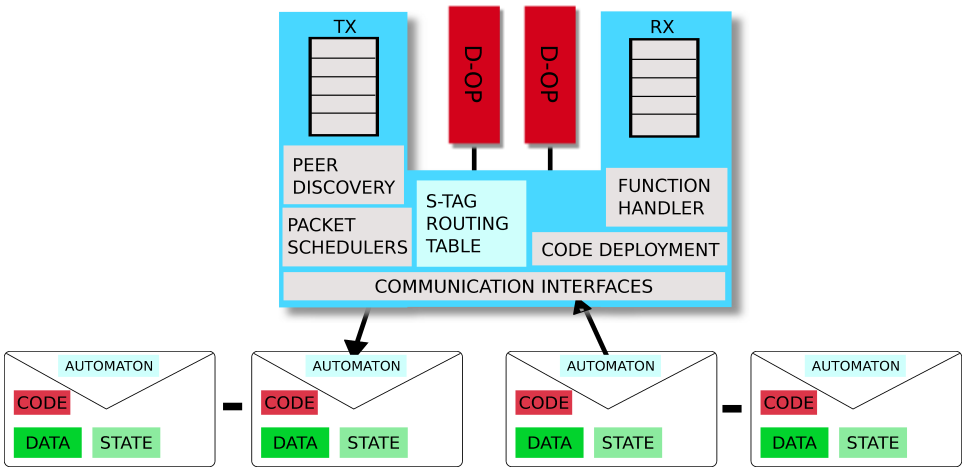


Figure 4.7: Anatomy of a PUMPKIN node. Connectivity is provided by a set of handlers (right), the core components (center) provide data packet handling and routing while the functions are the selection functions described in Sec. 4.4.

The architecture of a single PUMPKIN node in the network is illustrated in Fig. 4.7. The architecture builds around the concept of dynamically loading functions as is done in many web service containers. In addition to dynamically loading functions, a PUMPKIN node

[1]https://github.com/recap/pumpkin

implements a stack of control functionality to achieve the P2P capabilities. Most notable is the data routing based on s-tags. Each PUMPKIN node exposes a set of interfaces for accessibility. Communication interfaces listen for incoming packets. Packets can include code and data (section 4.5). The relevant information about the *d-op* to invoke is extracted from the packet. If the *d-op* is not present on the current node, the packet is re-dispatched.

```
47  ##{ "object_name": "bedpostX",
48  ##"parameters": [{ "state" : "DTI_PREPROC" }],
49  ##"return": [{ "state" : "DTI_FIBER | ERROR"}] }
50
51  class bedpostX(PmkSeed.Seed):
52      def on_load(self):
53          #Staging dependencies.
54          pass
55      def pre_run(self, pkt, data_object):
56          #Packet pre-check return True to accept or
57          #False to reject
58          pass
59      def run(self, pkt, data_object):
60          #Main routing called on every data packet.
61          new_data_object = self.process(data_object)
62          if new_data_object:
63              self.dispatch(pkt,new_data_object, "DTI_FIBER")
64          else:
65              self.dispatch(pkt,data_object, "ERROR")
66          pass
67      def on_unload(self):
68          pass
69      def split(self, pkt, data_object):
70          #Split data_object, create new pkts and
71          #dispatch them using DTI_PREPROC s-tag
72          pass
73      def merge(self, pkt, data_object):
74          #Accumulate packet fragments
75          #until a merge is necessary.
76          pass
```

Listing 4.1: A simple *d-op*, *bedpostX()* (Section 6.5), template defining 2-state transitions *DTI_PREPROC* → *DTI_FIBER* and *DTI_PREPROC* → *ERROR*. The s-tags are defined in lines 1 to 3 of the code. The *dispatch()* function is called to release the data object so that it can be sent to the next *d-op* which can accept *DTI_FIBER* or *ERROR* s-tags.

Listing 4.1 shows a typical *d-op* implementation as described in Section 4.7. PUMPKIN is written in Python and *d-op* are classes which override a number of functions. After calling the *dispatch()* function, *Pumpkin* will lookup is state routing table to find any possible node that are hosting *d-op* which accept the new state of data. PUMPKIN will then activate flow control routines discussed in Section 4.6. If multiple instances of the same *d-op* are found, PUMPKIN will try to load balance the packets on all the nodes using a default round robin method (other schedulers can be implemented). In the case that multiple different *d-op* are found that accept the new state of data then the packets are replicated to all different instances. The PUMPKIN core implements two queue buffers for packet input *rx* and for dispatch *tx*. These buffers separate the interface abstraction layer and the *d-op* layer (Figure 4.8. The interface abstraction layer includes a list of communication adapters such as ZeroMQ[2] , RabbitMQ[3] and shared memory. Once PUMPKIN puts a packet in the *tx* queue then its up to lower interface abstraction layer to choose the adequate adapter to use for sending the packet. Similarly each adapter is listening for packets on their respective interfaces and queue packets in the *rx* when packets are received. It is then up to the upper layer of PUMPKIN to parse the packet and invoke the correct *d-op*.

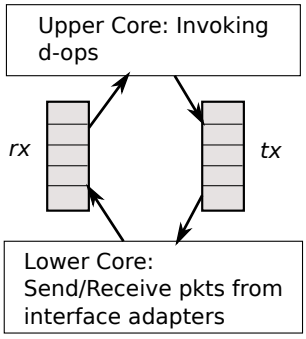


Figure 4.8: PUMPKIN architecture split into lower core and upper core. Upper core deals with *d-op* invocation while the lower core deals with packet transmission, reception and flow control.

Node discovery is done in two main ways. Nodes on the same local network will discover each other through UDP broadcasts. This allows PUMPKIN to be easily setup on local networks. A second method for discovering globally distributed nodes is done through a publish subscribe method. Nodes can be configures with a RabbitMQ server where each node can broadcast its presence including which *d-op* are being hosted and connectivity information such as public IPs and ports. Nodes are configured in groups. Nodes within

[2] `http://zeromq.org/`
[3] http://www.rabbitmq.com/

a group are discoverable by each other. The grouping limits interference from discovering nodes from other users. Through broadcasting information each group builds an identical routing table which allows each node to immediately route to packet to the next node. Although this method might not seem scalable with larger routing tables it is adequate enough for the number of nodes we use per group. Nonetheless, if need be, a more dynamic discovery can be envisioned using distributed hash tables where node lookups can be done remotely. The contents of the data routing table is based on s-tags. What the routing table describes is how to reach nodes that can make transitions from a certain s-tag. So for an s-tag RAW we would have an entries in the table which would point to all the *d-op* that can transit RAW to other s-tags.

Nodes broadcast their means of communication. These are referred to as *endpoints*. Every node can have multiple ways to be reached e.g. multiple network interfaces, message queues and distributed file systems. In PUMPKIN we treat every communication possibility as an endpoint of the node. Typical endpoints would be a private IP, a public IP, a pipe and a message queue name. This information is broadcast to other nodes and the sender is responsible to choose which ones work and allows nodes to communicate on local networks as well as nodes behind restrictive NATs.

Evaluation of the described implementation is given in chapter 6 where we demonstrate two different application scenarios: a streaming data processing application and a file based biomedical application. We show how both can be described using our model and how prediction-based data processing flow control on data packets is applied.

4.8 Summary

In this chapter we introduced a new data processing paradigm based on modeling data as automata. This paradigm tackles the challenge of describing data processing at an abstract level independent from the task ordering and execution specifics. We believe that this model is complementary to traditional task ordering models. The model takes a data centric approach to describe abstract data processing as a sequence of state transitions. Through PUMPKIN implementation we showed how the automata provides information about data during the fluidity of processing which guides data to computing. The distributed decentralized architecture of PUMPKIN and the self-routable data packets creates a data processing plane where data processing is reduced to a protocol which enables clients to inter-operate. The usage of data packets as data processing parcels allows us to investigate added data routing attributes. In the presented model data is routed based solely on its state. Additional attributes can be easily added to the packet such as energy and security which would allow packet schedulers to choose were to send the data based on such attributes. PUMPKIN is our approach to data processing as a data transform network

as illustrated in Figure 1.2 where it is placed ontop of the SDNs and programmable infrastructure layer. The data transformation network layer provides information about data processing such as data processing times and data routing which can be picked up by the underlying layer and used to setup and optimize infrastructure. This technique including PUMPKIN has been presented and demonstrated as part of the big future of data [87].

In our implementation, data states are tags. These tags give limited context to the data while processing. So as to aid collaboration tags need to be given a meaning, this we believe can be done by associating tags with ontologies thus a s-tag would in essence be a URL to an OWL (Web Ontology Language) class and the automata would define the transitions between such OWL classes. Since our architecture uses the automata as a means to route data on the network, the combination of OWL and PUMPKIN would enable ontology-based routing for data processing. The latter would be a step towards combining data processors through semantics as envisioned in [88]. Since the data packeting mechanism is intrinsically a protocol, heterogeneous application layers can be brought together using the same protocol. Ongoing research is investigating the use of PUMPKIN to extend the traditional resources using web browsers [89]. Another area of interest is to apply our model to data stores [90] making them *smarter* whereby data objects are files and each file having an automaton associated with it describing the possible states of the file thus allowing users to request different states of the file e.g. image file resolution and a compute back-end can generate the file on-demand.

CHAPTER 5

LINKING DATA PROCESSING THROUGH SEMANTICS

In the previous chapters we have discussed and presented data processing in its execution form where we dealt with infrastructure, scaling and modeling. The latter chapters dealt with data processing as isolated actions where we assume the user has whole knowledge of the data processing system. To extend data processing beyond isolated groups so as to ameliorate data science, knowledge about data processing needs to be shared. Such knowledge includes the semantics of processes and data involved in data processing which would allow groups to extend and reuse this knowledge. In this chapter we will go beyond traditional distributed computing frameworks and investigate the possibility of automatically connecting global processes in a similar way as envisioned in linked data. Our take on the matter is the fact that data and process are intertwined and solely linking data is but half the story. We believe that linking processes will enrich the data. The results presented in this chapter formed the bases of the following publication:-

- Reginald Cushing, Marian Bubak, Adam Belloum, and Cees de Laat. Beyond Scientific Workflows: Networked Open Processes. In *IEEE 9th International Conference on eScience*, pages 357–364, 2013.

5.1 Introduction

Scientific Workflow Management Systems (SWMS) have, for the past years, been instrumental in the area of distributed scientific computing. Although many SWMSs such as Taverna [50], WS-VLAM [39, 92] and Pegasus [93] have emerged with different capabilities, most share some unique characteristics. Most noticeable is the fact that workflows are designed by humans whereby a scientific programmer implements the component or process functionality and the domain scientist builds the workflow representing the experiment. This method, recently enhanced with the idea of research objects [94], works well when dealing with a handful of processes but with the ever increasing number of processes and services for scientists to choose from, designing workflows is a tedious task. For example, BioCatalogue which hosts a catalog for bioinformatics web services contains over 2500 services. Furthermore, the proliferation of repositories such as myExperiment [95], SADI [96], ProgrammableWeb [97] indicate that sharing services within a scientific community is commonplace. Building on top of scientific catalogs by means of reasoning about federated service registries could open new ways for building complex workflows. For these reasons we believe that the next generation of distributed scientific computing will deviate from traditional isolated SWMS and move towards open systems that can autonomously construct workflows using a global space of processes and minimal declarations by scientists to construct experiments. With a global space of processes, adequate semantics and tools to work with these, interoperability between processes can be discovered giving rise to *Networked Open Processes - (NOP)*.

The Web of Data a.k.a Linked Open Data (LOD) is the principle whereby data and the relations between scattered data sources are exposed on the Web. This effort is derived from research on the Semantic Web [98] and the need for scalable structured machine readable data. Linked Open Data follows 4 main principles as stated in [99]:

- Use URIs as names for things,
- Use HTTP URIs so that people can look up those names,
- When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL),
- Include links to other URIs, so that they can discover more things.

The main enabler behind Linked Open Data is RDF (Resource Description Framework) [100] which is a meta-data model for describing concepts in a machine readable structure. RDF describes objects using a subject-predicate-object structure. This simple model allows the description of complex objects by creating relations between RDF statements:

- a:person foaf:name “Asimov” .
- a:person auth:wrote book:Foundation .
- book:Foundation book:hasGenre “Science Fiction” .

These set of RDF statements create a link between subject person and book. Since one of the principles of LOD is that names should be given URIs then person and book would be global unique identifiers and publicly accessible since HTTP URIs are used. The publicly accessible URIs means that linked data is open and anyone can link to them. RDF relations between scattered triple store (RDF databases) builds the fabric of the global data space and is referred to as LOD. SPARQL is the common method for accessing LOD. SPARQL is a RDF query language similar in syntax to SQL but specifically tailored to deal with subject-predicate-object statements.

In LOD, we often think of data as representing something concrete such as data about a city or a protein but here we distinguish between two kinds of data. This traditional data we refer to as *inanimate* data which is just value-based data. For example, *city hasName “Amsterdam”*, Amsterdam is an inanimate datum which means it is just a name and nothing else. In contrast, *animate* data is the type that deals with data other than dead-end values such as *service hasOperation “sayHelloWorld”*, sayHelloWorld is a function and thus can be invoked and executed.

Describing processes or services (in this chapter we differentiate between process and service; by process we mean any kind of task such as a cloud or grid job, a service is a subclass of a process and refers to traditional services such as RESTful and SOAP) in the context of LOD is an emergent research field. Some work which we consider of particular importance in this field is done in [101, 102]. SADI [101] is a web service framework for publishing and discovery of scientific web service. This is made possible by using semantics in every level from publishing to consuming and producing data. SADI services consume and produce RDF thus inputs and outputs are defined in OWL-DL. This ability allows SADI services to embed into the Web of Data as services consume and produce linked data. The RDF store describing SADI services is an example of an animate data store. This store is used later on this chapter and underpins the motivation behind our research.

Another similar approach is Linked Open Services [102]. In this approach, traditional services are wrapped so that they consume and produce RDF. A mechanism of *lowering* RDF input to the native syntactic input type such as JSON is employed. The same technique is used for output in which case the syntactic native type is *lifted* to the semantic RDF type. The aim of this work is to facilitate the interactions between services and linked data.

In [103] the authors describe the semantic integration approach to modeling and transcribing complex science domain knowledge into well-structured information models based on semantics.

The first two mentioned related work are similar in approaches; they employ semantics to annotate web services and devise methods for services to consume and produce RDF data. The focus of both works is the blurring between LOD and RDF-capable services. These type of services produce readily linked data and are referred to as Linked Open Services or Processes. Our work is focused on process-to-process interaction rather then service-data interaction. To limit confusion between Linked Open Services and our approach we refer to our approach as Networked Open Processes (NOP).

Other semantic service repositories such as bioCatalogue [104], myExperiment [95] and ProgrammableWeb [97] focus more on web APIs (another way how to look at web services) and their mesh-ups (combining web APIs together in a pipeline fashion). The emergence of such repositories further motivates our motivation of applying structured reasoning by federating semantic repositories.

5.2 Building Networks of Interoperable Processing

The concept of Network Open Processes (NOP) stems from the similar notion being tackled with data as the Linked Open Data (LOD). In LOD data are exposed publicly in an RDF/OWL form. This semantic annotation of data coupled with the capability to query such data with dedicated SPARQL endpoints means that complex queries can be answered. Linking and uniquely identifying web data means the web is transformed into a global data space. To date many datasets have been exposed using LOD principles these include governmental, life sciences, literature, and media data. We use a process to deliberately distinguish between traditional web service (RESTful, SOAP, etc) and task oriented processes which do not necessarily follow a Service Oriented Architecture (SOA) such as Grid jobs but nonetheless can still be semantically described using the basic notions of *operation, input, output* and given an adequate platform can still be accessible using the same techniques as for services.

Similarly to LOD, the same notion can apply to processes and services by semantically annotating them in a way that makes it easy for discovering networks between processes. Many of the processes and services developed for scientific workflows are either never made available, locked in process silos, or else are cataloged with no real effective means to to reuse them. Many of these processes can be described in a way that makes them linkable, discoverable, and reusable. The ability to link processes through semantics depends on what type of semantics are used. To date many semantic descriptions have been proposed mainly in the area of web services namely; OWL-S[105], WSMO[106],

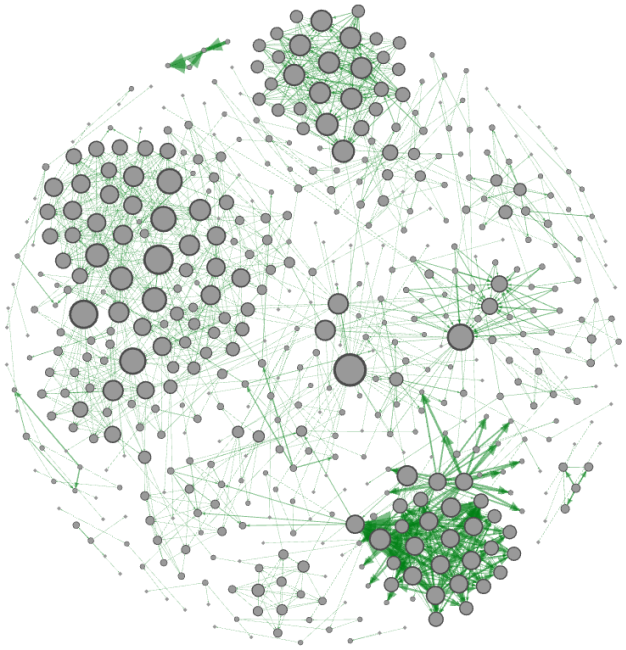


Figure 5.1: A NOP built from an RDF store describing SADI-services [96]. Vertexes are operations described in BioMoby Semantics provided. Edges show connections between output and input parameter.

