A survey on the Distributed Computing stack

Cristian Ramon-Cortes[a,*], Pol Alvarez[a], Francesc Lordan[a], Javier Alvarez[a], Jorge Ejarque[a], Rosa M. Badia[a]

[a] *Barcelona Supercomputing Center (BSC), Barcelona, Spain*

Abstract

In this paper, we review the background and the state of the art of the Distributed Computing software stack. We aim to provide the readers with a comprehensive overview of this area by supplying a detailed big-picture of the latest technologies. First, we introduce the general background of Distributed Computing and propose a layered top-bottom classification of the latest available software. Next, we focus on each abstraction layer, i.e. *Application Development* (including Task-based Workflows, Dataflows, and Graph Processing), *Platform* (including Data Sharing and Resource Management), *Communication* (including Remote Invocation, Message Passing, and Message Queuing), and *Infrastructure* (including Batch and Interactive systems). For each layer, we give a general background, discuss its technical challenges, review the latest programming languages, programming models, frameworks, libraries, and tools, and provide a summary table comparing the features of each alternative. Finally, we conclude this survey with a discussion of open problems and future directions.

Keywords: Distributed Systems, Distributed Programming Models, Distributed Computing, Cloud Computing, Task-based Workflows, Dataflows, Graph Processing, Streaming, Data Sharing, Resource Management, Infrastructure Managers

*Corresponding author

Email addresses: cristian.ramoncortes@bsc.es (Cristian Ramon-Cortes), pol.alvarez@bsc.es (Pol Alvarez), francesc.lordan@bsc.es (Francesc Lordan), javier.alvarez@bsc.es (Javier Alvarez), jorge.ejarque@bsc.es (Jorge Ejarque), rosa.m.badia@bsc.es (Rosa M. Badia)

Contents

1. Introduction and Context

Several years ago, the IT community shifted from sequential programming to parallel programming [1] to fully exploit the computing resources of multi-core processors. The current generation of software applications requires more

resources than those offered by a single computing node [2]. Thus, the community has been forced to evolve again towards distributed computing, that is, to obtain a larger amount of computing power by using several interconnected machines [3]. However, one of the major issues that arise from both parallel and distributed programming is that writing parallel code is not as easy as writing sequential programs [4]. Frequently, experienced developers find it difficult to write efficient parallel code. In the line of "writing programs that scale with increasing number of cores should be as easy as writing programs for sequential computer" [5], several libraries, tools, frameworks, programming models, and programming languages have emerged to help developers to handle the underlying infrastructure.

Currently, the IT community requires parallelisation and distributed systems to handle large amounts of data [6, 7]. Towards this, *Big-Data Analytics (BDA)* [8] appeared some years ago; allowing the community to store, check, retrieve and transform enormous amounts of data in acceptable response times. In addition, several programming models have also arisen that are completely different to the ones used by the *High Performance Computing (HPC)* community. In the race towards *Exascale Computing* [9], the IT community has realised that unifying *HPC* platforms and *Big-Data (BD) Ecosystems* is a must. Currently, these two ecosystems differ significantly at hardware and software level, but "programming models and tools are perhaps the biggest points of divergence between the scientific-computing and *Big-Data Ecosystems*" [10]. In this respect, "there is a clear desire on the part of a number of domain and computer scientists to see a convergence between the two high-level types of computing systems, software, and applications: *Big Data Analytics* and *High Performance Computing*" [11].

This paper reviews the state of the art of the Distributed Computing stack by providing a comprehensive classification of the latest technologies. More specifically, we propose a layered top-bottom classification that includes *Application Development* (i.e., Task-based Workflows, Dataflows, and Graph Processing), *Platform* (i.e., Data Sharing and Resource Management), *Communication* (i.e.,

Remote Invocation, Message Passing, and Message Queuing), and *Infrastructure* (i.e., Batch and Interactive systems). Furthermore, for each layer, we describe the general background, discuss its technical challenges, provide a summary table comparing the features of each alternative, review the latest software, and analyse the key differences that can help readers when choosing the adequate software for their needs. The rest of the paper is structured as follows. Section 2 gives a general overview of the proposed taxonomy by defining each layer and the interactions between them. Sections 3, 4, 5 and 6 explain in-depth each layer and categorise the latest frameworks. Finally, Section 7 concludes the paper and states some guidelines for future work.

2. General Overview

The distinction between HPC and BDA software stacks has blurred during the past years to develop Exascale applications. This research trend enables the new generation of applications to combine HPC and BDA software depending on their requirements. One of the major concerns when developing Exascale applications is the performance and the scalability of the chosen software stacks. Since these applications often rely on huge all-in-one software stacks (due to its feature richness and high abstraction), two major factors affect the performance and the scalability: (i) the components themselves and (ii) the interactions between them.

This survey covers the latest software alternatives available in each abstraction layer so that application developers can evaluate and consider the different alternatives for their applications. Although discussing the interactions between the different layers can be as important as discussing the software alternatives themselves, we believe that middleware developers should be in charge of these interactions; freeing application developers from this burden. Also, although there are still open questions such as how the HPC stack (e.g., GPFS, SLURM, MPI) and the Big Data stack (e.g., HDFS, YARN, TensorFlow) should borrow and adapt the different components, relevant work has already been published

targeting the HPC and BDA convergence and their interactions. For instance, S. Cano-Lores et al. [12] establish a research road-map to build a platform suitable for hybrid HPC and Big Data applications; thus reducing the gap between Big Data application models and data-intensive HPC. Also, C. Hsu et al. [13] discuss the progress towards big data and HPC convergence; focusing on big data systems and optimisations, novel techniques for big data analytics, sustainable architectures and applications.

Concerning the alternatives themselves, a large number of libraries, tools, frameworks, programming models, and programming languages have appeared to cover the *BDA* and *HPC* needs [14]. Certain contributions have gone one step further by building complex software stacks to provide a high-level abstraction for the development of distributed applications (such as Apache Spark [15], Apache Storm [16], or TensorFlow [17]). From the point of view of the developers of distributed applications, this fact has led to the possibility of choosing the most suitable software stack for the final application; fostering an (in)sane competition between them that has finally exploited in an uncontrollable and uncountable number of possibilities.

In this sense, we consider that there is no clear source of information to classify, describe, discuss, and review the wide variety of complex software stacks that are currently available for developing distributed applications. J. Liu et al. [18], K. Krauter et al. [3], and B. P. Rimal et al. [19] provide a partial but detailed survey on workflow managers, resource management, and cloud computing, respectively. Also, C. K. Emani et al. [20] provide a general picture about the Big Data software. In contrast, our survey describes and discusses the whole distributed computing stack.

As depicted in Figure 1, we propose a top-bottom layered taxonomy to sort the latest software from the highest abstraction frameworks, programming models, and programming languages to the most specific tools, and libraries. Notice that our proposed classification is designed to help developers to choose the best option for developing their distributed applications, and thus, each layer is based on answering the question *What does the software abstract the application*

developer from? Moreover, we have also defined the interactions between the different layers, and we have internally divided each layer by software purpose. To provide a more extensive guide, we have also selected the key features and elaborated comparison tables to compare the different options available in each layer.