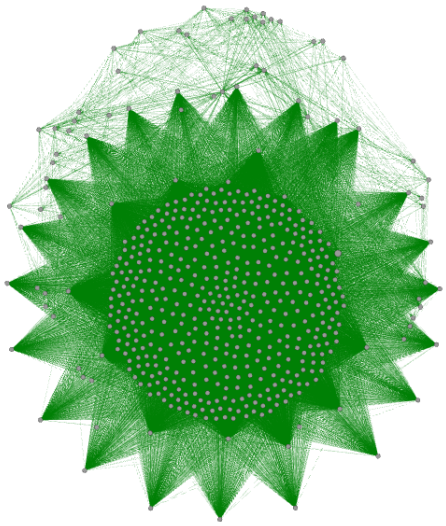


Figure 5.2: A dense cluster of services that has been extracted from Figure 5.1 for clarity and illustrates the lack of adequate semantics.

SAWSDL[107], MSM[108], and BioMoby[109]. F or effective process linking, the semantic of such processes should include the notion of inputs and outputs. This provides the framework by which processes can be semantically networked i.e. processes that have their output semantic datatypes as input to other processes. BioMoby [109] is one such semantic language used by the SADI framework. In Figure 5.1 and Figure 5.2 depicts the potential interoperability between the thousand or so operations. In this scenario interoperability means that a process output also exists as, at least, one input to another process so for a process $[o_1, ..., o_x] = F(i_1, ...i_x)$ a link exists if one of the outputs, o_x is the input, i_x to another process. We performed the linking on the text names of inputs and outputs, specifically on the *hasParameterNameText* predicate. The *objectType* predicate was ignored due to the convention of naming the object type of outputs with *_Output* instead of naming the actual data type. This results in operations never matching since an object named output will not be an input to another process. Using datatypes for linking requires deeper parsing of the object type, specifically looking into the OWL ontology of the input/output. Although this is possible, it is only possible for a limited number of services. Using text names gives us a clear idea about the potential interoperability which suffices as a motivation for this work.

An interesting point to note from Figure 5.1 is the formation of various clusters. A deeper inspection of these clusters shows that within the clusters, the nodes share the same few edge labels which in our case means input or output data names therefore these clusters are *type clusters* where the processes are acting on similar data types. For example the bottom right cluster deals with *alignment* edges and therefore functions which deal with sequence alignments such as *EMBOSS_Water* and *ClustalW* reside there. Type clusters tend to contain two main kinds of processes; producers and consumers. The previously mentioned processes are producers while other processes such as *runPhylipProtpars* which generate phylogenetic trees from alignments are consumers. What is also evident from the investigated network is that only a small number of processes provide the core scientific logic while a larger number of processes are tools for manipulating data types. The latter type we refer to as *data-tool* processes while the former are *core-logic* processes. Although one might think that data-tools are of lesser importance, in reality they provide much of the network connectivity. This is apparent when ordering nodes with their respective betweenness-centrality whereby most of the top scorers are data-tools. Furthermore, the graph in Figure 5.1 follows a power-law degree distribution though, its hard to generalize and claim that such process networks are scale free. Nonetheless it gives us an indication that some processes are much more popular than others and serve as network hubs. Although data-tools provide connectivity they can also induce noise in the graph as shown in Figure 5.2. The inadequate use of semantics meant that many processes shared the same edge label *sequence* which resulted in this dense cluster thus a trade-off lies in

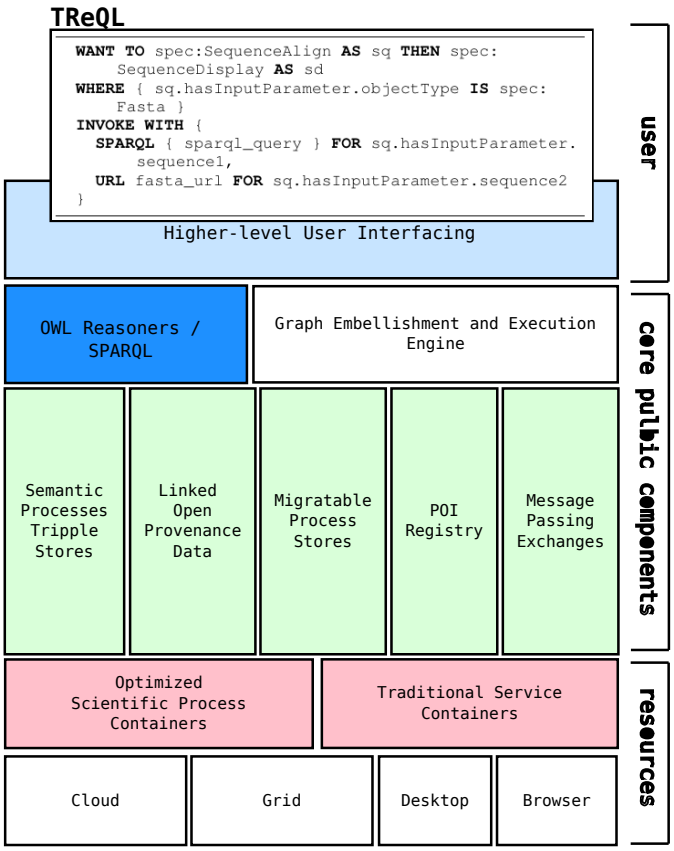


Figure 5.3: High-level Framework for Networked Open Processes. The user layer is the interface into the system such as TReQL. The core components represent the core services such as network reasoners, triple stores and message exchanges. The resource layer is where actual execution takes place.

the level of data types used; too generic and produce dense clusters, too specific and risk disjoint clusters.

Through RDF triple stores many scientific repositories like the one investigated here can be made available thus adding to the NOP network. These networks present a challenge to distributed scientific computing; how can one make sense of such networks? Traditional workflow systems rely on the fact the workflows are manually generated but with such networks where workflows can entail many processes this manual procedure is surely not scalable. For these reasons we propose a framework that aims to ameliorate the distributed scientific computing, re-usability and collaboration.

5.3 A Framework for Interoperable Processing

Figure 5.3 depicts the main building blocks of the proposed layered framework to realize NOP. At the user layer are the user and agent interfaces to the framework such as a SPARQL endpoint or TReQL (see Section 5.3.2). The top part of the second layer, core public services, form the core of the system. Here semantic reasoners are employed to find graph paths for the user requests. The discovered graphs, which could possibly be disjoint, are skeletons of workflows that can accomplish the complex task requested by the user. This abstract graph is embellished with necessary parameters to make it executable by a graph engine. The engine has at its disposal a plethora of core public services for embellishing abstract graphs such as messaging servers that can be used for process-to-process communication. Provenance and heuristics can be used to feed back information which can later be used to optimize the network. E.g. frequently used edges get emphasized and have better chance of being chosen in future searches. The lower layer of the framework stack are the resources and their container abstraction. To make most use of global computing resources the framework does not limit itself to traditional web service architecture where a service is simply publicly hosted. Common web service containers are not optimized for scientific computing and therefore are not easy to work with in typical scientific scenarios such parametric studies. Smart containers can harden services for scientific computing (see section 5.3.4)

5.3.1 Semantic Description of Processes

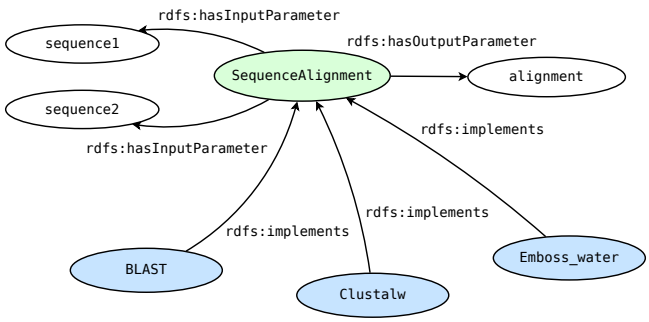


Figure 5.4: Semantic Function Templates act like interface classes in object orientation. They provide a template for which many different implementations can exist.

The fulcrum of NOP concept is semantics. It is only through semantics the processes are linked together at a higher level. Over the years quite a few semantic standards have been proposed mainly aimed at semantically annotating web services such as OWL-S, SAWSDL, etc. Most of these semantic description embody the concept of process with

⟨statement⟩→**WANT TO** ⟨processes_exp⟩ [⟨where_clause⟩] [⟨invoke_clause⟩]
⟨processes_exp⟩→⟨process_exp⟩ [**AND**|**THEN**|**OR**⟨processes_exp⟩]
⟨process_exp⟩→⟨function_domain⟩[⟨as_clause⟩]
⟨as_clause⟩→**AS**⟨id⟩
⟨where_clause⟩→**WHERE** { ⟨conditions⟩ }
⟨condition⟩→⟨left_operand⟩⟨operator⟩⟨right_operand⟩
⟨left_operand⟩→⟨function_domain⟩|⟨id⟩**.**⟨attribute⟩
⟨operator⟩→[**NOT**]**IS**|**PREFERRED**
⟨right_operand⟩→⟨value⟩
⟨invoke_clause⟩→**INVOKE WITH** { ⟨invoke_list⟩ }
⟨invoke_list⟩→⟨invoke_item⟩ [{**,** ⟨invoke_list⟩ }]
⟨invoke_item⟩→⟨sqarql⟩|⟨url⟩ **FOR** ⟨process_input⟩
⟨process_input⟩→⟨function_domain⟩|⟨id⟩**.**⟨input_predicate⟩**.**⟨attribute⟩
⟨sparql⟩→**SPARQL** { ⟨sparql_select⟩ }

Listing 5.1: A simple SQL-like declarative language (dubbed TReQL) aimed to specify the minimum amount of steps in a workflow.

inputs and outputs and thus are all candidates for a NOP network. The depicted graphs (Figures 5.2, 5.1)above use BioMoby semantics but our proposed framework is not limited to using any one semantics description. The aim is not to reinvent yet another semantic description language but to work with what is already developed. With the aid of semantic adapters RDF graphs can be built which in essence would represent the NOP graph. Although semantic description languages are in abundance, some conventions need to be followed to realize a NOP network which boils down to correctly describing the inputs and output using ontologies. For example, in SADI framework [101] an operation named named *getEcGeneComponentPartsRat* has as input *KEGG_ec_Record* while the output type is *getEcGeneComponentPartsRat_Output* which, if taken literally, does not make sense for a NOP network as such outputs are rarely inputs to other processes. But the output points to an OWL ontology which describes the type as a collection of *KEGG_ec_Record*. Thus, looking deeper into the ontologies of inputs and outputs we can make better reasoning about process interactions.

In addition to input/output semantics would be semantics about the process itself, most importantly what function domain does the process belongs to. Our experience has shown that within scientific workflows only a few tasks are core-logic processes while most are data-tool processes. Furthermore, the core logic can often be categorized into higher function domains where each domain has similar input/output patterns. For example, a *Se-*

quenceAlignment function domain may have many implementations of the alignment algorithm but all follow a similar pattern of having one or more sequence inputs and produce and alignment output. In BioCatalogue [104], service are categorized using tags but using ontologies to describe such domains would be more powerful. We refer to function domains as Semantic Function Templates (SFT). A SFT is akin to an interface class in object orientation (see Figure 5.4). SFTs describe the high level structure of processes.

5.3.2 Network Reasoning

RDF graphs are well suited for reasoning. By reasoning we mean that learning implicit information on how processes are related by following edges. Since the NOP graphs are already in RDF triple stores, SPARQL combined with OWL reasoners can be used to build workflows from implicit relationships in the NOP graphs. For example, from the above graph if a user wants to get a phyogenetic tree from fasta sequence alignments then a reasoner can follow RDF triples to construct the graph in Figure 5.5.

Combining semantics with reasoning, workflows can be constructed with minimum knowledge of the processes involved thus the main interface to create and run workflows would be SPARQL endpoints that query and reason about the network. SPARQL is a low-level language for RDF graphs, one can consider it as the meta language for RDF on top of which other simpler declarative languages can be built that can be simpler to use. Such a higher-level query language is TReQL (Type Reasoning Query Language), aimed at capturing the users' core logic which can be synthesized to SPARQL. In TReQL the user specifies the major core processes such as a *SequenceAlignment* and their order in the experiment, the input data type and optionally any restrictions such as output datatypes, sequence algorithms, etc.

In TReQL processes form part of a function domain. Function domains are ontologies describing a semantic function template for similar functions (see Section 5.3.1). Thus conceptually many processes can be interchanged with some effort in handling data type conversions. In essence, a SFT adorns a process with a type so we not only have data types but also process types. These two in combination provide a powerful means of reasoning about a NOP network. In a typical scenario, a user wants to perform a sequence alignment, he/she knows the input data type as *fasta* so in TReQL it would translate to *WANT TO spec:SequenceAlign AS sq WHERE sq.input IS spec:Fasta.* This will query the NOP network for processes of type *spec:SequenceAlign* and have as inputs *spec:Fasta.* If a process is found but does not have the same input types, the reasoner will walk the network links to find if any process can transform the data type into the one required by the process.

The simplified grammar presented in Listing 5.1 shows the main constructs of the language. The user specifies the main processes to perform with the *WANT TO* statement

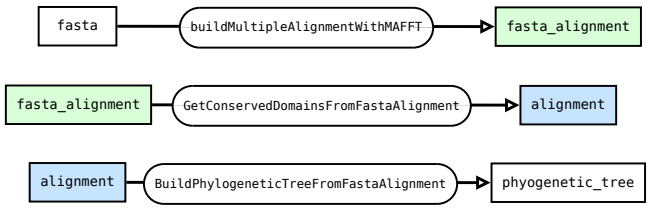


Figure 5.5: A path found by following network production rules.

and in which order. The *AND, THEN, OR* define the order of execution. *AND* means that the two processes are in a parallel construct, *THEN* means they are ins sequence and *OR* means that any of the processes will do. The inputs and outputs are stated in the *WHERE* clause. This allows the user to limit the search to parts of the graph that accept the input and output types. The operator *IS* matches an exact condition while *PREFERRED* will try to match the value. The *NOT* negates aforementioned operators. An optional *INVOKE WITH* can be used to attempt to start the networked processes. This clause points to the input data needed by initial processes. The data can itself be a sourced from LOD as a SPARQL query or through URLs clause. Using TReQL the graph depicted in Figure 5.5 can be generated by the query in Listing 5.2.

The query will output a list of paths that can satisfy the request. An ambiguity arises when multiple paths are returned to this end the user will be responsible for choosing between paths or refining the search. The chosen path is then embellished with the necessary parameters to make it executable and passed to execution engines. A common obstacle in interoperability between processes is data mismatch which can either be at the data format level or at the semantic level. There is no golden solution to magically solve these obstacles but some effort can be made to ease the problem. At the data level a common approach is to implement adapters that transform the data into desirable formats. Another complementary approach is to bridge the mismatch at the semantic level through ontologies. Solving such mismatches will help in linking multiple process databases.

```
77 WANT TO spec:SequenceAlign AS sq THEN spec:SequenceDisplay AS sd
78 WHERE { sq.hasInputParameter.objectType IS spec:Fasta }
79 INVOKE WITH {
80   URL fasta_url FOR sq.hasInputParameter.sequence1,
81   URL fasta_url FOR sq.hasInputParameter.sequence2
82 }
```

Listing 5.2: TReQL statement for Figure 5.5.

5.3.3 Process Object Identifier

Process Object Identifier (POI) (see Figure 5.6) is a feature that deals with making processes easily distributable, replicable and better in data provenance. Data can be tagged with the POI of the process that generated it. Using a POI system the correct process that acted on the data can be retrieved. This offers a better mechanism than traditional URLs to point to web services since services behind URLs can change thus invalidating the data provenance.

A service hosted at a URL and addressed by the same URL is anchored to that location and dynamic replication is done within the domain. One disadvantage is that if we want to dynamically replicate the service on a global distributed architecture then there is no easy way how to address the service farm. With a POI a service is addressable with a location agnostic address. Resolving the POI will result in a list of of hosted service endpoints. This is analogous to Internet DNS, websites are addresses using their domain name and not IP, hosting service can change but the domain name always resolves to the correct hosting machine. The POI system uproots services from their fixed location and allows for more dynamic handling as would be the case in dynamically scaling services to perform a parameter sweep studies.

POIs also help in provenance data. Tagging a data object as being modified or generated by a URL is not saying much since the service behind the URL can change completely or even cease to exist. A POI points to a specific process even if archived and not actively hosted. Through POI resolution, a list of identical services or processes can be looked up. This allows a system to, for example, distribute workload to these replicas. The intention of POI is to point to immutable processes thus when a new version of a service is available a new POI is assigned to it. This distinction between different versions of the same process is paramount for provenance capture as generated data can be tagged with the process POI.

5.3.4 Process Containers

In the NOP framework any object that is able to compute is a process and an effort is made to make everything linkable. The framework is not specifically bound to web services although these are the simplest form of linkable processes. The framework relies on a system of containers for exposing as many resources as possible. Traditional web services are already hosted within containers such as Axis2 or Tomcat. The same technique can be applied to generic processes. As part of ongoing research in process containers optimized for scientific computing, we have implemented two types of containers; a modified Axis2 container for running traditional services on deep network machines such as grid machines which have restricted Internet access. This technique which is described in detail in chapter 3 relies on message passing through a message exchange. The Axis2 container was

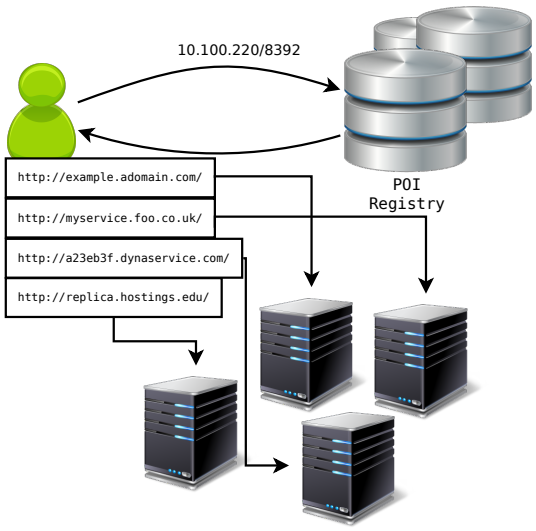


Figure 5.6: Process Object Identifiers (POI) enable unique identification of many identical processes hosted at different locations.

modified so data invocation requests are queued on message queues which are picked up by the Axis2 container and pushed up to the service. The output is done in reverse thus the container outputs the return result to the message queue. This messaging interface allows processes to communicate directly with each other since containers are made to listen on specific queues while a message routers within the message exchange is responsible to move messages from one process to the next. Messaging servers such as activeMQ implement the AMQP open protocol thus any container able to use AMQP can communicate with the rest of the processes. Messaging is a common pattern to achieve distributed computing as is done with MPI (Message Passing Interface) on dedicated computing clusters.

Another implemented container is a Python container (used in PUMPKIN) which implements a more workflow oriented process. The Python container uses the same AMQP to communicate externally thus there is no distinction between an Axis2 service and a Python task and thus interoperability is as straightforward as pushing messages through the message exchange. The Python container implements the concept of ports as opposed to parameters as is done in web services. Ports are typed data channels which tasks use to communicate. A Python task implements a *run()* function which is called when the task is loaded into the container. The task will then listen for data to process on the message queue. This container mechanism makes it possible for replicating processes since containers are able to host many different processes.

Figure 5.7 depicts the architecture of an optimized process container. Using AMQP, the container can listen for input data on message queues. This allows a container to execute

on restrictive networks such as grids or desktops. A container can host many processes by dynamically deploying processes on demand. Data handling is done by the container using the data handler which can achieve peer-to-peer communication for containers that are reachable such as on clusters. A provenance component sends provenance data to RDF stores.

Process containers also make it possible to distribute and replicate processes anywhere containers are running. This coupled with POIs has the ability to distribute processes globally. This method of containers makes resource acquisition much simpler. For example, in a cluster/grid architecture the container is submitted as a pilot job which then can run web services or Python tasks; in cloud infrastructures, templates are primed with a container so that when the VM becomes active the container can start hosting processes. Containers also find their way into unorthodox computing resources as is in web browsers [17]. While containers vary greatly in what type of processes they host so as to increase the computing resource outreach, semantically, the processes are indistinguishable and thus a NOP network can traverse many resources types.

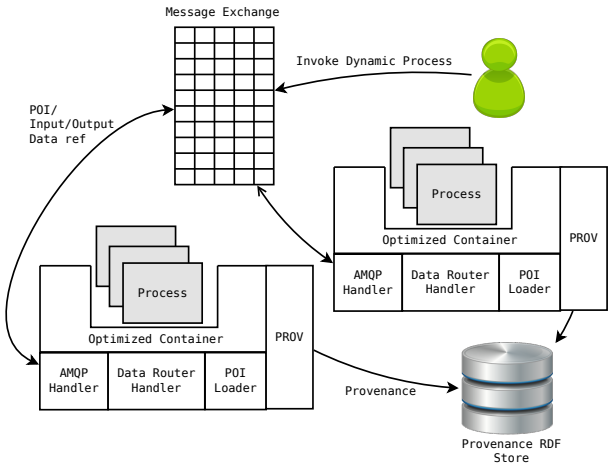


Figure 5.7: Python Optimized Process Container. The container is intended to run task oriented scientific applications on a variety of resources including resources with limited Internet connectivity such as cluster nodes and private clouds. The container can run any Python task and handles all communication in a *pull* fashion thus, allowing tasks to be invoked behind firewalls.

5.3.5 Usage Scenario

Figure 5.8 combines all the described framework components into a typical usage scenario. In (1) a user submits a TReQL query such as described in Listing 5.2. This query is

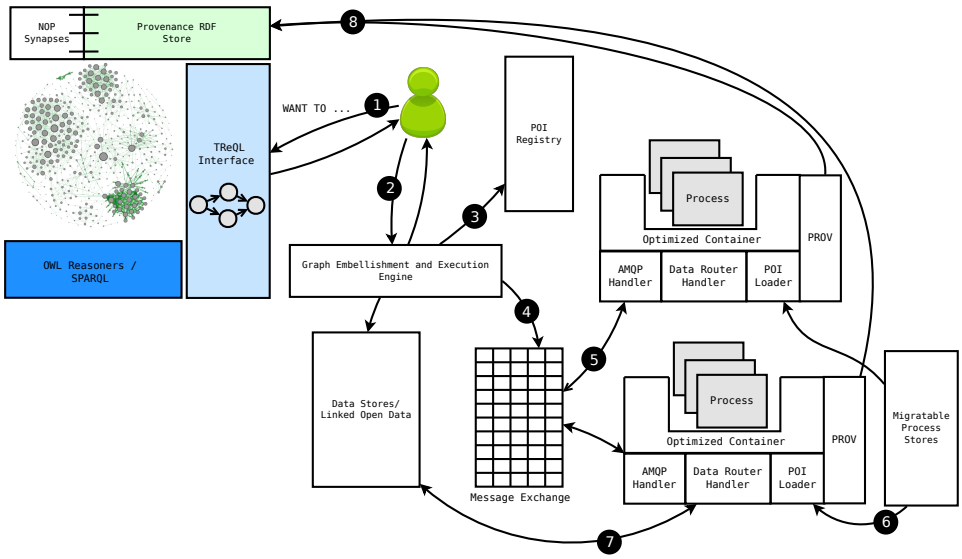


Figure 5.8: Diagram combining all components into a framework and presenting sequence of actions of an usage scenario (detailed description in the text).

decomposed into SPARQL queries and run against distributed RDF triple stores which describe the Networked Open Processes. At this point the user is presented with paths which need not all be connected. In the case that a desirable path is selected, the path with the *INVOKE* part of the TReQL is passed to the Graph Execution Engine in (2) where all necessary dependencies are included to make the graph executable. The Graph Engine is responsible to resolve POI for processes in (3). This will result in list of identical processes (if any) and the user can opt to use all and partition the input data amongst the farm of identical processes. If the process in question is not active on any of the containers, a reference to its archive (migratable process stores) is retrieved. In (4) the engine invokes the processes (the scenario assumes using optimized containers) by publishing a request for the deployment of the process onto one or more containers. If the process is active then the engine proceeds to submit the input data to the message queues on which the processes are listening. The AMQP handler in (5) retrieves input data or process deployment requests and either invokes the process or retrieves the archived process (6) and deploys it then proceeds to invoke it. The optimized containers are responsible to retrieve data through the data handler (7). This relieves any other framework component from handling data which could potentially cause bottlenecks. The optimized containers also have the possibility of publishing provenance data back (8) into linked open data. This provenance feedback loop is used in NOP to perform network optimization. This stage emphasizes edges in the NOP network so that frequently used links will be preferred in future TReQL searches.

5.4 Summary

Our investigation shows that the level of semantics on the Internet is reaching a level where processes or services can be linked together through their common data transformations. This semantic knowledge represents a top layer in our scheme (Figure 1.2) and would constitute the primary as the primary route finder in the underlying data transformation network. In light of the ever increasing scientific services/processes, we have shown how future distributed and collaborative scientific computing can be done through the new concept of Networked Open Processes NOP. We described a framework for tackling a complex NOP network. In this chapter we introduced a number of new concepts including the concept of Process Object Identifiers (see section 5.3.3), TReQL, a minimalistic SQL-like language for specifying main experiment processes (see Section 5.3.2), Semantic Function Templates which are templates for core scientific processes (see Section 5.3.1), and Optimized scientific process containers which increase the computing resource outreach (Section 5.3.4). Although a full reference implementation of the framework is not available yet, optimized containers have already been investigated and shown to be successful in uprooting services to exploit grid resources [40]. In Chapter 4 we have also shown how a protocol can be implemented which would aid the decentralized data processing and communication. The automata model described in Chapter 4 also lays the foundation for semantic transformation. In our model we talk about state transformation where states are described as tags. Using references to OWL classes in place of tags, one can envision data transformations with semantics.

CHAPTER 6

EVALUATION OF DATA PROCESSING MODELS

In chapters 3, 4, 5 we have studied scaling, moddelling and usage of semantics in the context of distributed data processing. Based on this research we have built adequate experimental prototypes to validate the elaborated models. In this chapter we present experimental results of various models discussed in the course of this thesis. Specifically we demonstrate results for the predication-based scaling discussed in chapter 3, section 3.3, fuzzy-based scaling and load balancing discussed in chapter 3, section 3.4, the WFaaS-based task farming discussed in chapter 3, section 3.2, and two applications for the automata-based modelling approach presented in chapter 4. The results presented in this chapter formed the bases of the publications listed in chapters 3, 4, and 5.

6.1 Prediction-based Auto Scaling

In this section we demonstrate the prediction-based scaling approach using the dataflow model presented in chapter 3. The method is applied to a image processing workflow. Figure 6.1 shows an example application using Octave[1] [110] aimed at illustrating the task level scaling implemented in the Dataflow engine. The workflow has two starting points one being the *DirectoryReader* which, as the name suggests, reads images from a directory. The second starting task is the *Parameter* task. This task acts as a parametric engine which supplies parameters to the *Histogram* task. The *Normalize* task normalises RGB images. The tasks, *Converter2* and *Converter1* convert images into different colour spaces. *Histogram* calculates the euclidean distance between the two new colour space histograms. At the end of the workflow the results are collected by *Results* while intermediate images are collected by *ImageCollector*. The core tasks of the workflow i.e. *Normalize*, *Converter2*, *Converter1*, and *Histogram* are said to be embarrassingly parallel in nature. These tasks have no casual dependency between messages on the same port and therefore are ideal for a driving test case to test our Dataflow and scaling systems.

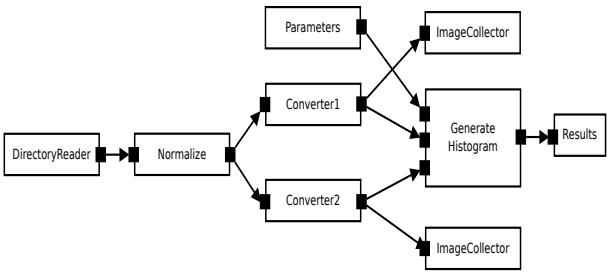


Figure 6.1: An Octave image processing workflow. The workflow converts images into two different colour spaces and calculates the histogram difference between the two new colour spaces. The parameter setting for the Histogram is the histogram bin size.

As a resource pool back-end we had access to the Distributed ASCI SuperComputer 3 (DAS3) [111] which is a five wide area distributed system. Tasks where submitted across all clusters. Each site hosts a GridFtp server which were used to transmit data between tasks and therefore allow inter-cluster task communication. The timings illustrated in Figure 6.2, Figure 6.3 show the execution time of each task in the workflow. The execution time incorporates the the scientific logic execution time as well as overheads associated with communication, startup and clean up times.

Converter2 and *Converter1* are set to auto-scaling. In these cases the Dataflow engine is responsible for gauging the load on the designated data partition input queue and decide

[1]Octave is an open source Matlab implementation.

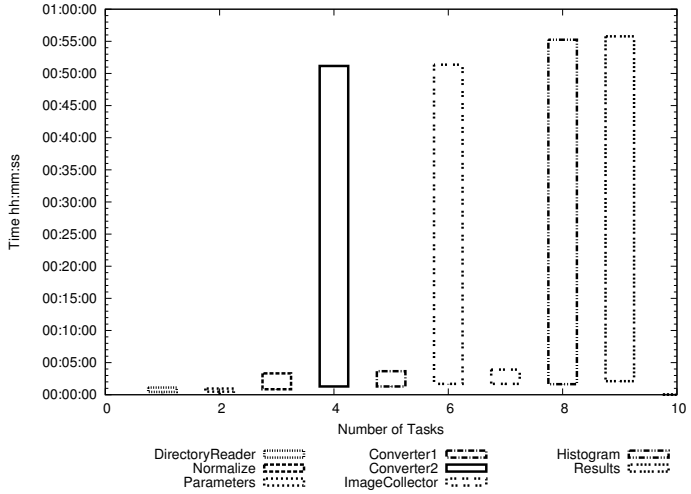


Figure 6.2: Workflow execution without scaling. The length of the bar represents the total execution time.

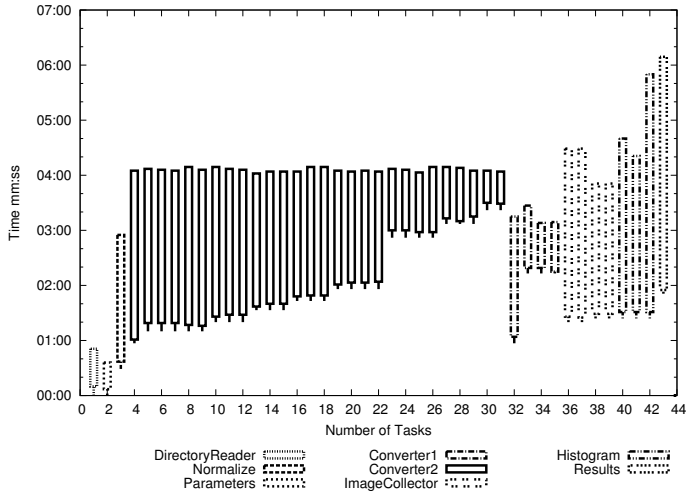


Figure 6.3: Workflow execution with scaling. Lines preceding the bars represent waiting time while the length of the bars represent actual execution time of an instance of the workflow task. Bars with the same line encoding are replicated instances of the same task. Time 0 represents the start waiting time for the first task.

on how many clones to submit using a user defined threshold. The *Parameter* task acts as the parameter engine by reading parameters from a file and sending messages containing parameters to the *Histogram*. The latter is not set to auto-scale but instead is scaled on a per parameter bases. Thus each parameter from *Parameter* creates a new instance of

Histogram. *ImageCollector* is set to a fixed replication where the user specifies the number of clones.

At time 0 *DirectoryReader* and *Parameters* are submitted as these have no dependencies. Other tasks are only submitted when some data is available for input. This is clearly shown by the difference in starting times for each task in figure 6.3. The computation overlap between tasks show the effect of message pipelining where tasks can start processing data immediately as it is produced and need not to wait for the dependant tasks to terminate before starting execution.