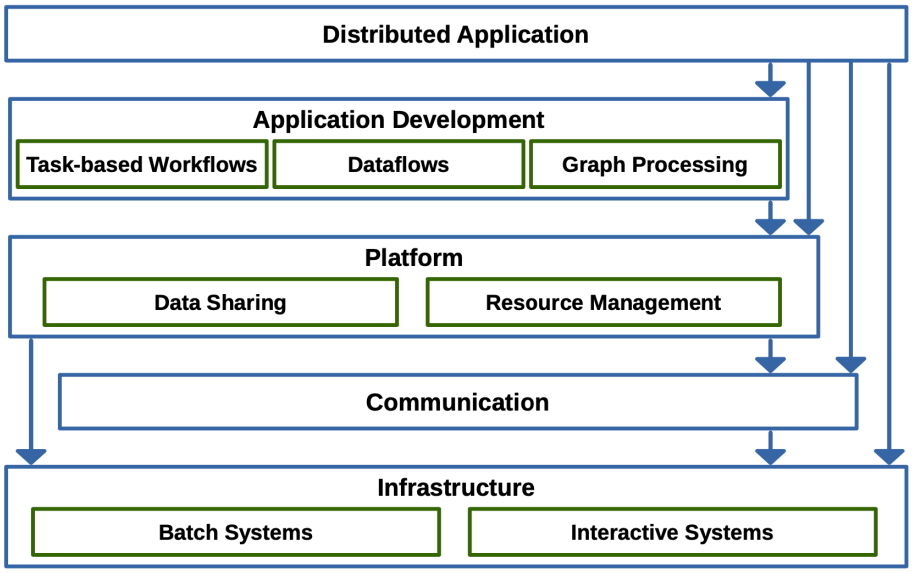

Figure 1: Classification Layers based on the abstraction level

Following the previous figure from top to bottom, *Distributed Applications* always require an *Infrastructure* (either batch or interactive systems) that can be handled explicitly (right-most arrow) or by intermediate frameworks. We categorise these frameworks into different layers depending on the level of abstraction that they can provide.

The *Application Development* layer provides high-abstraction software that, more or less transparently, handles the distributed computation, the data, and the resources. These programming languages, programming models, or frameworks are all-in-one solutions to develop *Distributed Applications* that, often, rely on huge software stacks (built from software from below layers).

On the other hand, *Distributed Applications* can also rely on *Platform* software to orchestrate, communicate, and manage the *Infrastructure*. Since this abstraction layer includes *Data Sharing* and *Resource Management*, the users must take care of distributing the computation among the available resources but do not need to deal with resource allocation and deallocation, fault-tolerance, data transfers, or distributed processes coordination.

Finally, *Distributed Applications* might only use *Communication* libraries or protocols to ease the communication between distributed processes. However, there is no abstraction at this level, and users need to deal with all the parallel and distributed challenges directly.

3. Application Development

The distributed computing premise is simple: solving a large problem with an enormous amount of computations as fast as possible by dividing it into smaller problems, dealing with them parallelly and distributedly, and gathering the results back. However, its implementation is not that simple because it can either lead to significant speed-ups or overheads due to the distributed computing challenges. These challenges range from the *Resource Management* to the *Data Distribution*, going through the *Coordination* and the *Monitoring* of the different distributed components.

In this sense, the community increasingly prefers to rely on high-abstraction frameworks to focus only on the application development by using any programming language, programming model or framework that fully abstracts the user from the distributed computing challenges; either by relying on other state of the art software, or by handling them explicitly. Moreover, these frameworks are expected to be easy to install, configure, and use so that they can be rapidly adapted to any application.

Representing the highest layer of the software stack and providing an almost ready-to-use option to implement distributed applications are the crucial points of the success of the *Application Development* software. However, they are also

the worst black spots at the technical level since high-abstraction can only be achieved by building huge software stacks or extensive frameworks that are, in both cases, hard to maintain. Also, ready-to-use tools require automatic configurations that must support heterogeneous underlying platforms that are continuously upgraded.

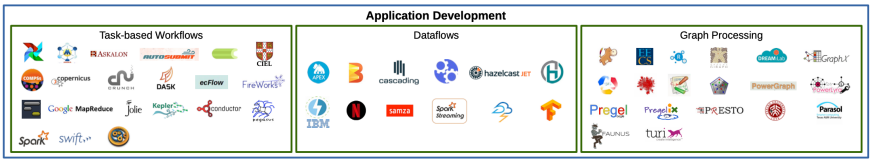

Figure 2: Application Development Layer

As shown in Figure 2, we have divided this layer into three categories depending on the software purpose. Next subsections provide further information about each category and its latest software.

3.1. Task-based Workflows

The software targeting *Task-based Workflows* allows the users to define pieces of code to be remotely executed as *tasks* and dependencies between tasks to combine them together into *workflows*. The main common feature in this family of software is that the principal working unit is the *task*.

3.1.1. Taxonomy

Table 1 presents our taxonomy of the surveyed task-based frameworks, where the different frameworks have been categorised in 3 main categories: programming, task-flow definition and runtime features. Regarding the programming category, we can classify tools by the programming interfaces as Graphical User Interface (G in the table), Command Line Interface (C), receipt file (R), annotations or pragmas (P), programming API (A), or programming language (L); as well as the supported language such as Java (J), Scala (S), Python (P), C++ (C), Visual C (V), R (R), Pearl (L), Bash (B), XML (X), JSON (N), YAML (Y), and OCaml (O).

Software		Features																				
		Prog.		Definition						Runtime Execution												
		Interface	Language	Model	Dependency Definition	Task Types	Dynamic	Nested	Stream Support	Parallel Execution	Load Balancing	Configurable Scheduling	Performance Analysis	Supported Environ.	Elasticity	Checkpointing	Fault tolerance	Security	Actively Maintained	License
Task-based Workflows	Airflow [21]	GA	P	D	E	A	⋆	-	-	⋆	-	-	O	Cs,Cd	A	-	Rs	⋆	⋆	1
	Aneka [22]	G	V	B	/	A	-	-	-	⋆	⋆	-	O	Cs,Cd	A	-	Rs,F	⋆	⋆	8
	Askalon [23]	GR	/	D	I	A	-		-	⋆	⋆	⋆		Cs	A	A	Rp,Rs	-	-	8
	AUTOSUBMIT [24]	R	B	D	E	U	-	-	-	⋆	-		OP	Cs	-				⋆	3
	Celery [25]	PA	P	D	I	A	⋆	⋆	-	⋆			O	Cs	A	-	-	⋆	⋆	5
	CIEL [26]	LA	JO	D	I	A	⋆	⋆	⋆	⋆	⋆	-	-	Cs	-	-	L,Rs,F	-	-	5
	COMPSs [27]	PA	JPC	D	I	A	⋆	-	-	⋆	⋆	⋆	OP	All	A	-	Rs	-	⋆	1
	Copernicus [28]	R	PX	D	E	P	-	-	-	⋆			O	Cs	-	AM	Rs,F	⋆	⋆	2
	Crunch [29]	CA	J	D	I	A	⋆	-	⋆	⋆									⋆	1
	Dask [30]	GA	P	D	I	A	⋆	⋆	⋆	⋆	⋆	⋆	OP	Cs	-	-	Rs,F	⋆	⋆	5
	EcFlow [31]	CAG	PB	D	I	A	-	-	-	⋆	-			All	-	-	Rs,F	⋆	⋆	1
	FireWorks [32]	R	PNY	D	E	A	⋆	⋆	-	⋆	-	-	OP	Cs	-	-	Rs	-	⋆	5
	Galaxy [33]	G	/	D	E	S	-	-	-	⋆	-			/	/	-	-	⋆	⋆	7
	Google MapReduce [34]	A	C	S	I	U	-	-	-	⋆	⋆	-	-	Cs	-	-	Rp,Rs,F	-	-	8
	Jolie [35]	L	/	B	/	S	-	-	-	⋆	-	/	-	All	-	-	-	-	⋆	4
	Kepler [36]	G	/	D	E	A	-	⋆	-	-	-	/	-	/	/	-	-	-	⋆	5
	Netflix Conductor [37]	R	N	D	I	S	⋆	⋆	-	⋆		-	OP	/	/				⋆	1
	Pegasus [38]	RA	JPL	D	E	A	⋆	⋆	-	⋆	⋆	⋆	OP	All		A	L,Rs	-	⋆	1
	Spark [15]	CA	JSPR	D	I	A	⋆	⋆	⋆	⋆	⋆	⋆	O	All	A	-	All	⋆	⋆	1
	Swift [39]	L	/	D	I	A			-	⋆			OP	Cs	-	A	Rs	-	⋆	1
	Taverna [40]	G	/	D	E	S	-	⋆	⋆	⋆	-	/	OP	/	/	-	Rs,F	⋆	⋆	4