Figure 6.2 shows the workflow execution with scaling disabled. The results clearly show that *Converter2* is relatively slow to process the data and hence causes a flow bottleneck. This has a ripple effect on the dependant tasks (*Histogram*, *ImageCollector* and *Results*) which spend most of their time in an idling state waiting for new messages to be delivered to their input ports. Since scaling is completely disabled, the parameter sweep scenario where the *Parameters* is the task parameter engine does not take effect and hence *Histogram* is not replicated. This results in The *Histogram* task processing all parameters. The mean runtime for the non-scaled workflow is around 54 minutes.

Figure 6.3 shows the same example with the same inputs but this time enabling scaling features. The results immediately show how the previous bottleneck was circumvented through replication. The *Converter2* was replicated as many times as needed hence increasing the data consumption and production. The *Histogram* is replicated 3 times which follows the parameter sweep scenario whilst we have four *ImageCollectors* as defined by the user. All in all the 8 task workflow unfolded into 44 separate tasks through scaling. In this example the effect of just auto-scaling achieved a 9 fold improvement over the non scaled workflow. The single task *Converter2* achieves a much better improvement which is approximately 16 times faster. The execution profile for *Converter2* tasks also shows the burst threshold in action since tasks are replicated in bursts which gives rise to the staircase profile. From figure 6.3, *Converter1* is also dynamically scaled up but since its faster it has a lower replication count.

6.2 Fuzzy-based Auto Scaling

In this section we demonstrate fuzzy-based scaling and load balancing which were presented in chapter 3. The methods are applied to a workflow of web services.

The workflow depicted in Figure 6.4 illustrates a typical bio-inforamtics workflow. The workflow consists of two independent pipelines. The pipelines compute sequence alignment using data supplied by the UniProtKB [112]. Each component is a SOAP Axis2 web service. The source represents the bootstrapping component while the sink represents the result gathering client. The workflow is induced with 22550 alignments for each pipeline

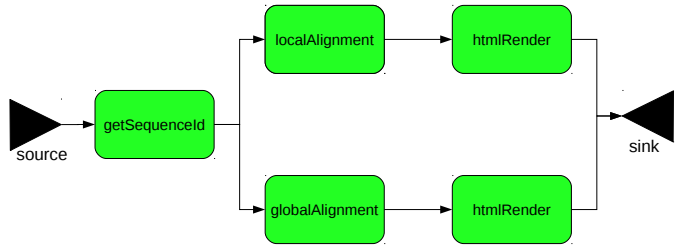


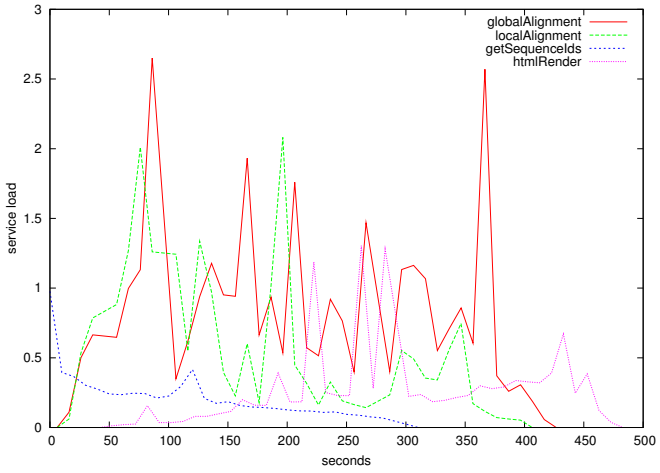
Figure 6.4: A web service workflow using BioJava to implement sequence alignments.

therefore the whole workflow computes 45100 alignments. The `getSequenceIds` web service reads a list of sequence ids and returns the actual sequence data for the ids. `localAlignment` performs a local alignment on the passed sequences while `globalAlignment` performs a global alignment. Both alignment web services use the BioJava API [113] for processing the biological data. `htmlRender` transforms the results into HTML tags which are then made accessible through a web browser. The sink concatenates the results into HTML pages.

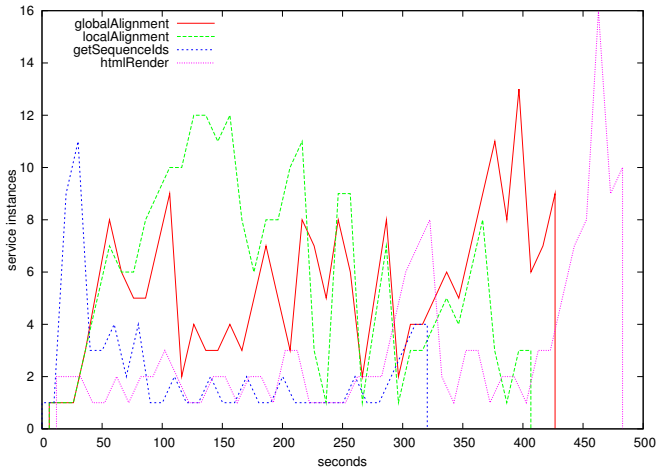
As a resource pool back-end we had access to the Distributed ASCI SuperComputer 3 (DAS3) which is five wide area distributed system. For purpose of testing the resource competitiveness between web services we used a single 29 node cluster from the University of Amsterdam (UvA). The UvA cluster nodes each have 2 2.2GHz AMD Opteron DP275 processors with 4GB of main memory.

The results in Figure 6.5(a) and Figure6.5(b) illustrate the execution pattern of the workflow in Figure 6.4. Figure 6.5(a) shows predicted input load for each web service during the execution lifetime at intervals of 5 seconds. Spikes in the load graph signify when a considerable amount of data has been queued on the web service input queues. The spikes in the service load are short lived since the fuzzy controller immediately responds by initiating multiple instances to deal with the increased load. The response to the service load spikes is illustrated in Figure 6.5(b) which shows the number of web service instances simultaneously running at any particular time. Thus spikes in the service-load graph 6.5(a) are shortly followed by spikes in the service-instances graph 6.5(b).

Dissecting some notable regions within these results we can note that at the beginning of the execution `getSequenceIds` starts with a load close to 1. Since no other web service is running at this stage, the fuzzy controller does not waist time and aggressively scales the service up. This can be noted with a spike in 6.5(b). With the autonomous orchestration feature, as soon as `getSequenceIds` produces output it also initiates its dependent successors. Since `getSequenceIds` produces output for both `localAlignment` and `globalAlignment`, the multiple instances immediately increase the input load on both these web services. The spikes for the simultaneous load increase is illustrated between the



(a) Web service load for all workflow components. A service load of 1 means that the web service is expected to complete its task within the specified time, service load of 2 means it will take twice as much to complete.



(b) Number of web service instances running for each workflow module at any given time.

Figure 6.5: Results showing the calculated service load 6.5(a) and the number of web service instances initialized by the fuzzy controller 6.5(b) to control the service load.

50-100 second mark in 6.5(a). As expected, the fuzzy controllers take action and respond by replicating the instances. At this point the controller on `getSequenceIds` senses the increase in resource load and also notes its own load has diminished hence it downscale itself to make way for other services. Whilst still having a light load, `getSequenceIds` will tentatively replicate itself slowly when it detects dips in resource usage. This can be

noted in the region 100-200 seconds in 6.5(b) where sudden dips by `globalAlignment` result in slow increase by both `getSequenceIds` and `htmlRender` simultaneously.

As was the case for `getSequenceIds` at the start of execution, a relative small spike (between 400 and 450 mark)in the load for `htmlRender` at the end of execution triggers an aggressive replication since it is the only running web service at that time. These results show that the workflow of cooperating web services cooperate on three fronts: cooperation through communication, cooperation through orchestration, and cooperation through fair resource usage. During the whole execution, the load on the resources was at an average of 72%. This is very close to the ideal with regards to the fuzzy controller configuration where 75% had the highest probability in the *normal* membership function.

6.3 WFaaS-based Task Farming

In this section we demonstrate the WFaaS-based approach to task farming as presented in chapter 3. The WFaaS-based approach to task farming was applied to a biomedical study for which 3000 runs were required to perform a global sensitivity analysis of a blood pressure wave propagation in arteries (Figure 6.6). Patient-specific simulations involves many parameters based on data measured in-vivo and subject to uncertainties [114]. The relationship between the model parameters and the simulated output is complex. Thus, a global sensitivity analysis is an appropriate method to investigate how the uncertainties in the model output can be attributed to the different sources of uncertainty in the model input. A patient specific model was set up for the major arteries of the arm. In a Monte-Carlo study, 11 model parameters (e.g. Young's modulus, vessel diameter, artery length) were varied randomly within their respective uncertainty ranges over 3000 model runs.

The primary goal of running the use case was to speed up the entire sets of farmed workflow; this speed up is limited by Virtual Organization (VO) membership of the users: the more this VO membership gives access to computing resource the more the system will be able to farm concurrent jobs and the faster is the execution of the entire experiment. In the current setting the workflows are farmed by groups of fixed size; if more resources become available new groups can be farmed. The results where obtained on the Dutch ASCII supercomputer (http://www.cs.vu.nl/das3/). Figure 6.7 shows the execution and waiting times obtained using two scheduling approaches: WFaaS (left), and the original WS-VLAM farming (right). Each workflow submitted using WFaaS approach performs multiple simulations, any computing resources that become available is added to the pool of resources to process the remaining simulations, in total 28 workflows performed all the 100 targeted simulation. While in the original WS-VLAM farming each submitted workflow performs only one simulation, which lead to 100 separate submissions. It is clear that when computing resources are limited and multiple applications are competing

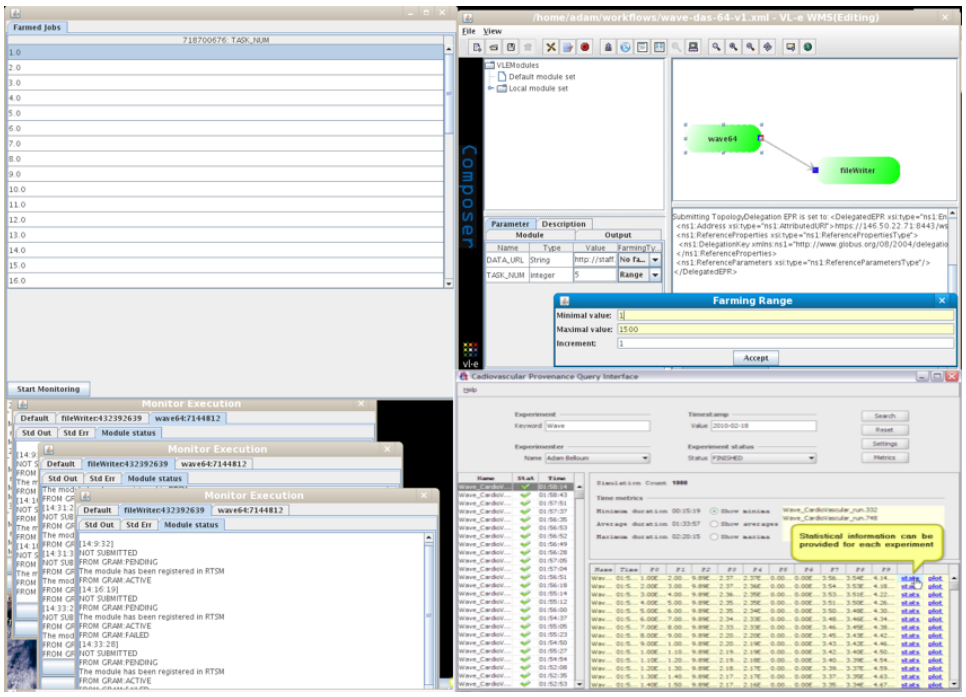


Figure 6.6: Screenshot of the scientists' desktop. On the top right corner, the scientist can compose the workflow in intuitive way. The user can also specify how he/she wants to farm his workflow (list of input files, a range of application parameter). When the user executes the workflow a monitoring window (window on the top left corner) shows the farmed workflows and the user can monitor each run separately (cascade of windows on left bottom corner).

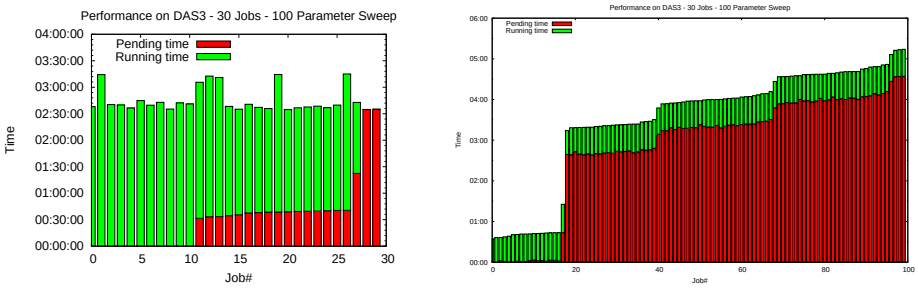


Figure 6.7: Performance of WFaaS. On left: 100 simulations of the wave workflow takes about 3h:15mn using the WFaaS, on right: the same number of simulations take 5h:15mn farming each workflow separately. Each workflow submitted by following the WFaaS approach performs multiple simulations which reduce considerably the waiting time. In both cases workflows are competing to use 28 computing nodes. For the WFaaS example 30 tasks where submitted thus the last 2 tasks are stuck on waiting queues. Once the slots are freed they terminate immediately since the other tasks performed the work using WFaaS.

to get access to these resources, the WFaaS approach has a significant advantage as it reduces the waiting time considerably, leading to the overall speed up of the entire set of farmed workflows. The overhead of the WS-VLAM in term of data movement among the workflow component is low and has been discussed in [61].

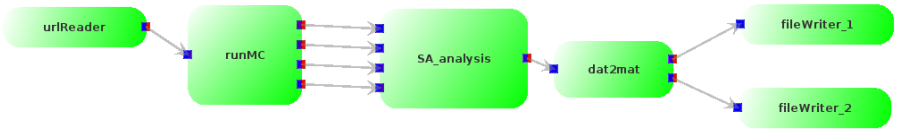


Figure 6.8: A skeletal workflow intended to simulate blood flow for treatment of thrombosis. This is an extension of the first experiment and is intended to run patient specific blood flow simulation using many input parameter variations [115].

A second example (Figure 6.8) aims at demonstrating the WFaaS at the task-level. This experiment is an extension of the previous example where the core module *SA_analysis* contains the logic of the previous example. In this experiment, the tasks are given short workloads to simulate short lived tasks. This workflow is executed on the BigGrid (dutch eScience grid). With short lived tasks, resources are occupied for a relatively small time span making the queue waiting more apparent which cause noticeable overhead. The experiment was run on the BigGrid (http://www.biggrid.nl/) which is the Dutch grid initia-

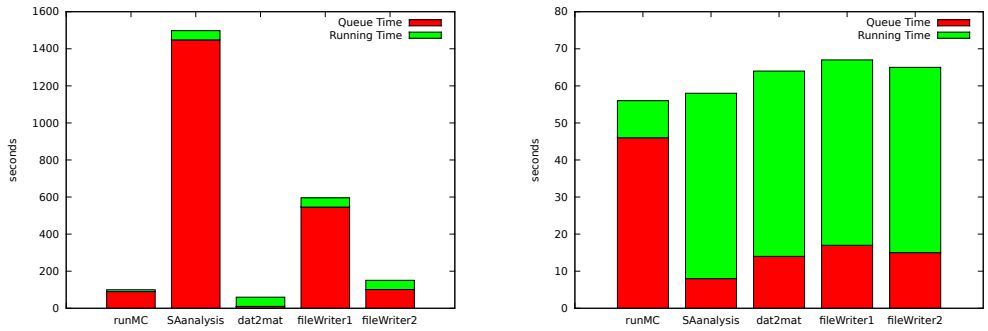


Figure 6.9: Workflow as a Service at task-level. On left: traditional workflow submission with unpredictable and, potentially, large queue waiting times. On right: Workflow harness jobs act as services by running multiple workflow jobs thus circumventing queue waiting times. The apparent queue waiting time on the right is due to the harness job polling interval.

tive.With task-level WFaaS we can instruct a harness to load a sequence of tasks instead of just loading one task and terminate immediately. This mechanism makes most use of the resource allocated to the workflow. The results in Figure 6.9 show the difference of the workflow run with task-level service oriented characteristics (right) and without (left). On the left shows the typical scenario with grid queue waiting times; queue waiting times are unpredictable and can vary from seconds to hours. Queue waiting times in shared grid resources play a crucial role in overall performance of workflow execution since they can span from minutes to hours [116]. This tend to cause problems in data intensive workflow execution because data produced by a task, for example *runMC*, needs to be buffered until the consuming tasks (*SA_analysis*) come into play. The data being produced might be too large to be buffered thus the faster the consumers comes start executing, the better the system can handle the load. This is achieved through task-level WFaaS (Figure 6.9 (right)) where acquired resources are reused to execute multiple tasks in sequence. The first task *runMC* has the highest queue waiting time since it has to pass the submission queues. Other tasks can be loaded immediately into already existing harnesses without waiting on any queue. This results in a reduced and predictable queue waiting time.