Legend: ⋆ Available - Not available / Not Applicable

Table 1: *Task-based Workflows* software classified into the *Application Development* layer

Regarding the task-flow definition, the main distinction is made between the supported patterns: bag of tasks (B in the table), *skeleton* (S), or DAG (D). We have also considered whether the users must explicitly (E) or implicitly (I) define the task dependencies and classified the supported task types into only pre-defined methods (P), only services (S), only user defined methods (U), or any (A). Other interesting properties in the task-flows definition are the support for workflows that can vary during the application's execution (dynamic workflows), for nested executions, and for data streams.

When focusing on the application execution, the main difference is the framework's capability of executing task in parallel or not. However, we have also distinguished more advanced features such as load balancing techniques, the support for customisable scheduling policies built-in tools for the application's

execution analysis - online (O) or post-mortem (P) -, the capability of managing heterogeneous infrastructures - clusters(Cs), clouds(Cd), and containers (Ct)- and the resource elasticity during the application's execution time, categorising this feature into automatic (A) or manual (M).

Other interesting features for advanced users might be fault tolerance mechanisms and security. For this purpose, we have also distinguished those frameworks that provide any kind of checkpointing - automatic (A) or manual (M) -, those that provide fault tolerance mechanisms such as lineage (L) (i.e., the ability to re-generate a lost or corrupt data by executing again the chain of operations that was used to generate it), replication(Rp), re-submission(Rs) or fail-over(F), and those that provide any kind of security mechanism.

Finally, we consider that the framework's availability is a high-priority issue for application developers. Thus, we indicate whether the framework is actively maintained (active developments registered during the past year) and its license following the next nomenclature: (1 in the table) Apache 2.0 [41], (2) GNU GPL2 [42], (3) GNU GPL3 [43], (4) GNU LGPL2 [44], (5) BSD [45], (6) MIT License [46], (7) other public open source software licenses (e.g., Academic Free License v3 [47], Mozilla Public License 2 [48], Eclipse Public License v1.0 [49]), and (8) custom private license or patent.

3.1.2. Analysis

The main classification of tasks-based workflow can be done by the workflow definition category specially by the supported workflow model. Some frameworks, like Aneka or Jolie, require application users to create tasks and add them to a bag explicitly. The tasks inside the bag are then selected to be executed by the model with equal probability. In this sense, the main drawback of using a *Bag of Tasks* is that users need to handle data dependencies between tasks before introducing a new task to the bag.

Other frameworks restrict the workflow to a predefined parallelism pattern (*Skeleton* programming), such as MapReduce [34]. In this kind of models, programmers only need to specify a set of methods that compose the predefined

workflow. In contrast to the previous approach, skeleton models do handle data dependencies between tasks, but the users' application is pigeonholed into the predefined parallelism pattern.

Finally, other models go one step further by generalising the *Skeleton* model and allowing users to define Directed Acyclic Graph (DAG) of tasks. In this approach, applications are represented as DAGs, where tasks are represented by vertices, and data dependencies are represented by edges. In contrast with *Skeleton* models, DAG models allow application users to describe any kind of workflow with any custom operation. The main difference within this group of frameworks is about the way that user have to define dependencies between tasks. On the one hand, some models require to explicitly define the workflow by means of a Graphical User Interface (such as Taverna, Kepler or Galaxy), a Command Line Interface (such as Copernicus), a receipt file (such as Askalon, AUTOSUBMIT, Fireworks, or Netflix Conductor) or a language API (such as Pegasus, Apache Airflow, or ecFlow). This methodology allows users to specifically control the dependencies between the different stages and have a clear overview of how the framework executes their application but makes tedious to design complex, large workflows. On the other hand, there are programming models and languages that opt to automatically infer the workflow from the user code, e.g., Spark, COMPSs, Dask, Apache Crunch, Celery, and Swift. This workflow definition allows users to develop applications in an almost sequential manner, without explicitly handling the tasks spawned, and reducing the programming complexity to almost zero. However, the main disadvantage is that the users do not know beforehand how the framework will execute their application (for example, how many tasks will be created in a specific call). From our point of view, the frameworks using explicit task dependency definition are more suitable for small applications while frameworks using implicit task dependency definition are better for large and complex application workflows.

Another important feature to distinguish task-based framework is the supported task types, most of the frameworks support user-defined (U) or any type (A) of tasks except Copernicus which is developed for pre-defined methods and

Galaxy, Jolie, Netflix Conductor, and Taverna which are developed for services.

Regarding programming, some frameworks include support for several languages (e.g., CIEL, COMPSs, Copernicus, EcFlow, FireWorks, Pegasus, and Spark), but the rest of them only supports a single language. Hence, application developers should consider the application's language before selecting the appropriate framework. Also, it is worth mentioning that CIEL uses a custom language (Skywriting) but also provides APIs for Java and OCaml.

Finally, we are surprised by the lack of support for advanced workflow features (i.e., dynamic and nested workflows, and support for streams) and new infrastructures (mainly the cloud and containers). Although the software might still be evolving, modern applications require complex workflow features and elasticity mechanisms to automatically handle the application's resource usage (i.e., by managing the available computing resources). In this same line, we also believe that many frameworks have been designed for cluster infrastructures; which explains the lack of security mechanisms (i.e., secure communication, data encryption or user authentication). In terms of fault tolerance, while resubmission and fail-over are common techniques among all the different software, only a few of them include checkpointing (Askalon, Copernicus, Pegasus, and Swift) or lineage (CIEL, Pegasus, and Spark). We know that fault tolerance comes up with a non-negligible performance degradation but, since application runs are lasting longer and longer, we believe that this is a key feature when selecting the appropriate framework.

3.2. Dataflows

Similarly to Task-based Workflows, *Dataflows* allow the application developers to define pieces of code to be remotely executed as *tasks*; however, Dataflows build on *Data Flow Graphs (DFG)* rather than Task Dependency Graphs (TDG). On the one hand, TDGs define a task completion relation between tasks so that the only information travelling among the graph nodes is the task completion status and, thus, tasks need to share the data in a graph-independent way. On the other hand, Dataflows assume that tasks are persistent

executions with state that continuously receive/produce data values (streams) and, therefore, tasks are treated as stages or transformations that data must go through to achieve the destination. DFGs define the data path: the nodes represent stateful tasks processing the data transmitted through the graph edges.

Closely following this definition, there are platforms that are specifically built for Dataflows such as TensorFlow [17]. However, this approach is also used for *stream processing*, *real-time processing* and *reactive programming* which, for the case, are basically subsets of each other. Thus, in stream processing words, a Dataflow is a sequence of data values (*stream*) where we apply a series of operations (*kernel* functions) to each element of the stream in a pipelined fashion.

3.2.1. Taxonomy

Table 2 presents our taxonomy of the surveyed Dataflow frameworks. Regarding the programmability of the frameworks, we consider the same aspects as with the task-based workflows; however, some new programming languages – Go (G), Clojure (L), and JRuby (R) – appear on the table.

	Software	Features																					
		Prog.		Definition									Execution Runtime										
		Interface	Language	Primitive	Multi-subscriber	Ordered	Window	Buffering	Drop Messages	Back Pressure	Delivery Pattern	Stateful Operations	Process Unit	Load Balancing	Performance Analysis	Supported Environ.	Elasticity	Checkpointing	Fault-tolerance	Security	Actively Maintained	License	
Dataflows	Apex [50]	CA	J	T	⋆	⋆	S	⋆	-	-	LME	⋆	M	⋆	OP	Cs	-	A	Rs, F	⋆	-	1	
	Beam [51]	CA	JPG	B	⋆	-	TS	⋆	⋆	-	LME	⋆	MO	-		Cs	-	-	Rs, F		⋆	1	
	Cascading [52]	CA	J	T	-	-							O			Cs	-	M	-		⋆	1	
	Gearpump [53]	GCA	J	K	⋆	⋆	T	-	-	-	LE		O		OP	Cs	-	A	Rs, F		⋆	1	
	Hazelcast jet [54]	CA	J	T	⋆	⋆	TS	⋆	⋆	⋆	L		O	⋆	-	Cs,Cd	⋆	-	Rs, F	-	⋆	1	
	Heron [55]	GCA	JSP	T	⋆	⋆	TS	⋆		⋆	LME	⋆	O	-	OP	Cs	-	-	Rs, F		⋆	1	
	IBM Streams [56]	GA		T	⋆						LE	-	O		OP	Cd	-	A	F		⋆	8	
	Netflix Mantis [57]	GCA	J	S	⋆			⋆	⋆	⋆		⋆	MO		OP	Cs,Cd	⋆	-	Rs, F		⋆	8	
	Samza [58]	CA	J	M	⋆	⋆	TS	⋆	⋆	⋆	LE	⋆	O	-	-	Cs	-	A	Rs, F	-	⋆	1	
	Spark Streaming [59]	CA	JSP	D	⋆	-	TS	⋆		-	LME	⋆	M	⋆	OP	Cs	-	A	Rs, F	⋆	⋆	1	
	Storm [16]	CA	JSLR	T	⋆	-	TS	⋆	⋆	-	LM	⋆	O	M	OP	Cs,Ct	⋆	A	Rs, F	⋆	⋆	1	
	TensorFlow [17]	GCA	JPC	R	⋆						E	⋆	M	⋆	OP	Cs	-	M	Rs, F	-	⋆	1	

Legend: ⋆ Available - Not available / Not Applicable

Table 2: *Dataflows* software classified into the *Application Development* layer

Concerning application developers, we consider the main distinction between frameworks relies on the dataflow definition. To this purpose, we distinguish the stream primitive between message (M in the table), tuple (T), bolt (B), DStream (D), source (S), task (K), and tensor (R). We also distinguish between single-subscriber and multi-subscriber models, and between ordered and unordered streams. We have also categorised the windowing support between time window (T) and size window (S) for those frameworks that allow to process a group of stream entries that fall within a window based on timers or data sizes. Furthermore, we have categorised the delivery pattern into at-least-once (L), at-most-once (M), and exactly-once (E) for those frameworks that ensure that messages are never lost, never replicated or both. We have also distinguished more advanced features such as support for stateful operations, or workload management mechanisms such as buffering, message dropping, and back pressure.

Regarding the execution of the dataflow, we have considered most of the features in the previous case (see Section 3.1.1 for further details) and we distinguish the processing unit between one-record-at-a-time (O) or micro-batch (M).

3.2.2. Analysis

Although Task-based Workflows target any type of computation, stream processing has become increasingly prevalent for processing social media and sensor devices data. A large majority of the software – Apache Samza, Apache Storm, Twitter Heron, IBM Streams, Netflix Mantis, Cascading, or Apache Beam – has been explicitly developed for stream processing, what is reflected in a one-record-at-a-time (O in the table) process model. Conversely, other already-existing frameworks have evolved to include stream processing while maintaining the functionalities of the rest of their framework through the micro-batching technique (e.g., Apex, TensorFlow, Spark Streaming) or moving out from the databases environment into in-memory computation (e.g., Hazelcast).

Regarding the programming of streaming frameworks, all alternatives use

a programming API (A in the table) combined with a supporting easy-to-use Graphical User Interface (G) or Command Line Interface (C). Generally, the offered interfaces are more modern, attractive, and accessible than the ones offered by frameworks targeting Task-based Workflows, probably because the Dataflow software is newer.

A narrow minority of such frameworks offer support for several languages (e.g., Beam, Heron, Spark Streaming, Storm, and TensorFlow). As we stated for software targeting Task-based Workflows, we consider that the application developers should consider the application's language before selecting the appropriate framework.

We observe that almost every framework has its own primitive, being the tuple (T in the table) the most commonly used. Although this may not be a problem when developing applications, it hardens the portability of applications between frameworks. Regarding the stream definition, all the frameworks are multi-subscriber (except Cascading), allow the users to configure time and size windows (except Apex and Gearpump), and include buffering techniques (except Gearpump). Although all frameworks support the at-least-once delivery pattern (except TensorFlow), there is a significant variety when supporting the at-most-once, and exactly-once delivery patterns. We believe that this is a key feature to classify frameworks that application developers should consider to select the appropriate one. Most of the frameworks include stateful operations (except IBM Streams); however, only a few of them support other advanced techniques such as ordered streams (Apex, Gearpump, Hazelcast jet, Heron, and Samza), message dropping (Beam, Hazelcast jet, Netflix Mantis, Samza, and Storm) or back pressure (Hazelcast jet, Heron, Netflix Mantis, and Samza).

We are surprised by the lack of support for new infrastructures – almost none includes elasticity mechanisms– and security mechanisms. On the other hand, regarding fault tolerance, we are gratefully surprised to notice that the Dataflow frameworks are largely better than Task-based workflows.

Finally, as with task-based frameworks, we observe that most of the frameworks (except IBM Streams, and Netflix Mantis) are available through different

public open licenses and are supported by large user communities.

3.3. Graph Processing

Many people may think about Big Data as a huge amount of unstructured data. However, nothing could be further from the truth, as data always presents some sort of structure or relationship. Based on these relationships, there are many representation schemes suited to handle different types of data. Within these representations, Graphs (also known as Networks) are ubiquitous to represent business models, event chains, relationships, etc. The proof is that many tech-giants such as Google, Facebook, LinkedIn, and PayPal are currently using graph databases. Within this context, Graph Processing emerges as the way to process such databases by keeping the vertices and edges on the machine that performs the computation and reducing the communication to only control messages.

It is obvious that many general-purpose *task-based* and *Dataflow* frameworks (e.g., MapReduce, Microsoft Dryad, Pegasus, COMPSs, Apache Spark, or Storm) are capable of managing graph databases. However, general-purpose frameworks are essentially functional, which forces the application developer to express a graph algorithm as a chain of functions (e.g., a chained MapReduce) that requires passing the entire state of the graph from one stage to the next (increasing the serialisation and the communication between computational nodes). Moreover, coordinating the steps of the chain adds a programming complexity that is avoided by Graph Processing frameworks (e.g., Pregel's iteration over super-steps). In this section, we focus on specialised distributed graph processing frameworks that are tuned for this kind of computations and provide specific tools to application developers to work with graph processing.