6.4 Automata-based Tweeter Filtering

A Twitter workflow is used as a demonstrating application for the distributed architecture using the automata model described in chapter 4. The choice of Twitter is that it provides

real world data loads while also providing fine grained data atomicity (a tweet message) thus creating a scenarios for pronounced overhead.

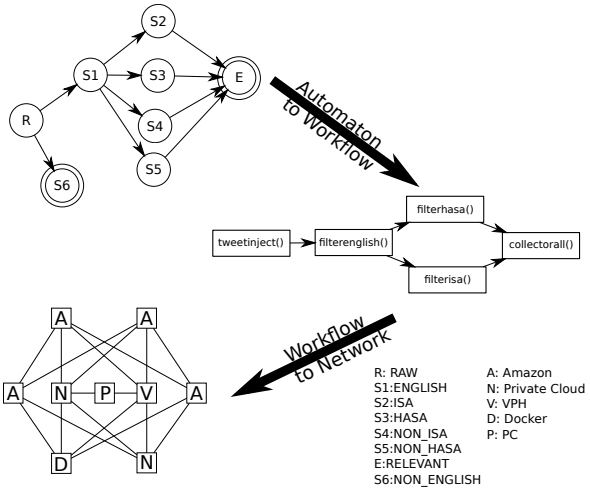


Figure 6.10: Top: Automaton for stream processing on Twitter data. Right: Function workflow for performing the transformations. Bottom: Nodes on EC2 (Ireland, Oregon), VPH (Krakow), local cloud (Amsterdam), Docker and laptop (Amsterdam).

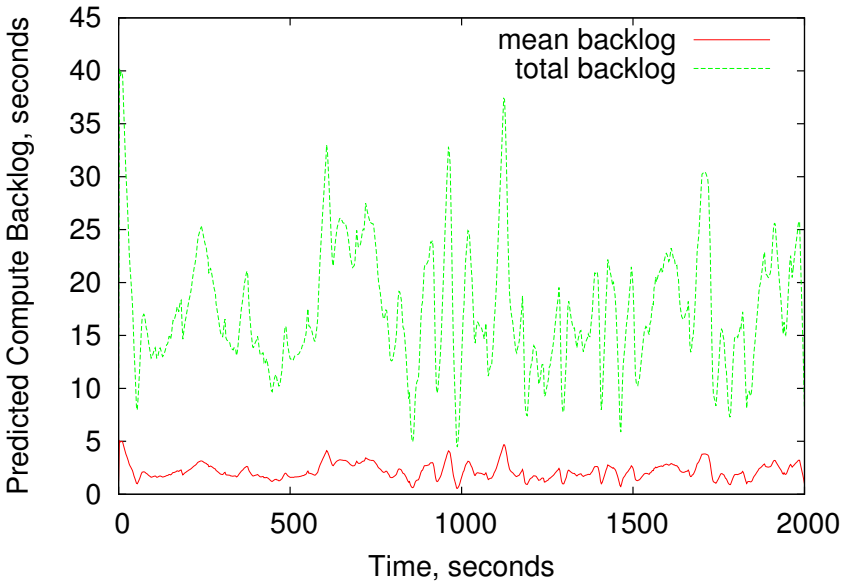


Figure 6.15: Aggregate and mean of predicted compute backlog on network over time.

The PUMPKIN testbed used for this application scenario is intended to demonstrate the applicability to globally distributed resources on controlled and uncontrolled resources.

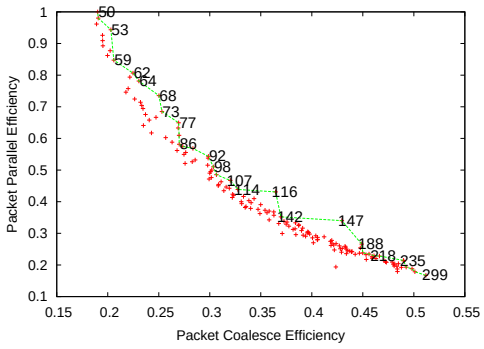


Figure 6.11: Packet coalescing processing efficiency, $ceff()$, between EC2 (Ireland) and private cloud (Amsterdam) vs parallel packet efficiency, $peff()$, assuming data window of 1000 tweets and 20 nodes. The numbers next to the data points are the actual number of packets in the grouping.

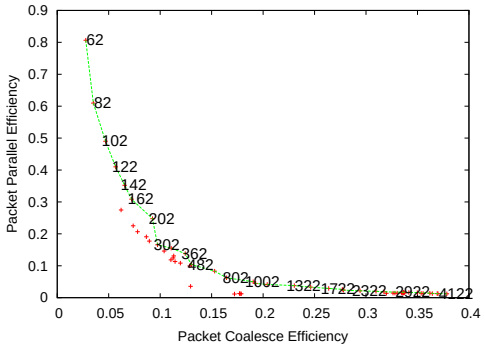


Figure 6.12: Packet coalescing processing efficiency, $ceff()$, between EC2 (Ireland) and EC2 (Oregon) vs parallel packet efficiency, $peff()$, assuming data window of 1000 tweets and 20 nodes. The numbers next to the data points are the actual number of packets in the grouping.

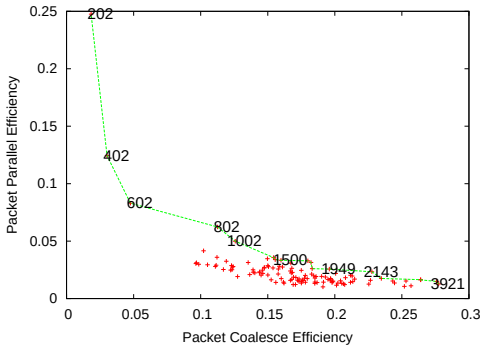


Figure 6.13: Packet coalescing processing efficiency, $ceff()$, between VPH-Share cloud (Cracow) and private cloud (Amsterdam) over RabbitMQ vs parallel packet efficiency, $peff()$, assuming data window of 1000 tweets and 20 nodes. The numbers next to the data points are the actual number of packets in the grouping.

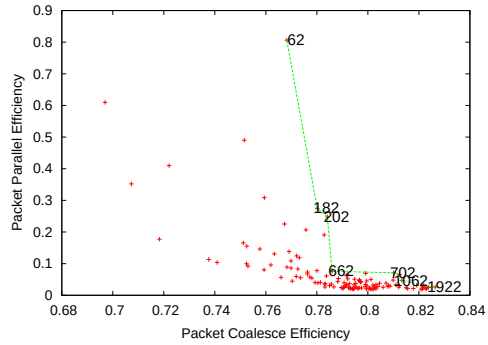


Figure 6.14: Packet coalescing processing efficiency, $ceff()$, between Docker VM and Docker host vs parallel packet efficiency, $peff()$, assuming data window of 1000 tweets and 20 nodes. The numbers next to the data points are the actual number of packets in the grouping.

Figure 6.10 illustrates the node network which included 3 nodes on EC2[2] (Ireland), 1 node

[2] http://aws.amazon.com/ec2

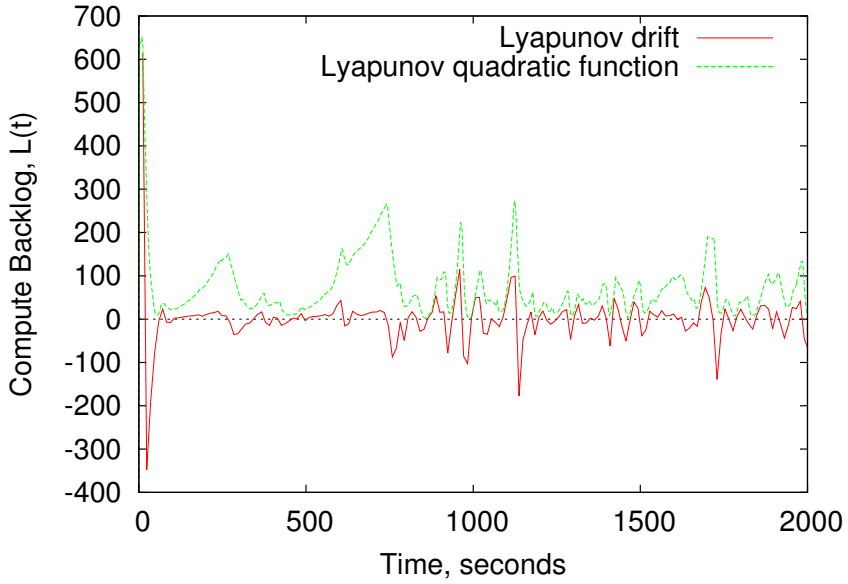


Figure 6.16: Maintaining system stability by minimizing backlog. Predicted compute backlog calculated using Lyapunov quadratic function, $\frac{1}{2}\sum_{i=1}^{n} Q_i(t)^2$ where $Q_i(t)$ is the backlog for node i at time slot t, and the Lyapunov drift.

on EC2 (Oregon), 1 node on VPH computing platform[3], 2 nodes on private cloud (Amsterdam), 1 Docker[4] node (Amsterdam), 1 PC (Amsterdam). Although EC2 nodes have public facing IP addresses the other nodes are all behind NATs and thus direct communication was not possible. In this case the nodes automatically use the RabbitMQ fallback.

The controller-less, dispersed and diverse resources are setup as a state machine using the automaton described in Figure 6.10. The code is deployed dynamically through the packet system described in Section 4.7.1 using the laptop node as the bootstrapping node. Twitter feed data[5] is also injected from the same node. The PC node in the network acts as an interface to the network and also as an optional compute node. The filters where given an additional synthetic load. This load aims at creating a backlog scenario where each successive processing node is slower than the previous. The loads for *tweetinject(), filterenglish(), filterisa(), filterhasa()* are 0, 0.1, 0.2, 0.3 respectively where the load signifies sleep time on every packet. In this scenario every Tweet is packaged as a data packet and given the initial state of *RAW*. These packets are injected into the network and traverse the

[3] `http://www.vph-share.eu`
[4] `http://www.docker.com`
[5] http://snap.stanford.edu/data/twitter7.html

appropriate nodes to end in a final state. Each state transition acts as a filter thus tagging the data along the way with a state tag. The first filter *filterenglish()* will tag the tweets with state $ENGLISH$ or state $NONENGLISH$. The $ENGLISH$ tagged data packets will move forward into the network while the other packets are dumped since a final state was reached. The last final state in Figure 6.10 is an extraction state to which relevant data transitions too. The node network achieves 3 levels of parallelism; through pipelining processing and communication overlap each other, *d-op filterisa()* and *filterhasa()* are intrinsically independent which means they run in parallel and the third level of parallelism is derived from data partitioning where multiple instances of *d-op filterenglish()*, *filterisa()* and *filterhasa()* split the streams between themselves.

Figures 6.11, 6.12, 6.13, 6.14 illustrate how the control flow described in Section 4.6 is able to adapt to the traffic traversing the node network in Figure 6.10. As is expected, the faster and more stable connections, the higher packet efficiency can be achieved at lower coalescing factor. Figure 6.14 shows in-memory communication between Docker[6] VM and host where high efficiency is achieved with low coalescing. The trade-off between parallel efficiency and coalesce efficiency produces a Pareto fronts. From Figure 6.14 between 60 and 100 tweets are enough to obtain adequate efficiency. On the other hand, Figure 6.12 shows that the range 60 to 100 provides an abysmal coalesce efficiency which means a higher coalescing number is needed.

In Figures 6.15 and 6.16 we demonstrate how we manage to control backlogs in the network using the formulation described in Section 4.6. Figure 6.15 shows both the aggregate predicted compute backlog over time which never goes over 40 seconds which without flow control, backlog would spiral out of control. The stark difference between the aggregate and mean boils down to the EC2 (Oregon) VM which had a slow, low bandwidth connection which resulted in more pronounced flow controller fluctuations. The Lyapunov drift in Figure 6.16 shows the effort of the flow controller maintaining a 0 drift progressively correcting spikes in predicted backlog.

6.5 Automata-based Tracking Brain Regions

A second application based on the automata-based data processing model is a medical workflow.

Figure 6.17 shows a medical workflow for analyzing brain regions [117] using MRI (Magnetic Resonance Imaging) and DTI (Diffusion Tensor Imaging). The characteristics of this workflow are that is it is a long running workflow with few messages passed between nodes. The automaton in the application represents patient data. In this case having

[6] www.docker.com

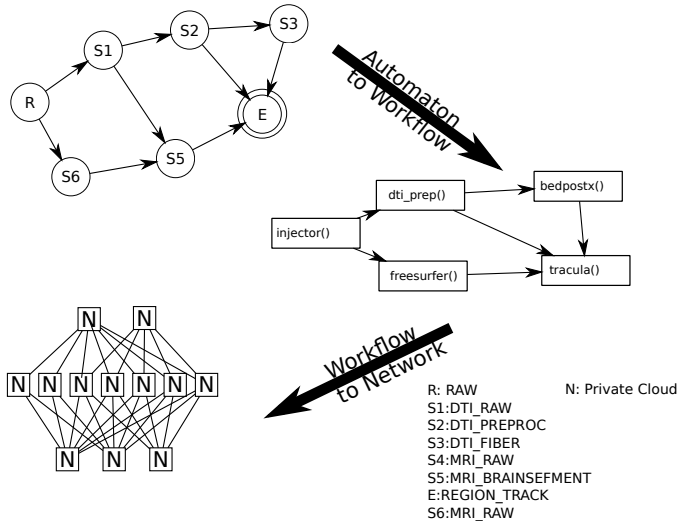


Figure 6.17: Tracula workflow for tracking fibers between brain regions. Top: tis the patient data automaton. Right: the workflow as a sequence of *d-op*. Bottom shows the VM network hosting various *d-op* on a private cloud.

a data processing structure represented as a data transition graph aids tracking patient processing data by knowing, at any time, in which state the data is. This is important since in such applications provenance is paramount. Provenance is captured intrinsically within the packet framework since each packet can carry the data and information from the previous transition. Listening on the network allows the collection on runtime packet data. The main functions in the workflow are distributed on different virtual machines.

Patients' data have an associated automaton which keeps track of data processing progress. The architecture implicitly allows for scaling up by adding more virtual machines hosting certain functions of the workflow. This replication of tasks allows multiple patient data graphs to be processed simultaneously.

A particular feature of this application is the merging of 3 states into the $BRAIN\_REGION\_TRACK$ (reconstructed of major white-matter pathways from diffusion-weighted MR images). Merging states in a distributed system is a non trivial procedure especially when replicating the merge node. In the application the *d-op tracula()* is responsible for merging 3 s-tags from 3 different nodes into 1. In the case where we only have 1 *tracula()* running, merging is a matter of waiting for all inputs. PUMPKIN allows multiple instances of the same *d-op* to run in which case one packet can be received by one *trcula()* instance while the other packet from a different node can be sent to other instances. To solve this, *d-op* can implement a *pre_run()* function. This function can reject packets based on some *d-op* specific logic. For example in the case of this application

different priority is given to the 3 different input states. If *tracula()* receives a packet with highest priority and is idle it will accept the packet and wait for the other two packets with lower priority and same ship id (for packet format see Section 4.5. Any other packet is rejected. Upon a rejection, PUMPKIN, will send the packets to other *tracula()* instances. The outcome is that low priority packets will hop from one instance to the next until the corresponding high priority packet is found on some node. Although this distributed merge routine works it is also quite expensive and thus should be avoided in streaming application scenarios.

6.6 Summary

Through the presented results we demonstrated that various approaches to scaling data processing are possible. Predication-based looks at the data charachteristics to gauge the state of the processing system, fuzzy-based takes this one step further by factoring in also the state of the resources and coordinates multiple competing and colleborating tasks on same resources. WFaaS-based approach to task farming demonstrates to effective use of resources by focusing of data partitioning and scheduling.

The automata approaches demonstrate that a data processing model can also be used as part of the concrete execution of tasks and an important component in a protocol aimed at data process routing. The results illustrate various techniques of flow control mechanisms based on data process performance and prediction.

This chapter brings to an end the methodology life-cycle as outlined in the objectives whereby we formulated models of solutions, designed artifacts that reflected the models, implemented artifacts and tested artifacts to validate the models. As the modeling phase in our methodology we presented a prediction-based, fuzzy-based, WFaaS, automata-based models. In the design phase we designed and implemented artifacts for various environments and technologies such grids, clouds, clusters and web services. The implementation of the artifacts was done in accordance to the studied models. Testing of the artifacts allowed us to validate the models as described in the previous chapters.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

Our research hypothesis was that given the increasing intricacy and volumes in data processing and the dynamism of infrastructures, **data-centric distributed computing should be tackled jointly from both the abstract data processing, semantic models and the infrastructure fronts so as to increase the knowledge and availability of data.** To tackle our hypothesis we set out four research objectives which were:-

1. To investigate new and emerging resources and resource compositions.
2. To investigate and propose new scaling techniques to fit contemporary data and resource set ups.
3. To study new models of data processing and its applicability to concrete data processing.
4. To study the role and implications of semantics in the data processing flora.

Through the course of this thesis we have investigated our research objectives by means of studies, designing/modeling, implementing and testing through demonstration of experimental results. In chapter 1 we have illustrated a taxonomy for data which was categorized on an availability-knowledge axis (Figure 1.1). The taxonomy acted as an underlying guide for each objective where the goal was to increase the data in any of the dimensions. Our work on the **first objective**, the investigation of new and emerging resources and resource compositions as a means to increase data availability (Figure 1.1), resulted in the addition of Internet browsers as a resource to perform scientific computing. To prove this we demonstrated how a cluster of such resources can be easily and globally setup (simply sharing a URL) and how a typical scientific application dealing with sequence alignments was used on this *new* resource. We also showed the potential in virtual infrastructures and how new compositions of infrastructure and SWMS-like middleware paves the way for an

inter-cloud system. Specifically we presented how a symbiosis between application and infrastructure can be achieved whereby infrastructure can mold around the application while we introduced the concept of a new generation of applications that expose new metrics pertaining to there data performance and network-ability.

Our work on the **second objective**, the investigation of new data processing scaling techniques, has resulted in various approaches that increase the knowledge capacity of data (Figure 1.1) through models and methods for data processing. A prediction-based auto-scaling approach that can be applied to data-centric workflows as a way to accelerate data processing rates within scientific workflows. The ability of scaling tasks independently enables replication of tasks to match the data production rate. This minimizes workflow bottlenecks and reduces total makespan. Through task harnessing we showed how scientific logic can be separated from underlying communication and data transport intricacies. We have shown that the WFaaS approach to task farming is a very promising approach for large parametric studies. The WFaaS paradigm at the data-level as well at the task-level reduces common scheduling overheads such as queue waiting times in shared distributed infrastructures and makes better use of the computing resources by making most of the allocated time slot given to each task. Another presented approach to scaling was autonomous scaling of web services using fuzzy controllers. Autonomous orchestration has been achieved with web services containers having myopic view on the workflow. The implementation of the architecture demonstrated that the above attributes to dynamic web service handling can be achieved in a non-intrusive manner thus not modifying the actual web service code. To make service more scale friendly we implemented a late binding mechanism and reversed communication pattern so that services can be *submitted* as jobs to traditional resources.

Within the scope of the **third objective**, the studying of new models of data processing, we presented a new data processing paradigm based on modeling data as automata. The model also lends itself to compose infrastructures as state machine networks. The bi-objective model increases data in both knowledge and availability dimensions (Figure 1.1) since it provides for modeling data processing (knowledge) and compute infrastructure (availability). The model takes a data-centric approach to describe abstract data processing as a sequence of state transitions. Through PUMPKIN implementation we showed how the automata provides information about data during the fluidity of processing which guides data to computing. The distributed decentralized architecture of PUMPKIN and the self-routable data packets creates a data processing plane where data processing is reduced to a protocol which enables clients to inter-operate. The usage of data packets as data processing parcels allows us to investigate added data routing attributes. In the presented, model data is routed based solely on its state. Additional attributes can be easily added to the packet such as energy and security which would allow packet schedulers to choose were

to send the data based on such attributes. We investigated several protocol control flow mechanisms pertaining to this model. We introduced the idea of packet-parallel efficiency and packet-coalesce efficiency which are based on predicting data processing performance. Using the same mechanism we demonstrated how queue backlogs can be managed.

Finally, the results of our **fourth objective**, to study the role and implications of semantics in the data processing flora as a means to further expand data knowledge (Figure 1.1) through processing, have shown that through semantics, processes can be linked together. To capitalize on this, we introduced a number of new concepts including the concept of process object identifiers, TReQL, a minimalistic SQL-like language for specifying main experiment processes, semantic function templates which are templates for core scientific processes, and optimized scientific process containers which increase the computing resource outreach.

Returning to our hypothesis, our first and second objectives aimed at tackling data processing from the infrastructure level while our third and forth objectives tackle distributed data processing from an abstract and semantic stance. The objectives were guided by the taxonomy (Figure 1.1) We have shown that these objectives work towards a common goal of data processing to increase data in both knowledge and availability dimensions of our data taxonomy (Figure 1.1). This confirms the hypothesis that data processing can add more value to data when tackled holistically from both fronts.

7.2 Vision and Future Work

Our contributions are a step towards intricate data processing on large scale distributed infrastructures where a novel model for data processing based on automata formalism is used to model data as a state system. Heterogeneous resources are assimilated through a common protocol which is the result of synthesizing the automata models into a data routing mechanism. We believe that the future of knowledge and data processing will be coupled into a feedback loop where processing of data produces knowledge which invokes more processing of data. Knowledge graphs have been shown to be successful (e.g. adding semantics to Google searches). Similarly we believe knowledge of data and its processing will be crucial for future data-based knowledge discovery. This means that such a semantic layer would need to be coupled with data processing. Our vision of combining various data processing dimensions is through a layered system (Figure 7.1). The layers are loosely coupled discrete plane of functionality revolving around data processing. Inter communication between layers would be done though interfaces.

The bottom layer in Figure 7.1 shows the physical resources in combination with the virtualized resources. These are represented as one layer but still need different treatment from higher layers. Layer 2, The immediate resource controllers of the are represented in

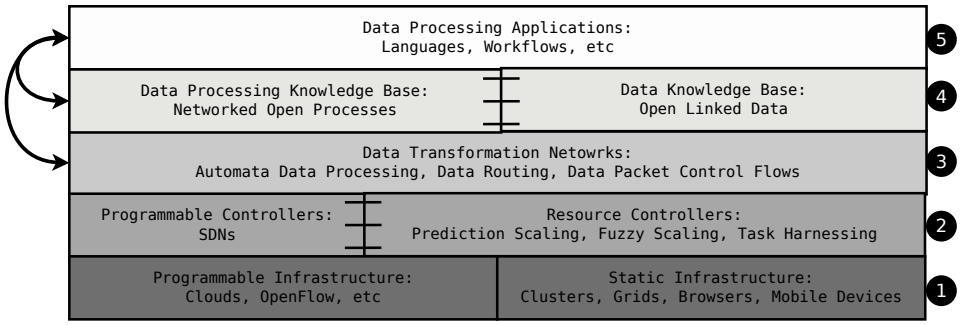


Figure 7.1: Envisioned layered system for future data processing incorporating contributions from our work. Layer 1: programmable and static resources, not all resources. Layer 2: the immediate controllers on top of the resources, programmable resources need additional controllers. Layer 3: data transformation networks such as an automata for data. Layer 4: relations between data, processes and states allows for reasoning such as inferencing at a higher level. Layer 5: the application layer where distributed networked applications such as distributed data-centric workflows are designed and executed based on the lower layers.

this layer. For the programmable part of the resources additional controllers are needed which implement management routines such as SDNs, interclouds, migration etc. Alongside these controllers, low level scheduling and scaling routines are also present in this layer. The techniques and models presented in chapter 3 are meant to take place at this level close to the resources. Managing this layer means acquisition of resources and optimizing various aspects by exploiting the underlying dynamism such as migration, re-routing, and scaling. Layer 2 is where SDNs are created. The potential of SDNs is the ability to reconfigure themselves but reconfiguring means that they need a goal to do so. This goal comes from the layer above which is layer 3. The first class citizen in layer 3 is data and its states. This layer implements models similar to the automata model described in chapter 4. In such a layer data is self-routable between processing nodes in a similar way to TCP is self-routable over the Internet. Nodes in layer 3 are compute nodes as well as data store nodes. This makes data store nodes *smart* since data stores can store and process data in transit. At layer 4 we have a semantic layer which captures data processing knowledge from different groups. Semantic reasoning is used here to discover and explore new data processing paths. Layer 5 is the application layer. This layer applications can query the network which invokes workflow-like executions in the underlying layers. Interaction between layers is both-ways but we only envision communication between successive layers.

Layer 2 can talk down to layer 1 e.g. configuring OpenFlow[1] switches and can also talk up to layer 3 to reconfigure its layer based on the data transition network. Similarly layer 3 can talk up to layer 4 as a feedback loop from invocations brought about from layer 4. These require interfaces between layers and is part of our research in [19].

7.3 Future Research

The future work in this area is to concretely realize this vision. Most notably are the layers concerned with knowledge generation from data processing and feedback loops from data processing into the knowledge base. The shift from data to knowledge, we believe, will shift the focus of processing from the current data-centric processing to knowledge-centric processing. At the lower levels, further advancement in virtualization, standards and policies is needed to facilitate true restriction-free intercloud. One such example would be seamless migration of virtual machines between cloud service providers. This would result in further dynamism where VMs can autonomously migrate to *better* providers. The Internet of Things (IoT) must also be factored into the picture as it is becoming increasingly evident that capable smart devices with potentially many sensors are becoming widespread. How will such devices shape the future of data processing? The need to easily collect data in various stages of processing from the many devices reinforces the need for a data processing plane.

[1] https://www.opennetworking.org/sdn-resources/openflow

Bibliography

[1] Tony Hey, Stewart Tansley, and Kristin Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, Redmond, Washington, 2009.

[2] Presidents Council of Advisors on Science and Technology. Leadership under challenge: Information technology r&d in a competitive world an assessment of the federal networking and information technology r&d program. Technical report, August 2007.

[3] Frans Gens. IDC Predictions 2015: Accelerating Innovation and Growth on the 3rd Platform. *IDC predictions*, 2014.

[4] IDC 3rd Platform. `https://www.idc.com/getdoc.jsp?containerId=252700`. accessed: [18-01-2015].

[5] Jeff Hawkins and Sandra Blakeslee. *On Intelligence*. Henry Holt, 2004.

[6] Linked Data. `http://linkeddata.org`. accessed: [18-01-2015].

[7] Evans David. The internet of thingshow the next evolution of the internet is changing everything. *Cisco Internet Business Solutions Group (IBSG)*, April 2011. `https://www.cisco.com/web/about/ac79/.../IoT_IBSG_0411FINAL.pdf`.

[8] Charles W Schmidt. Trending now: Using social media to predict and track disease outbreaks. In *Environmental Health Perspectives*, 2012.

[9] Albert-Lszl Barabsi and E. Bonabeau. Scale-free networks. *Scientific American*, 288(60-69), 2003.

[10] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3), 2001.

[11] Uwe Schwiegelshohn, Rosa M. Badia, Marian Bubak, Marco Danelutto, Schahram Dustdar, Fabrizio Gagliardi, Alfred Geiger, Ladislav Hluchy, Dieter Kranzlmller, Erwin Laure, Thierry Priol, Alexander Reinefeld, Michael Resch, Andreas Reuter, Otto Rienhoff, Thomas Rter, Peter Sloot, Domenico Talia, Klaus Ullmann, Ramin

Yahyapour, and Gabriele von Voigt. Perspectives on grid computing. *Future Generation Computer Systems*, 26(8):1104 – 1115, 2010.

[12] Rajkumar Buyya, James Broberg, and Andrzej M. Goscinski. *Cloud Computing Principles and Paradigms*. Wiley Publishing, 2011.

[13] David P. Anderson. Boinc: A system for public-resource computing and storage. In Rajkumar Buyya, editor, *GRID*, pages 4–10. IEEE Computer Society, 2004.

[14] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - a hunter of idle workstations. In *ICDCS*, pages 104–111. IEEE Computer Society, 1988.

[15] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, MA, USA, 2004.

[16] Krishnaprasad Thirunarayan and Amit Sheth. Semantics-empowered approaches to big data processing for physical-cyber-social applications. 2013.

[17] Reginald Cushing, Ganeshwara Herawan Hananda Putra, Spiros Koulouzis, Adam Belloum, Marian Bubak, and Cees de Laat. Distributed computing on an ensemble of browsers. *Internet Computing, IEEE*, 17(5):54–61, 2013.

[18] Rudolf Strijkers, Reginald Cushing, Marc X Makkes, Pieter Meulenhoff, Adam Belloum, Cees de Laat, and Robert Meijer. Towards an operating system for intercloud. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 2, pages 63–68. IEEE, 2013.

[19] Marc X Makkes, Reginald Cushing, Mikolaj Branowski, Adam Belloum, Cees de Laat, and Rob Meijer. Data Intrinsic Networked Computing. Manuscript to be submitted for publication in IEEE Internet Computing, 2015.

[20] Clare Sansom. The dna deluge. *Scientific Computing World*, August 2007. `http://spectrum.ieee.org/biomedical/devices/the-dna-data-deluge`.

[21] P. Zikopoulos, D. deRoos, K. Parasuraman, T. Deutsch, J. Giles, and D. Corrigan. *Harness the Power of Big Data – The IBM Big Data Platform*. Mcgraw-Hill, 2012.

[22] Globus. `https://www.globus.org/`. accessed: [18-01-2015].

[23] Unicore. `https://www.unicore.eu/`. accessed: [18-01-2015].

[24] Egi. `http://www.egi.eu/`. accessed: [18-01-2015].

[25] Prace. `http://www.prace-ri.eu/`. accessed: [18-01-2015].

[26] Maciej Malawski, Maciej Kuzniar, Piotr Wjcik, and Marian Bubak. How to use google app engine for free computing. *IEEE Internet Computing*, 17(1):50–59, 2013.

[27] A new crankshaft for v8. `http://blog.chromium.org/2010/12/new-crankshaft-for-v8.html`. accessed: [15-01-2015].

[28] Node.js. `https://nodejs.org/`. accessed: [15-01-2015].

[29] Peercdn. `https://peercdn.com/`. accessed: [18-01-2015].

[30] Computer language benchmarks. `http://shootout.alioth.debian.org/u32/benchmark.php`. accessed: [15-01-2015].

[31] Albert-Laszlo Barabasi, Vincent W. Freeh, Hawoong Jeong, and Jay B. Brockman. Parasitic computing. *Nature*, 412:894–897, 30 August 2001.

[32] Ganeshwara Herawan Hananda Putra. Workflow orchestration on weevilscout. Master's thesis, University of Amsterdam, 3 2013. `https://staff.fnwi.uva.nl/a.s.z.belloum/MSctheses/thesis_Ganesh.pdf`.

[33] Spiros Koulouzis, Reggie Cushing, Kostas Karasavvas, Adam Belloum, and Marian Bubak. Enabling web services to consume and produce large datasets. *IEEE Internet Computing*, 16(1):52–60, 2012.

[34] X. Wen, G. Gu, Q. Li, Y. Gao, and X. Zhang. Comparison of open-source cloud management platforms: Openstack and opennebula. In *Fuzzy Systems and Knowledge Discovery (FSKD)*. IEEE, 2012.

[35] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, and Gabriel Iszlai. Exploring alternative approaches to implement an elasticity policy. In Ling Liu and Manish Parashar, editors, *IEEE CLOUD*, pages 716–723. IEEE, 2011.

[36] Tania Lorido-Botran, Jose Miguel-Alonso, and JoseA. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12(4):559–592, 2014.

[37] Rudolf Strijkers, Marc X. Makkes, Cees de Laat, and Robert Meijer. Internet factories: Creating application-specific networks on-demand. *Computer Networks*, 68(0):187 – 198, 2014. Communications and Networking in the Cloud.

[38] Reginald Cushing, Marc X. Makkes, Rudolf Strijkers, and Adam Belloum. Interclouds: Grid prosthesis for workflow systems. *The Fifth IEEE International Scalable Computing Challenge (SCALE 2012)*. `http://www.cloudbus.org/ccgrid2012/cfp-scale.html`.

[39] Reginald Cushing, Spiros Koulouzis, Adam Belloum, and Marian Bubak. Applying workflow as a service paradigm to application farming. *Concurrency and Computation: Practice and Experience*, 26(6):1297–1312, 2014.

[40] Reginald Cushing, Spiros Koulouzis, Adam Belloum, and Marian Bubak. Dynamic handling for cooperating scientific web services. In *E-Science (e-Science), 2011 IEEE 7th International Conference on*, pages 232–239. IEEE, 2011.

[41] Reginald Cushing, Spiros Koulouzis, Adam Belloum, and Marian Bubak. Prediction-based auto-scaling of scientific workflows. In *Proceedings of the 9th International Workshop on Middleware for Grids, Clouds and e-Science*, page 1. ACM, 2011.

[42] Reginald Cushing, Adam Belloum, Vladimir Korkhov, Dmitry Vasyunin, Marian Bubak, and Carole Leguy. Workflow as a service: an approach to workflow farming. In *Proceedings of the 3rd international workshop on Emerging computational methods for the life sciences*, pages 23–31. ACM, 2012.