3.3.1. Taxonomy

Table 3 presents our taxonomy of the surveyed Graph Processing frameworks. As with Dataflow frameworks, we classify the frameworks according to

Software		Features															
		Prog.		Definition						Execution Runtime							
		Interface	Language	Model	Flow Type	Cut Type	Sync/Async	Dynamic	Nested	Load Balancing	Supported Environ.	Elasticity	Checkpointing	Fault-tolerance	Actively Maintained	License	
Graph Processing	Apache Hama [60]	AC	J	VMD	P	E	S	-	-	-	Cs, Cd	-	A		-	1	
	CombBLAS KDT [61, 62, 63]	A	P	M	P	V	S	-	-	-	Cs	-	-	-	-	5	
	Distributed R [64]	AC	R	M	P	E	S	-		⋆	Cs	-	A	-	-	2	
	Giraph [65]	A	J	V	P	E	S	⋆			Cs	-	M	Rs,F	⋆	1	
	GoFFish [66, 67]	AC	J	G	L	E	A	-		-	Cs, Cd	-	-		-	8	
	GraphX [68, 69]	AC	J	G	P	V	S	-	⋆	⋆	Cs, Cd	-	A	F	⋆	1	
	Graph Engine (Trinity) [70, 71, 72]	AG	C	G	B	/	B	-		⋆	Cs, Cd	⋆	A	F	⋆	6	
	GPS [73, 74]	AC	J	V	P	E	S	⋆		⋆	Cs	-	A		-	5	
	Imitator [75, 76]	AC		V	P	E	S	-		-	Cs, Cd	-	-	Rs,F		1	
	Parallel BGL [77, 78]	A	C	GI		E	B	-	-	-	Cs	-	-	-	-	8	
	PowerGraph [79]	A	C	V	B	V	B	-		-	Cs	-	A	-	-	1	
	PowerLyra [80, 81]	A	C	V	L	V	S	-	-	-	Cs, Cd	-	A	-	-	1	
	Pregel [82]	A	J	V	P	E	S	⋆		-	Cs	-	A		-	8	
	Pregelix [83, 84]	A	J	V	P	E	S	⋆	-	-	Cs	-	A	F	-	1	
	Presto [85]	AC	R	M	P	E	S	-		⋆	Cs	-	A	-		8	
	Seraph [86]	A		V	P		S	⋆	-	-	Cs, Cd	-	A	Rs	⋆		
	STAPL Graph Lib [87, 88]	A	C	GVI	L	E	B	⋆	-	-	Cs	-	-	-	⋆		
	Titan Hadoop (Faunus) [89]	AC	G	V	P	E	S	⋆		⋆	Cs	-	-	Rs,F	⋆	1	
	Turi Create (Distributed GraphLab) [90, 91]	A	P	V	B	E	B	-	-	-	Cs	-	A	-	⋆	5	

Legend: ⋆ Available - Not available / Not Applicable

Table 3: *Graph Processing* software classified into the *Application Development* layer

the programmability features presented in Section 3.1.1; however, in this case, the new programming languages are R (R), and Gremlin (G).

As noticed by N. Doekemeijer et al. [92], one of the main differences between frameworks is the graph model which can be categorised into DAG (D in the table), matrix (M), vertex-centric (V), graph-centric (G), and visitor models (I). We have also categorised the flow type - between push (P), pull (L), and both (B) -, and the cut type - between edge (E), and vertex (V) -. Moreover, we have evaluated if the frameworks are synchronous (S), asynchronous (A) or both (B); and whether they support (or not) dynamicity and nesting in their computations.

When focusing on resource management, fault tolerance mechanisms, active maintenance, and licensing, we have considered the same features than in the previous cases (see Section 3.1.1 for further details). In contrast with the previous cases, we have not included security features because none of the analysed

frameworks provide them. To the best of our knowledge, this is because the frameworks are mainly cluster-based and delegate security to the underlying storage (usually, a specific Graph Database).

3.3.2. Analysis

The basis of Graph Analytics is the Bulk Synchronous Parallel model [93]. The high-level organisation of BSP programs consists of a sequence of iterations, called super-steps. During a super-step, the framework parallelly invokes a user-defined function for each vertex (or edge). The user-defined function itself can modify the vertex (or edge) state, send messages to any other vertex (or edge) that will be received during the next super-step, receive messages from other vertexes (or edges) that were sent in the previous super-step, or even modify the graph's topology. Also, all the vertexes (or edges) can vote to halt; terminating when a consensus is reached.

Demonstrating the interest for Graph Analytics, Google's Pregel, its open-source version Apache Giraph, and Microsoft's Graph Engine (previously known as Trinity) have adopted the BSP model as its major technology. On its own, GraphLab has developed several frameworks during the past years; such as PowerGraph or PowerLyra. Its attempts have finally evolved into Turi Create, which is based on Distributed GraphLab. Finally, many open-source projects also focus on efficiently processing large graphs; such as Apache Hama and Apache Spark's GraphX.

There does not seem to be a consensus on the Graph Definition neither regarding the computational model, the cut and flow types nor the synchronicity of the execution. The vertex-centric model (V in the table) is the most popular because it easily distributes graph. However, it is difficult to estimate the performance degradation of this computational model when executing arbitrary algorithms since the efficient implementation is still up to the user. To our belief, the other differences are due to the fact that the optimal parameters also vary from application to application, so frameworks try to cover the maximum features as possible and let the application developers select the optimal ones.

In contrast to general-purpose Task-based Workflows and Dataflows, Graph Processing software has neither invested nor in updating neither its interfaces nor its platform technologies. None of the proposed frameworks (except Microsoft's Graph Engine) has a GUI to develop, monitor, or analyse the application; less than half of them support cloud technologies and none of them supports containers or dynamic elasticity. Conversely, they are strongly concerned about fault tolerance mechanisms and automatic checkpointing.

Finally, as general-purpose frameworks, we observe that most of the software (except GoFFish, Pregel, and Presto) are available through different public open licenses.

4. Platform

Instead of providing an all-in-one distributed computing solution, software within the *Platform* layer focuses on easing the application development by resolving a single computing challenge. Although this may seem much more straightforward than *Application Development* frameworks, it actually translates in highly customisable tools and frameworks that provide a single solution for many underlying infrastructures. Moreover, this type of software no longer targets simple high-end application developers, but application developers with high knowledge about their underlying infrastructure and/or about the data structures of their application.

The combination of these two facts leads to *Platform* tools and frameworks with highly customisable features and huge configuration files that allow application users to fine-tune them for their specific application requirements. Thus, *Platform* software is also hard to maintain because of the variability of the existing and the new underlying technologies.

As shown in Figure 3, we have divided this layer into two categories depending on the abstraction target, i.e., data sharing and resource management. Next subsections delve into the latest software within each.

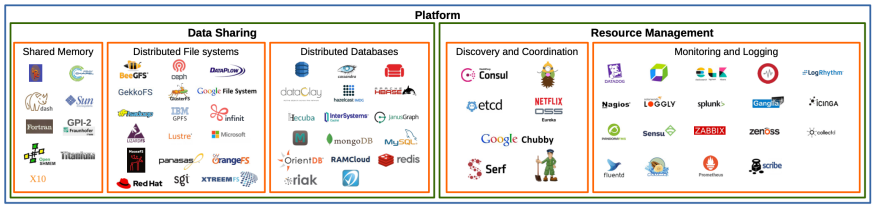

Figure 3: Platform Layer

4.1. Data Sharing

When running parallel applications, sharing data among execution threads is already a nightmare. The users need to care about the data consistency to ensure that each process retrieves the correct value of the data, and that every process is aware of the data updates. This may seem trivial, but most of the accesses require synchronisation between processes which finally leads to noticeable slowdowns if not treated correctly.

When running distributed applications, we do not only need to handle data consistency but also data transfers. Since data must be sent through networks, transfers must be correctly scheduled to avoid harming the performance. There is a large number of algorithms, patterns, and solutions for efficient data transfers and synchronisations but this section does not intend to cover such aspects but rather the models that help users to share data between the processes of their application.

For this purpose, we have categorised the models considering the different ways of sharing data among distributed processes. Next subsections provide further information about models developed to share memory, files, and data structures.