[43] Tony Hey, Stewart Tansley, and Kristin Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, Redmond, Washington, 2009.

[44] Vladimir Korkhov, Dmitry Vasyunin, Adianto Wibisono, Victor Guevara-Masis, Adam Belloum, Cees de Laat, Pieter Adriaans, and L.O. Hertzberger. WS-VLAM: Towards a scalable workflow system on the grid. In *WORKS '07: Proceedings of the 2nd workshop on Workflows in support of large-scale science*, pages 63–68, New York, NY, USA, 2007. ACM.

[45] David Abramson, Rajkumar Buyya, and Jonathan Giddy. A computational economy for grid computing and its implementation in the nimrod-g resource broker. *Future Generation Comp. Syst.*, 18(8):1061–1074, 2002.

[46] Asim YarKhan, Jack Dongarra, and Keith Seymour. Gridsolve: The evolution of a network enabled solver. In PatrickW. Gaffney and JamesC.T. Pool, editors, *Grid-Based Problem Solving Environments*, volume 239 of *IFIP The International Federation for Information Processing*, pages 215–224. Springer US, 2007.

[47] Yoshio Tanaka, Hidemoto Nakada, Satoshi Sekiguchi, Toyotaro Suzumura, and Satoshi Matsuoka. Ninf-g: A reference implementation of rpc-based programming middleware for grid computing. *J. Grid Comput.*, 1(1):41–51, 2003.

[48] Henri Casanova, Graziano Obertelli, Francine Berman, and Richard Wolski. The apples parameter sweep template: User-level middleware for the grid. In Jed Donnelley, editor, *SC*, page 60. ACM, 2000.

[49] Henri Casanova and Fran Berman. Parameter sweeps on the grid with apst. In *Concurrency: Practice and Experiance*, 2002.

[50] Duncan Hull, Katy Wolstencroft, Robert Stevens, Carole A. Goble, Matthew R. Pocock, Peter Li, and Tom Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34(Web-Server-Issue):729–732, 2006.

[51] Andrew Harrison, Ian Taylor, Ian Wang, and Matthew Shields. Ws-rf workflow in triana. *Int. J. High Perform. Comput. Appl.*, 22(3):268–283, 2008.

[52] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew B. Jones, Bertram Ludscher, and Steve Mock. Kepler: An extensible system for design and execution of scientific workflows. In *SSDBM*, pages 423–424. IEEE Computer Society, 2004.

[53] Ewa Deelman, James Blythe, A Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei hui Su, Karan Vahi, and Miron Livny. Pegasus: Mapping scientific workflows onto the grid. pages 11–20, 2004.

[54] Andreas Hoheisel. User tools and languages for graph-based grid workflows: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1101–1113, 2006.

[55] Adam Barker, Jon Weissman, and Jano van Hemert. The Circulate architecture: avoiding workflow bottlenecks caused by centralised orchestration. *Cluster Computing*, 12:221–235.

[56] Shayan Shahand, Stephen J. Turner, Wentong Cai, and Maryam Khademi H. DynaSched: a dynamic web service scheduling and deployment framework for data-intensive grid workflows. *Procedia Computer Science*, 1(1):593 – 602, 2010. ICCS 2010.

[57] Markus Keidl, Stefan Seltzsam, and Alfons Kemper. Reliable web service execution and deployment in dynamic environments. In *In Proceedings of the International Workshop on Technologies for E-Services (TES*, pages 104–118, 2003.

[58] Vladimir Korkhov, Jakub T. Moscicki, and Valeria V. Krzhizhanovskaya. Dynamic workload balancing of parallel applications with user-level scheduling on the grid. *Future Generation Comp. Syst.*, 25(1):28–34, 2009.

[59] J.T. Moscicki, M. Lamanna, M. Bubak, and P.M.A. Sloot. Processing moldable tasks on the grid: Late job binding with lightweight user-level overlay. *Future Generation Computer Systems*, 27(6):725 – 736, 2011.

[60] Michael Russell Ed Seidel Gabrielle Allen, Tom Goodale and John Shalf. Classifying and enabling grid applications. In A. J. G. Hey F. Berman, G. Fox, editor, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley, 2003.

[61] Vladimir Korkhov, Dmitry Vasyunin, Adianto Wibisono, Adam Belloum, Mrcia A. Inda, Marco Roos, Timo M. Breit, and Louis O. Hertzberger. VLAM-G: Interactive data driven workflow engine for grid-enabled resources. *Scientific Programming*, 15(3):173–188, 2007.

[62] Rudolf J. Strijkers, Willem Toorop, Alain van Hoof, Paola Grosso, Adam Belloum, Dmitry Vasuining, Cees de Laat, and Robert J. Meijer. Amos: Using the cloud for on-demand execution of e-science applications. In *eScience*, pages 331–338. IEEE Computer Society, 2010.

[63] Erik Elmroth, Francisco Hernndez, and Johan Tordsson. Three fundamental dimensions of scientific workflow interoperability: Model of computation, language, and execution environment. *Future Generation Comp. Syst.*, 26(2):245–256, 2010.

[64] Warren Smith, Ian T. Foster, and Valerie E. Taylor. Scheduling with advanced reservations. In *IPDPS*, pages 127–132. IEEE Computer Society, 2000.

[65] W3C. `http://www.w3.org/TR/ws-archg`. accessed: [20-01-2015].

[66] Axis2. `http://axis.apache.org`. accessed: [20-01-2015].

[67] ActiveMQ. `http://activemq.apache.org`. accessed: [20-01-2015].

[68] Reginald Cushing, Adam Belloum, Marian Bubak, and Cees de Laat. Automata-based dynamic data processing for clouds. In *Euro-Par 2014: Parallel Processing Workshops*, volume 8805 of *Lecture Notes in Computer Science*, pages 93–104. Springer International Publishing, 2014.

[69] Reginald Cushing, Adam Belloum, Marian Bubak, and Cees de Laat. Towards Computing Without Borders: Data Processing Plane. Manuscript submitted for publication in Future Generation of Computer Systems, 2015.

[70] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(23):202 – 220, 2009. Distributed Computing Techniques.

[71] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[72] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[73] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[74] Apache Storm. `http://hortonworks.com/hadoop/storm/`. accessed: [18-01-2015].

[75] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[76] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[77] Paolo Missier, Stian Soiland-Reyes, Stuart Owen, Wei Tan, Aleksandra Nenadic, Ian Dunlop, Alan Williams, Thomas Oinn, and Carole Goble. Taverna, reloaded. In M. Gertz, T. Hey, and B. Ludaescher, editors, *SSDBM 2010*, Heidelberg, Germany, June 2010.

[78] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew B. Jones, Bertram Ludscher, and Steve Mock. Kepler: An extensible system for design and execution of scientific workflows. In *SSDBM*, pages 423–424. IEEE Computer Society, 2004.

[79] Reginald Cushing, Spiros Koulouzis, Adam Belloum, and Marian Bubak. Applying workflow as a service paradigm to application farming. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2013.

[80] Farhad Arbab. Composition of interacting computations. In Dina Goldin, ScottA. Smolka, and Peter Wegner, editors, *Interactive Computation*, pages 277–321. Springer Berlin Heidelberg, 2006.

[81] Mauricio Cortes. A coordination language for building collaborative applications. *Computer Supported Cooperative Work (CSCW)*, 9(1):5–31, 2000.

[82] Hector Fernandez, Cdric Tedeschi, and Thierry Priol. A chemistry-inspired workflow management system for scientific applications in clouds. In *eScience*, pages 39–46. IEEE Computer Society, 2011.

[83] Open Provenance Model (OPM). http://openprovenance.org/. accessed: [18-01-2015].

[84] G. Berry. The constructive semantics of pure esterel. http://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf, 1999.

[85] Michael J. Neely. *Stochastic Network Optimization with Application to Communication and Queueing Systems*. Morgan and Claypool Publishers, 2010.

[86] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[87] Marc X. Makkes and Reginald Cushing. Building the Internet of the Future. In *The Big Future of Data*, pages 64–65. COMMIT, 2014. http://www.commit-nl.nl/sites/default/files/TheBigFutureofData-ThedemosLow.pdf.

[88] Reginald Cushing, Marian Bubak, Adam Belloum, and Cees de Laat. Beyond scientific workflows: Networked open processes. In *eScience (eScience), 2013 IEEE 9th International Conference on*, pages 357–364, Oct 2013.

[89] Reginald Cushing, Spiros Koulouzis, Adam Belloum, and Marian Bubak. Dynamic handling for cooperating scientific web services. In *E-Science (e-Science), 2011 IEEE 7th International Conference on*, pages 232–239, Dec 2011.

[90] Spiros Koulouzis, Dmitry Vasyunin, Reginald Cushing, Adam Belloum, and Marian Bubak. Cloud data federation for scientific applications. In *Euro-Par 2013: Parallel Processing Workshops*, volume 8374 of *Lecture Notes in Computer Science*, pages 13–22. Springer Berlin Heidelberg, 2014.

[91] Reginald Cushing, Marian Bubak, Adam Belloum, and Cees de Laat. Beyond Scientific Workflows: Networked Open Processes. In *IEEE 9th International Conference on eScience*, pages 357–364, 2013.

[92] Adam Belloum, Mrcia A. Inda, Dmitry Vasunin, Vladimir Korkhov, Zhiming Zhao, Han Rauwerda, Timo M. Breit, Marian Bubak, and Louis O. Hertzberger. Collaborative e-science experiments and scientific workflows. *IEEE Internet Computing*, 15(4):39–47, 2011.

[93] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia C. Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for

mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[94] Sean Bechhofer, Iain E. Buchan, David De Roure, Paolo Missier, John D. Ainsworth, Jiten Bhagat, Philip A. Couch, Don Cruickshank, Mark Delderfield, Ian Dunlop, Matthew Gamble, Danius T. Michaelides, Stuart Owen, David R. Newman, Shoaib Sufi, and Carole A. Goble. Why linked data is not enough for scientists. *Future Generation Comp. Syst.*, 29(2):599–611, 2013.

[95] myExperiment. `http://www.myexperiment.org`. accessed: [18-01-2015].

[96] SADI. `http://sadiframework.org`. accessed: [18-01-2015].

[97] Programmable Web. `http://www.programmableweb.com/`. accessed: [18-01-2015].

[98] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001.

[99] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data - the story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009.

[100] Resource Description Framework. `http://www.w3.org/RDF/`. accessed: [18-01-2015].

[101] Mark Wilkinson, Benjamin Vandervalk, and Luke McCarthy. The semantic automated discovery and integration (sadi) web service design-pattern, api and reference implementation. *Journal of Biomedical Semantics*, 2(1):8, 2011.

[102] John Domingue, Carlos Pedrinaci, Maria Maleshkova, Barry Norton, and Reto Krummenacher. Fostering a relationship between linked data and the internet of services. In *Future Internet Assembly*, volume 6656 of *Lecture Notes in Computer Science*, pages 351–366. Springer, 2011.

[103] Tomasz Gubala, Katarzyna Prymula, Piotr Nowakowski, and Marian Bubak. Semantic integration for model-based life science applications. In Tuncer ren, Janusz Kacprzyk, Leifur . Leifsson, Mohammad S. Obaidat, and Slawomir Koziel, editors, *SIMULTECH*, pages 74–81. SciTePress, 2013.

[104] Bio Catalogue. `http://www.biocatalogue.org`. accessed: [18-01-2015].

[105] OWL-S. `http://www.w3.org/Submission/OWL-S`. accessed: [18-01-2015].

[106] Dieter Fensel, Holger Lausen, Axel Polleres, Jos De Bruijn, Michael Stollberg, Dumitru Roman, and John Domingue, editors. *Enabling Semantic Web Services: The Web Service Modeling Ontology*. Springer-Verlag, Heidelberg, 2006.

[107] SAWSDL. `http://www.w3.org/2002/ws/sawsdl`. accessed: [18-01-2015].

[108] Carlos Pedrinaci, Jacek Kopecký, Maria Maleshkova, Dong Liu, Ning Li, and John Domingue. Unified lightweight semantic descriptions of web apis and web services. In *W3C Workshop on Data and Services Integration*, 2011.

[109] Mark Wilkinson, Heiko Schoof, Rebecca Ernst, and Dirk Haase. Biomoby successfully integrates distributed heterogeneous bioinformatics web services. the planet exemplar case. *Plant Physiol*, 138(1):5–17, May 2005.

[110] Gnu octave. `http://www.gnu.org/software/octave`. accessed: [20-01-2015].

[111] The distributed ASCI supercomputer 3. `http://www.cs.vu.nl/das3`. accessed: [20-01-2015].

[112] UniProtKB. `http://www.uniprot.org`. accessed: [20-01-2015].

[113] BioJava API. `http://biojava.org`. accessed: [20-01-2015].

[114] Wouter Huberts. *Personalized computational modeling of vascular access creation*. PhD thesis, University of Maastricht, 2012. `http://digitalarchive.maastrichtuniversity.nl/fedora/get/guid:6063b962-9556-4cb4-9435-201df8e0145e/ASSET1`.

[115] Huberts Wouter et al. A pulse wave propagation model to support decision-making in vascular access planning in the clinic. *Medical engineering and physics*, 34(2), 2012.

[116] J. T. Moscicki. *Understanding and Mastering Dynamics in Computing Grids Processing Moldable Tasks with User-Level Overlay*. PhD thesis, University of Amsterdam, 2011. `http://dare.uva.nl/record/1/333467`.

[117] Tracula. `http://surfer.nmr.mgh.harvard.edu/fswiki/Tracula`. accessed: [18-01-2015].

[118] Reginald Cushing, Spiros Koulouzis, Rudolf Strijkers, Adam Belloum, and Marian Bubak. Service level management for executable papers. In *Euro-Par 2011: Parallel Processing Workshops*, pages 116–123. Springer, 2012.

[119] Rudolf Strijkers, Reginald Cushing, Dmitry Vasyunin, Cees de Laat, Adam S.Z. Belloum, and Robert Meijer. Toward executable scientific publications. *Procedia Computer Science*, 4(0):707 – 715, 2011. Proceedings of the International Conference on Computational Science, ICCS 2011.

[120] Reginald Cushing, Adam Belloum, Marian Bubak, and Cees de Laat. Towards open linked processes for scientific computing. `https://tnc2013.terena.org/core/poster/26`. *poster in TERENA Networking Conference (TNC) 2013*.

PUBLICATION AUTHORSHIP

Author contributions to the publications used in this thesis.

- **Chapter 2**
  - Reginald Cushing, Ganeshwara Herawan Hananda Putra, Spiros Koulouzis, Adam Belloum, Marian Bubak, and Cees de Laat. Distributed computing on an ensemble of browsers. *Internet Computing, IEEE*, 17(5):54–61, 2013

    R.C. designed, implemented and performed the experiments. G.H.H.P contributed in WebCL tests. S.K. contributed in performing the experiments. A.S.Z.B. consulted the study and publication. M.B. supervised the research and publication.
  - Rudolf Strijkers, Reginald Cushing, Marc X Makkes, Pieter Meulenhoff, Adam Belloum, Cees de Laat, and Robert Meijer. Towards an operating system for intercloud. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 2, pages 63–68. IEEE, 2013

    R.S. designed and performed the experiments. R.C. implemented the application middleware stack. P.M. and M.X.M contributed in the implementation of the infrastructure control stack. A.S.Z.B. consulted the study and publication. C.d.L and R.M supervised the work.
  - Marc X Makkes, Reginald Cushing, Mikolaj Branowski, Adam Belloum, Cees de Laat, and Rob Meijer. Data Intrinsic Networked Computing. Manuscript to be submitted for publication in IEEE Internet Computing, 2015

    M.X.M designed and performed the experiments. R.C. implemented the application middleware stack. A.S.Z.B. consulted the study and publication. C.d.L and R.M supervised the work.
- **Chapter 3**
  - Reginald Cushing, Spiros Koulouzis, Adam Belloum, and Marian Bubak. Applying workflow as a service paradigm to application farming. *Concurrency and Computation: Practice and Experience*, 26(6):1297–1312, 2014

    R.C. designed, implemented and performed the experiments. S.K. contributed in extending Freefluo workflow engine. A.S.Z.B. consulted the study and publication and M.B. supervised the research and publication.

– Reginald Cushing, Spiros Koulouzis, Adam Belloum, and Marian Bubak. Dynamic handling for cooperating scientific web services. In *E-Science (e-Science), 2011 IEEE 7th International Conference on*, pages 232–239. IEEE, 2011

  R.C. designed, implemented and performed the experiments. S.K. contributed in the implementation of Axis2 transport handlers. A.S.Z.B. consulted the study and publication and M.B. supervised the research and publication.