4.1.1. Distributed Memory

Distributed memory architectures are formed by a set of processors with its memory and some sort of interconnection between them. Since each processor has its own private memory and communicates with others to retrieve remote data, the network topology impacts directly on the system's performance.

4.1.1.1. Taxonomy.

Table 4 presents our taxonomy of the surveyed distributed memory approaches. First, we have categorised the view model into global (G) and fragmented (F) since global view models focus on the whole system and fragmented view models are programmed at process level. Next, we have categorised the memory model into Partitioned Global Address Space (PGAS) [112], Asynchronous PGAS (APGAS) [113], and GASPI [114, 115, 116]. On the one hand, PGAS models provide a simpler, shared memory-like programming model, where the address space is partitioned, and the programmer has control over the data layout. However, PGAS models lack asynchrony since they focus on homogeneous execution contexts and the single program multiple data threading model. On the contrary, Asynchronous PGAS (APGAS) [113] programming models provide mechanisms for asynchronous execution while following the PGAS memory model. Specifically, they allow creating a computational unit that can run in parallel with the main program (called activity) and return immediately. Regarding the model, we have also categorised its language into C (C), C++ (C++), Fortran (F), Java (J) or custom (⋆), indicated whether

Software		Features														
		Model					Architecture				F. T.					
		View Model	Memory Model	Language	RDMA	External interoperability	Strong Types	Object-Oriented	Automatic DT Distribution	Reduction Operations	Node Failure	Data Failure	Actively Maintained	License		
Distributed Memory	(Berkeley) Unified Parallel C [94, 95, 96]	F	PGAS	C	⋆	/	-	-	-	⋆	-	-	⋆	5		
	Chapel [97, 98]	G	PGAS	⋆	⋆	⋆	⋆	⋆	⋆	-	⋆	-	⋆	1		
	DASH [99, 100]	F	PGAS	C++	⋆	/	⋆	⋆	-	⋆			⋆	5		
	Fortress [101]	G	PGAS	⋆		-	⋆	⋆	⋆	⋆	-	-	-	5		
	(GNU) Co-Array Fortran [102]	F	PGAS	F	⋆	/	⋆	-	⋆	-	-	-	-	3		
	GPI-2 [103]	F	GASPI	C++	⋆	/	⋆	⋆	-		-	⋆	⋆	3		
	OpenSHMEM [104, 105]	F	PGAS	C,F	⋆	/	⋆	-	-	-	-	-	⋆	5		
	Titanium [106, 107, 108]	F	PGAS	J	⋆	/	⋆	⋆	-	⋆			⋆	8		
	X-10 [109, 110, 111]	G	APGAS	⋆	⋆	⋆	⋆	⋆			⋆	-	⋆	7		

Legend: ⋆ Available - Not available / Not Applicable

Table 4: *Distributed Memory* software classified into *Data Sharing* box at the *Platform* layer

the software supports Remote Direct Memory Access (RDMA [117]), and indicated whether the systems can operate with external frameworks. This last point is particularly interesting to decouple the application language and the distributed-memory-solution language.

Regarding the architecture, we have evaluated which systems are object-oriented. Moreover, we have focused on how the systems treat data; distinguishing if they can support strong data types, automatically distribute strong data types, and support reduce operations. Also, regarding fault tolerance, we have evaluated whether the systems can tolerate node or data failures.

Finally, we have also included the maintenance and license considering the same features than in the previous cases (see Section 3.1.1 for further details).

4.1.1.2. Analysis.

First, we have selected Berkeley UPC and GNU CAF as reference implementations of the Unified Parallel C (UPC) and Co-Array Fortran (CAF) models, respectively.

Second, there seems to be a de-facto standard regarding the model since most of the surveyed alternatives are PGAS with fragmented view (F in the table) and RDMA support. UPC, CAF, and OpenSHMEM are built on top of GASNet [118, 119]; a language-independent networking middleware that provides network-independent, high-performance communication primitives including Remote Memory Access (RMA) and Active Messages (AM). On the other hand, global view models are very limited since Chapel and Frotress are the only PGAS alternatives, and X-10 is the only APGAS alternative. Also, these three alternatives options force applications to use a custom language although Chapel and X-10 provide support for external interoperability. Similarly, GPI-2 is the only GASPI model and provides a fragmented view model.

Finally, all the solutions seem quite poor concerning the advanced features and fault tolerance since only Chapel and X-10 provide mechanisms for node failures, and GPI-2 provides mechanisms for data failures. Moreover, we must highlight that all the programming models (except Titanium) are available through

different public open licenses.

4.1.2. Distributed File Systems

Distributed File Systems are used to share files partly or as a whole on different nodes via a computer network. Generally, files are shared transparently, and its locations are managed by the system so that the users are not aware of where the files are really stored. Their main goal is to allow IO scale proportionally to the number of servers.

4.1.2.1. Taxonomy.

Table 5 presents our taxonomy of the surveyed distributed file systems. First of all, we have focused on the system's architecture. We have distinguished between POSIX and non-POSIX compliant systems. Although the majority

Software		Features													
		Architecture				Redundancy			F. Tolerance			Security			
		POSIX Compliancy	Model	Logic	Storage Scale	Data Replication	Meta-data Replication	Erasure Codes	Node Failure	Master Failure	Data Integrity	Encryption in Transit	Encryption at Rest	Actively Maintained	License
Distributed File Systems	BeeGFS [120, 121]	⋆	DC	F	⋆	⋆	⋆	-	-	/	-	-	-	⋆	2
	CephFS [122]	⋆	D	FBO	⋆	⋆	⋆	⋆	⋆		⋆	-	-	⋆	4
	DataPlow SFS [123]	⋆	D	F	⋆	⋆	/		⋆	/	⋆	-	-		8
	GekkoFS [124, 125]	⋆	C	F	⋆			-	⋆	-	-	-	-	⋆	6
	Gluster FS [126, 127]	⋆	DC	F	⋆	⋆	/	-	⋆	/	⋆	⋆	-	⋆	3
	Google FS (GFS) [128]	-		O		⋆			⋆	⋆	⋆	⋆	-	⋆	8
	Hadoop FS [129]	-	D	F	⋆	⋆	-	-	⋆	-	⋆	-	-	⋆	
	IBM GPFS [130]	⋆	C	B	⋆	⋆	⋆	⋆	⋆	/	⋆	-	⋆	⋆	8
	Inifinit [131]	⋆	DC	FBO	⋆	⋆	⋆	-	⋆	/	⋆	⋆	⋆	-	8
	LizardFS [132]	⋆	D	F	⋆	⋆	⋆	-	⋆	⋆		-	-	⋆	3
	Lustre [133]	⋆	D	F	⋆	-	⋆	-	⋆	⋆	-	-	-	⋆	2
	Microsoft Cluster Shared Volumes (CSV) [134]	⋆		F	⋆	⋆			⋆				⋆	-	8
	MooseFS [135]	⋆	D	F	⋆	⋆	⋆	-	⋆	⋆	⋆	-	-	⋆	2
	Panasas ActiveScale File System [136, 137]	⋆	D	F	⋆	⋆	⋆	⋆	⋆	⋆				⋆	8
	PVFS2 (OrangeFS) [138, 139]	⋆	D	O	-	⋆	⋆	-	⋆	⋆		-	-	-	4
	Red Hat Global File System2 [140, 141]	⋆	DC	B	⋆	⋆	⋆	⋆	⋆	/	-	-	⋆	⋆	2
	SGI CXFS [142]	⋆	D	F	⋆	⋆	⋆	-	⋆	⋆	⋆	-	-		8
	XtreemFS [143]	⋆	D	FO	⋆	⋆	⋆	-	⋆		⋆	⋆	-	⋆	5

Legend: ⋆ Available - Not available / Not Applicable

Table 5: *Distributed File Systems* software classified into *Data Sharing* box at the *Platform* layer

are POSIX compliant to benefit from its uniform programming interfaces and its portability among a large family of Unix derivative operating systems, some alternatives trade-off some POSIX requirements for performance.

Next, the model represents the way the different servers and clients are arranged in a storage system and has a direct impact on the infrastructure's performance, scalability, redundancy, and fault tolerance. Centralised (C) systems store both the data and the meta-data on a single server, thus having a simple design but a single-point of failure. Distributed (D) storage systems follow a master-slave paradigm to distribute the data blocks between the different storage servers, thus having a more complex infrastructure but bigger capacity, scalability, and resiliency. On the other hand, decentralised (DC) systems distribute the data, meta-data, and requests between all the nodes that have the particularity of being equally unprivileged. Notice that the master-slave paradigm can suffer bottlenecks, single-point of failures, performance degradations, and cascading effects when having too many requests to process. The resulting systems are usually better in performance and scalability but noticeably more complex.

Next, we have categorised the way distributed file systems store data. Object (O) storages are often used for storing application-specific data. File (F) storages represent data in files organised in folders. Block (B) storages manage data as blocks within a virtual raw partition that can be individually controlled and formatted. Regarding the system's architecture, we have also evaluated whether the systems can dynamically scale the storage capacity over time without interruption.

We have also evaluated redundancy in terms of replication and erasure codes. Replicating data and meta-data ensures that the system can keep operating even if some data has been lost (e.g., because of a server failure or data corruption). On the other hand, erasure codes store some additional symbols of each data block to eventually help error-correcting corrupted blocks. Regarding fault tolerance, we have evaluated whether the systems can tolerate node and master failures and data integrity. In this context, node failures means that the system

can tolerate nodes going down and re-accepting new ones without rebooting. Also, data integrity means that the system can tolerate invalid data on a node that will be automatically erased or updated when required.

Furthermore, regarding security, we have evaluated whether the systems can encrypt the data in transit (between clients and servers) or at rest (once stored on a server). Although many systems do not encrypt data at all because it is usually stored in private clusters, this could be a key-point for applications containing sensitive data.

Finally, we have also included the maintenance and license considering the same features as in the previous cases (see Section 3.1.1 for further details).

4.1.2.2. Analysis.

When talking about data, users usually value stability over speed and tend to choose file-systems supported by the credit of big companies such as IBM GPFS, Google GFS, Hadoop FS, Red Hat GFS2, Microsoft CSV, DataPlow SFS, or SGI CXFS. However, there are many other solutions that differ significantly in cost, performance, stability, and implementation. Although these solutions might be harder to set up, diagnose, and repair, they also have a large and active community. For instance, Gluster FS, CephFS, and Lustre are three open-source options with a proven good performance. Panasas ActiveScale File System (PanFS) is a young commercial solution based on the CMU NASD research. MooseFS is designed similarly to Google GFS, Lustre or CephFS with multiple meta-servers and multiple chunk servers. LizardFS was released as a fork of MooseFS and is now an independent project. XtreemFS is also an open-source project based on a distributed architecture using the paxos model. BeeGFS has a similar architecture to Lustre although it has a single meta-data server and it is not really an open-source project. PVFS2 (Orange FS) is a growing project that provides an easy to configure solution but still lacks a solid community. Finally, GekkoFS is a highly-scalable burst buffer file system specifically optimised for data-intensive HPC applications.

Analysing the alternatives in-depth, the majority of distributed file systems

are POSIX compliant to benefit from its uniform programming interfaces and its portability. Only Google GFS, and Hadoop FS have trade-off some POSIX requirements for performance. Regarding the model, Distributed models are the most common because of the trade-off between scalability and complexity. However, Red Hat GFS2, Inifinit, Gluster FS, and BeeGFS are based on the decentralised model, and GPFS still uses the centralised approach. On the other hand, regarding the system logic, object storages are only useful for application-oriented systems such as Google FS, OrangeFS, and, optionally, XtreemFS. File storages seem to be the best option, or at least the most common one, although Red Hat GFS2, IBM GPFS, and, optionally, CephFS, and Infinit use Block storage.

The taxonomy also identifies the ability to dynamically scale storage, to replicate data, and to recover from node failures as the most important industry requirements. Similarly, most of the systems provide some sort of redundancy, and fault tolerance but do not provide security since distributed file systems are commonly used in clusters with restricted access.

Finally, notice that only half of the options are open-source although the vast majority requires a paid plan to obtain the necessary support for installation, customisation, and maintenance. Since users cannot freely try different distributed file systems, we recommend to make a preliminary effort to identify the requirements and gather information about the suitable systems before choosing the right option.

4.1.3. Distributed Databases

Many databases have evolved to distributed APIs because the data has grown enough to not fit within a single node and because many compute nodes require to access data concurrently. In general terms, distributed databases store data among several data nodes and accept queries from different processes, ensuring persistence, consistency, coherency, and, in some cases, replication, and fault tolerance.

4.1.3.1. Taxonomy.

Table 6 presents our taxonomy of the surveyed distributed databases. From the users' point of view, the key differences rely on the database model and the data scheme. Hence, the first column categorises the database model between document (D in the table), key-value (K), wide column (C), graph (G), object (O), and tabular (T). The second column details whether it is SQL or NoSQL, the third column distinguishes whether the data scheme is free (F) or fixed (D), and the fourth column categorises the implementation language: Java (J), Python (P), Erlang (E), Scala (S), C, or C++.

Regarding the data types, all the options support storing basic types. Thus, we have only distinguished advanced features such as support for complex types, foreign keys, and in-memory structures. Regarding built-in data operations, we have evaluated whether the software includes (or not) support for server-side operations; differentiating when possible between the supported programming

Software		Features																	
		General				Data			Storage Ops.		F. T.		Query Support						
		Model	SQL/NoSQL	Free/Fixed Scheme	Language	Complex Types	Foreign Keys	In-memory structures	Server-side	Triggers	Replication	Partitioning	Full Search	Secondary Indexes	MapReduce	Atomic Operations	Actively Maintained	License
Distributed Databases	Amazon DynamoDB [144]	D,K	-	F		⋆	-	-	-	⋆	C	S	⋆	⋆	-		⋆	8
	Cassandra [145]	C	-	F	J	⋆	-	-	-	⋆	C	S	⋆	⋆	⋆	⋆	⋆	1
	CouchDB [146, 147]	D	-	F	E	-	-	-	⋆	⋆	M,S	S	⋆	-	⋆		⋆	1
	DataClay [148, 149]	O	-	F	J	⋆	-	⋆	⋆	-	C	-	⋆	-	⋆	⋆	⋆	5
	Hazelcast IMDG [150]	K	-	F	J	⋆	-	⋆	⋆	⋆	C	⋆	⋆	⋆	⋆	⋆	⋆	1
	Hbase [151, 152]	C	-	F	J	-	-	-	⋆	⋆	C	S	⋆	-	⋆		⋆	1
	Hecuba [153, 154, 155]	O,C	-	F	P	⋆	-	⋆	-	⋆	C	S	⋆	⋆	⋆	⋆	⋆	1
	Intersystems Cache [156]	K,O,R	-			⋆	⋆	⋆	⋆	⋆	S	-	⋆	⋆	-		-	8
	JanusGraph [157] (prev. Titan [158])	G	-	D	J	⋆	⋆		⋆	⋆	C	⋆	⋆	⋆	⋆		⋆	1
	Memcached [159]	K	-	F	C	-	-		-	-	-	-	⋆	-	-		⋆	5
	MongoDB [160, 161]	D	-	F	C++	⋆	-	⋆	JS	-	S	S	⋆	⋆	⋆		⋆	7
	MySQL [162, 163]	T	⋆	D	C,C++	⋆	⋆	⋆	⋆	⋆	S,M	-	⋆	⋆	-	⋆	⋆	2
	OrientDB [164, 165]	D,G,K		F	J	⋆	⋆		J,JS	⋆	M	S	⋆	⋆	-		⋆	1
	RAMCloud [166, 167]	K	-	F	C++	-	-	⋆	-	-	⋆	-		⋆	-	⋆	-	7
	Redis [168, 169]	K	-	F	C	⋆	-	⋆	L	-	S,M	S	⋆	⋆	-	⋆	⋆	5
	Riak [170]	K	-	F	E	-	-	-	JS,E	⋆	C	S	⋆	⋆	⋆		⋆	1
	Virtuoso [171]	T	⋆	D	C	⋆	⋆	⋆	⋆	⋆	S,M	⋆	⋆	⋆	⋆	⋆	⋆	2