– Reginald Cushing, Spiros Koulouzis, Adam Belloum, and Marian Bubak. Prediction-based auto-scaling of scientific workflows. In *Proceedings of the 9th International Workshop on Middleware for Grids, Clouds and e-Science*, page 1. ACM, 2011

  R.C. designed, implemented and performed the experiments. S.K. contributed in extending Freefluo workflow engine. A.S.Z.B. consulted the study and publication and M.B. supervised the research and publication.

– Reginald Cushing, Adam Belloum, Vladimir Korkhov, Dmitry Vasyunin, Marian Bubak, and Carole Leguy. Workflow as a service: an approach to workflow farming. In *Proceedings of the 3rd international workshop on Emerging computational methods for the life sciences*, pages 23–31. ACM, 2012

  R.C. designed, implemented and performed the experiments. A.S.Z.B. consulted the study and publication and contributed to running the experiments. V.K., C.L. and D.V. provided the application use-case. M.B. supervised the research and publication.

▪ **Chapter 4**

– Reginald Cushing, Adam Belloum, Marian Bubak, and Cees de Laat. Automata-based dynamic data processing for clouds. In *Euro-Par 2014: Parallel Processing Workshops*, volume 8805 of *Lecture Notes in Computer Science*, pages 93–104. Springer International Publishing, 2014

  R.C. designed, implemented and performed the experiments. A.S.Z.B. consulted the study and publication. M.B. supervised the research and publication and C.d.L. performed an overall supervision.

– Reginald Cushing, Adam Belloum, Marian Bubak, and Cees de Laat. Towards Computing Without Borders: Data Processing Plane. Manuscript submitted for publication in Future Generation of Computer Systems, 2015

  R.C. designed, implemented and performed the experiments. A.S.Z.B. consulted the study and publication. M.B. supervised the research and publication and C.d.L performed an overall supervision.

▪ **Chapter 5**

– Reginald Cushing, Marian Bubak, Adam Belloum, and Cees de Laat. Beyond Scientific Workflows: Networked Open Processes. In *IEEE 9th International Conference on eScience*, pages 357–364, 2013

  R.C. designed implemented and performed the experimented. A.S.Z.B. consulted the study and publication. M.B. supervised the research and publication and C.d.L. performed an overall supervision.

PUBLICATIONS

Publications in peer-reviewed journals

- Reginald Cushing, Ganeshwara Herawan Hananda Putra, Spiros Koulouzis, Adam Belloum, Marian Bubak, and Cees de Laat. Distributed computing on an ensemble of browsers. *Internet Computing, IEEE*, 17(5):54–61, 2013.
- Reginald Cushing, Spiros Koulouzis, Adam Belloum, and Marian Bubak. Applying workflow as a service paradigm to application farming. *Concurrency and Computation: Practice and Experience*, 26(6):1297–1312, 2014.
- Reginald Cushing, Adam Belloum, Marian Bubak, and Cees de Laat. Towards Computing Without Borders: Data Processing Plane. Manuscript submitted for publication in Future Generation of Computer Systems, 2015.
- Spiros Koulouzis, Reggie Cushing, Kostas Karasavvas, Adam Belloum, and Marian Bubak. Enabling web services to consume and produce large datasets. *IEEE Internet Computing*, 16(1):52–60, 2012.

Publications in peer-reviewed conference proceedings

- Reginald Cushing, Adam Belloum, Vladimir Korkhov, Dmitry Vasyunin, Marian Bubak, and Carole Leguy. Workflow as a service: an approach to workflow farming. In *Proceedings of the 3rd international workshop on Emerging computational methods for the life sciences*, pages 23–31. ACM, 2012.
- Reginald Cushing, Spiros Koulouzis, Adam Belloum, and Marian Bubak. Prediction-based auto-scaling of scientific workflows. In *Proceedings of the 9th International Workshop on Middleware for Grids, Clouds and e-Science*, page 1. ACM, 2011.
- Reginald Cushing, Spiros Koulouzis, Adam Belloum, and Marian Bubak. Dynamic handling for cooperating scientific web services. In *E-Science (e-Science), 2011 IEEE 7th International Conference on*, pages 232–239. IEEE, 2011.
- Reginald Cushing, Marian Bubak, Adam Belloum, and Cees de Laat. Beyond Scientific Workflows: Networked Open Processes. In *IEEE 9th International Conference on eScience*, pages 357–364, 2013.

- Reginald Cushing, Adam Belloum, Marian Bubak, and Cees de Laat. Automata-based dynamic data processing for clouds. In *Euro-Par 2014: Parallel Processing Workshops*, volume 8805 of *Lecture Notes in Computer Science*, pages 93–104. Springer International Publishing, 2014.
- Reginald Cushing, Spiros Koulouzis, Rudolf Strijkers, Adam Belloum, and Marian Bubak. Service level management for executable papers. In *Euro-Par 2011: Parallel Processing Workshops*, pages 116–123. Springer, 2012.
- Rudolf Strijkers, Reginald Cushing, Dmitry Vasyunin, Cees de Laat, Adam S.Z. Belloum, and Robert Meijer. Toward executable scientific publications. *Procedia Computer Science*, 4(0):707 – 715, 2011. Proceedings of the International Conference on Computational Science, ICCS 2011.
- Rudolf Strijkers, Reginald Cushing, Marc X Makkes, Pieter Meulenhoff, Adam Belloum, Cees de Laat, and Robert Meijer. Towards an operating system for intercloud. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 2, pages 63–68. IEEE, 2013.
- Spiros Koulouzis, Dmitry Vasyunin, Reginald Cushing, Adam Belloum, and Marian Bubak. Cloud data federation for scientific applications. In *Euro-Par 2013: Parallel Processing Workshops*, volume 8374 of *Lecture Notes in Computer Science*, pages 13–22. Springer Berlin Heidelberg, 2014.

Other publications

- Reginald Cushing, Adam Belloum, Marian Bubak, and Cees de Laat. Towards open linked processes for scientific computing. `https://tnc2013.terena.org/core/poster/26`. *poster in TERENA Networking Conference (TNC) 2013*.
- Reginald Cushing, Marc X. Makkes, Rudolf Strijkers, and Adam Belloum. Interclouds: Grid prosthesis for workflow systems. *The Fifth IEEE International Scalable Computing Challenge (SCALE 2012)*. `http://www.cloudbus.org/ccgrid2012/cfp-scale.html`.

Supervision

- Ganeshwara Herawan Hananda Putra. Workflow orchestration on weevilscout. Master's thesis, University of Amsterdam, 3 2013. `https://staff.fnwi.uva.nl/a.s.z.belloum/MSctheses/thesis_Ganesh.pdf`.

SUMMARY

Distributed computing has always been a challenge due to the NP-completeness of finding optimal underlying management routines. The advent of big data increases the dimensionality of the distributed data processing problem whereby data partitionability, processing complexity and locality play a crucial role in the effectiveness of distributed systems. The flexibility and control brought forward by virtualization means that for the first time we control the whole stack from application down to the network layer but, to a certain extent, the best way to exploit this level of programmability still eludes us.

Tackling this problem means confronting data processing from different fronts. The spectrum of data processing ranges from the abstract, semantic level down to the actual execution and infrastructure levels. A breadth-first research approach to the data processing spectrum allows us relate and investigate the different dimensions of data processing while also studying the interaction between these different dimensions. Models and semantics of data processing allows us to reason about what data processing is A better understating of data in its transient processing state could be achieved through effective models or transformation schemas. A data processing schema is in effect an automaton describing the various stages of data processing. The data schema is a blue print of what can be done with data which also enriches provenance data since we know, at any stage of data processing, what has been done and what can be done with data. Using the schema as the fulcrum of our approach to distributed data processing we can map the data schema onto several layers including the workflow, the resources and the network so much so that data processing becomes a plane without borders.

Fitting with the current dynamism in infrastructures, an effective system is one which is also dynamic in nature and can change and evolve during runtime. Such dynamism includes: task farming in workflows, prediction based scaling, autonomous controlled scaling, and data processing path discovery. This thesis presents four steps leading to such a distributed data-oriented computing.

The Internet browser is, most probably, the single most widespread piece of software installed on mobile devices and computers alike. With 2 billion users online and the shift towards online services, computing through Internet browsers has the potential of amassing immense resources. We demonstrate how a cluster of globally distributed Internet browsers is used to compute thousands of bio-informatics tasks.

Next, we investigate an approach to farm workflows employing the service oriented paradigm in combination with the workflow manager to create, control and monitor workflows applications and their components. We study two types of workflow farming: task-level whereby task harness acts as services by being invoked on which task to load, and data-level where the actual task is invoked as a service with different chunks of data to process.

Then, we propose a new data-centric workflow task scaling technique. Scaling is achieved through a prediction mechanism where the input data load on each task within a workflow is used to compute the estimated task execution time. Through load prediction, the workflow system can take informed decisions on scaling multiple workflow tasks independently to improve overall throughput and reduce workflow bottlenecks. We demonstrate an autonomous mechanism for fair sharing of resources between competing yet collaborating tasks. Each task is running a fuzzy controller which is autonomously and continuously trying to ameliorate its resource usage without starving others.

The most important contribution is modeling data processing as units of data undergoing transformation which can be modelled using automata. The state graphs provides deeper insight into data manipulation during processing whereby provenance is intrinsically embedded in the model. Implementation-wise, the automaton acts as a routing information and thus, capsules of data, automaton, code and state are self routable through a distributed network

Going beyond data processing, we show how links between services can be made purely at the semantic layer. This introduced the notion of Networked Open Processes which, akin to Open Linked Data, tries to find interoperable services ith the aim of enriching the ever increasing open data.

The aforementioned contributions culminate in a system, PUMPKIN, which tries to encompass all techniques. PUMPKIN can be considered as an overlay data defined network for distributed data processing. In PUMPKIN we achieve computing interoperability through a data processing protocol. Application use-cases vary from streaming I/O intensive applications to long-running CPU intensive applications.

SAMENVATTING

De optimalisatie van onderliggende routines in gedistribueerde applicaties is uitdagend, van wege het voldongen feit dat deze routines NP-volledig zijn. De komst van *big data* verhoogt de dimensionaliteit van het gedistribueerde data verwerking probleem, hierbij gaan data partitioneerbaarheid, complexiteit en lokaliteit een cruciale rol voor de doeltreffendheid van het gedistribueerde systemen. De flexibiliteit en controle die voort wordt gebracht door virtualisatie betekent dan voor eerst alle lagen van de toepassing gecontroleerd kunnen worden, tot aan de netwerklaag met een zekere beperking. De beste manier deze controle te gebruiken op het niveau van programmeerbaarheid ontgaat ons nog steeds.

De aanpak van dit probleem betekend dat we data verwerking moeten confronteren van uit verschillende fronten. Het spectrum van de gegevensverwerking varieert van de abstracte, semantisch niveau naar de feitelijke uitvoering en infrastructuur niveaus. Een breedte-eerst speurtocht benadering van het verwerking van gegevens spectrum, laat ons relateren en onderzoeken naar de verschillende dimensies van het verwerken van data, terwijl we ook een studie doen naar interactie tussen verschillende dimensies. Modellen en semantiek van gegevensverwerking stelt ons in staat te redeneren over welke gegevens verwerking is. Het beter begrijpen van de data die verwerkt wordt kan worden bereikt door middel van effectieve modellen van tranformatieschemas. Een dataverwerkingsschemas is in feite een automaat die de verschillende fasen van de gegevensverwerking beschrijft. Een gegevensschema is een blauwdruk van de mogelijkheden van wat er met de data kan worden gedaan, dat zorcht er tevens voor dat het verrijkt worden met de verwerking history, en dat bij elke fase van de verwerking duidelijk word welke mogelijkheden er nog over zijn. Als we nu onze schema als uitgangspunt gebruiken van onze aanpak om gedistribueerde gegevensverwerking te doen, kunnen we het data schema op verschillende lagen in kaart brengen, waaronder: de workflow, de (computer)middelen en het netwerk, zozeer zelfs dat dataverwerking wordt als een vliegtuig zonder grenzen

Om met de huidige dynamiek in de infrastructuur om te gaan, zou een doeltreffend system zelf van dynamische aard moeten zijn, en kunnen veranderen en evolueren tijdens de looptijd. Dergelijke systeem dynamiek omvat: *task farming* in werkstromen, schalen op basis van voorspelling, autonoom gecontroleerd schalen, en automatisch ontdekken van het dataverwerking pad Dit proefschrift presenteert vier stappen die leiden tot een dergelijke een gedistribueerde data-georiënteerde computing.

De Internet browser is, hoogstwaarschijnlijk, de meest wijdverspreide stukje software en genstalleerd op mobiele apparaten en computers. Met 2 miljard online gebruikers en een verschuiving naar online-diensten, hebben computers, via Internet browsers, een potentieel van het vergaren van enorme hoeveelheden middelen. We laten zien hoe een cluster van wereldwijd gedistribueerde internet browsers wordt gebruikt om duizenden bioinformatica taken uit te voeren. Vervolgens hebben we een *farm* workflows met een dienst georiënteerde paradigma in combinatie met workflow manager om workflow applicaties en hun componenten te observeren en te controleren. We bestuderen twee typen *farm* workflows: op taak-niveau, waarbij een taak harnas fungeert als een dienst en de dienst wordt aangeroepen om een taak uit te voeren, en op data-niveau waarbij de taak wordt aangeroepen als een dienst met verschillende stukken data die moeten worden verwerkt.

Daarna, stellen we een nieuw schaalbaarheidstechniek voor, waarin data centraal staat in de workflow taak. Hier wordt schaalbaarheid bereikt door het een voorspelling mechanisme die berekend, aan de hand van de binnenkomen data, wat de verwachte data uitvoer tijd zal zijn voor elke taak.

Door het in acht nemen van de voorspelling van de uitvoertijd, kan het workflow systeem een wel overwogen beslissing nemen over het opschalen van meerdere onafhankelijke workflow taken om prestatie van het algehele systeem te verbeteren en knelpunten te voorkomen. We demonstreren een autonoom mechanisme, die voor eerlijke verdeling van middelen zocht, tussen concurrerende nog samenwerkende taken. Elke taak heeft een fuzzy controller, die autonoom is en voortdurend probeert om het gebruik van bronnen te verbeteren zonder andere processen te laten verhongeren.

De belangrijkste bijdrage van deze thesis is het modelleren van data verwerking als eenheden van de gegevens waarbij de transformatie die kan worden gemodelleerd met automaten. De status grafieken geven dieper inzicht in de manipulatie van gegevens tijdens het verwerken waarbij de herkomst intrinsiek is ingebed in het model. De staat grafieken geeft dieper inzicht in Praktisch gezien werkt het als volgt, de automaat werk als routeringsinformatie en word geëncapsuleerd samen met de data, status, en code, dit zorgt ervoor dat routeerbaar is in een gedistribueerd netwerk.

Overstijgende aan gegevensverwerking, tonen we aan hoe de banden tussen diensten uitsluiten kunnen worden gemaakt op de semantische laag. Deze introduceerd het begrip van *Networked Open Processes* die verwant zijn aan *Open linked data*, die zijn beurt probeert de interoperabiliteit te vinnn tussen diensen en het verrijken van de groeiende hoeveelheid aan open data.

De bovengenoemde bijdragen zijn uitmond in een systeem, PUMPKIN, die tracht alle technieken omvatten. PUMPKIN kan worden beschouwd als een boven op liggende data netwerk voor gedistribueerde gegevensverwerking. In PUMKIN hebben we de verwerkingsinteroperabiliteit opgelost door middel van een data verwerking protocol. Appli-

caties casussen die wij in acht hebben genomen verschillen van *Streaming I/O* intensieve toepassingen tot en met langlopende CPU-intensieve applicaties.

ACKNOWLEDGMENTS

The journey here was an expedition through many winding roads, literally and metaphorically, obstacles and sometimes seemingly impasses. It would not have been possible without the help of so many people who patiently stood at crossroads showing me the way forward. Words cannot but express my deepest gratitude for all the people who helped along the way; Adam Belloum who had to endure my daily jabbering and make sense of it, Marian Bubak who hosted me many times at Krakow for sessions of intense thesis hackathons and Cees de Laat for his valued insight into new research directions and new angles of attack. Ideas where, at many times, polished and waxed through collaborations and long discussions sometimes past the wee hours. For this, I like to thank Rudolf Strijkers, Marc Makkes, Spiros Koulouzis, Dmitry Vasunin, Mikołay Baranowski and Ana-Maria Oprescu. I would also like to thank the whole SNE group for the sense of cohesion, academically as well as socially, which created an amiable atmosphere to work in. The journey would not have been pleasant without the new friendships which were forged along the way; a big thank you to the usual suspects who found me not too annoying to hang around. Last but not least, Daniela, who through, thick and thin, has accompanied me throughout the journey and always found ways to brighten up the dark gloomy days.