Legend: ⋆ Available - Not available / Not Applicable

Table 6: *Distributed Databases* software classified into *Data Sharing* box at the *Platform* layer

languages: LUA (L), JavaScript (JS), Java (J), or Erlang (E). Also, we have detailed whether the software supports triggers or not.

Regarding fault tolerance, we have evaluated whether the systems provide replication from one database server to one or more database servers (S in the table), with multiple concurrent masters (M) that allow data to be updated by any member of the group, or with a custom replication scheme (C). Also, we have evaluated the support for any kind of partitioning ($\star$ in the table), sharding (horizontal partitioning with some enhancements, S in the table), or none (-).

Regarding queries, we have analysed whether the different systems have built-in support for full search, secondary indexes, map-reduce operations, and atomic operations.

Finally, we have also included the maintenance and license considering the same features than in the previous cases (see Section 3.1.1 for further details).

4.1.3.2. Analysis.

The data model is the most distinguishing characteristic of a database since it describes how the data is stored inside the database. Some options, such as MySQL or Virtuoso, use the traditional tabular structure (SQL and SQL-Like databases); relying on an established and well-known standard, and a large set of mature tools to work with. The key feature of SQL databases is the atomicity of operations; meaning that when an operation involving several entries is executed, either all the results are updated on the database (commit), or none of them is modified (roll-back).

On the other hand, NoSQL databases offer a more flexible data scheme. As shown in the taxonomy, the lack of preference regarding data model portrays the flexibility required by the applications. For instance, the most popular schemes are key-value (e.g., Memcached, RAMCloud, Redis, Riak), document (e.g., Amazon DynamoDB, MongoDB, CouchDB), wide column (e.g., Cassandra, HBase), graph (e.g., JanusGraph, the open-sourced version of Titan after its decommission), or object (e.g., DataClay, Hecuba). Also, we highlight that NoSQL databases are increasingly used in big data applications due to its sim-

pler design and better horizontal scaling, although they can compromise data consistency (many of them do not provide atomicity) and lack of an established standard.

More in-depth, while complex types, and storage operations are widely supported, only a few options support foreign keys (i.e., Intersystems Cache, JanusGraph, OrientDB, SQL and Virtuoso). Similarly, providing master-slave replication is the most common approach, since handling multiple concurrent masters requires a complicated architecture and might lead to huge overheads. Moreover, sharding is the preferred method to partition and distribute data across multiple machines and keep the horizontal and vertical scalability.

Furthermore, as previously stated, one of the major issues when using NoSQL databases is handling atomicity. Notice that only 3 NoSQL databases (i.e., Cassandra, Hazelcast IMDG, and Redis) provide support for atomic operations. Similarly, Hecuba and dataClay guarantee atomic object accesses. Regarding the rest of the options, the users can only rely on eventual consistency where changes are eventually propagated to all nodes and queries might return outdated data or might not return updated data immediately. Apart from atomic operations, almost all the databases provide support full search queries and secondary indexes, and half of them provide built-in support for map-reduce operations.

Finally, most of the software (except Amazon DynamoDB and Intersystems Cache) are available through different public open licenses; which allows developers to try different possibilities before choosing the right database for their application. However, in many cases, extended features and product support are provided through paid licenses.

4.2. Resource Management

Resource management is the other big distributed computing challenge at platform level. When executing distributed applications, we must allocate, initialise, coordinate, and deallocate the computing resources. Although the only requirement is to initialise the computing resources at the start of the execu-

tion and terminate them at the end of the execution, some frameworks provide elasticity mechanisms to dynamically allocate and deallocate resources at execution time. Also, during the application's execution, resources need to be coordinated and monitored so that the framework (and the users) can decide where to submit the computation jobs.

Since the *Resource Management* comprises several aspects, next subsections provide further information about *Discovery and Coordination*, and *Monitoring and Logging*.

4.2.1. Discovery and Coordination

Avoiding configuration conflicts becomes more and more handy as applications grow in terms of the number of resources and services. If this were not complicated enough, automatic scaling to optimise the use of the resources makes it even harder. Hence, software aiming at resource discovery and coordination need to transparently discover, configure, and coordinate the different resources and services of the system. Due to the heterogeneity of the current deployments, these frameworks are typically huge; with several connectors to handle machines and services in different computing infrastructures (see Section 6 for more details about infrastructure managers).

4.2.1.1. Taxonomy.

Table 7 presents our taxonomy for the resource discovery and coordination software. There are already many online comparisons about reference software [181, 182, 183], however, our taxonomy includes other alternatives and provides an homogeneous comparison between all of them. Since not all the frameworks provide a complete solution, the first three columns indicate whether the software can be used (or not) for resource discovery, coordination, and configuration, respectively. This information is particularly relevant to pre-select the software that meet the users' requirements.

Next, we analyse the systems' model in-depth. The client language indicates whether the users must use an API (A in the table) or and SDK (S).

Software		Features																	
		Purpose			Model					Protocols		F.T.	Security						
		Discovery	Coordination	Configuration	Client Language	Server Language	System Architecture	Data Architecture	Multiple DC	Consitency	Consensus	Health Check	Encryption	Authentication	ACL	Actively Maintained	License		
D. and C.	Consul [172]	⋆	⋆	⋆	A	Go	C	KV	⋆	W,S,E	R	G	⋆	⋆	⋆	⋆	7		
	doozerd [173]	⋆-	⋆	⋆	A	Go	C	KV	-	S	P	H	-	⋆	-	-	6		
	etcd [174]	⋆-	⋆	⋆	A	Go	C	KV	-	S	R	H	⋆	⋆	-	⋆	1		
	Eureka [175]	⋆	-	-	S	Java	C	/	⋆-	W	/	H	⋆	⋆		-	1		
	Google Chubby [176, 177]	-	⋆	⋆	S	C++	C	F	-	S	P	K			⋆		8		
	Serf [178]	⋆	-	⋆	S	Go	D	/	⋆	E	/	G	⋆	⋆	⋆	⋆	7		
	Zookeeper [179, 180]	⋆-	⋆	⋆	S	Java	C	KV	-	S	M	K	⋆	⋆	⋆	⋆	1		

Legend: ⋆ Available - Not available / Not Applicable

Table 7: *Discovery and Coordination* software classified into *Resource Management* box at the *Platform* layer

While APIs do not require any deployment, SDKs must be built on the client machines; providing a more complete environment but adding an extra complexity in its deployment. Also, we have categorised the systems' architecture between client-server (C) and distributed (D) because it can have a direct impact on the consistency and the scalability. Similarly, we have categorised how the data is stored into key-value (KV) or file-like (F). Finally, we have evaluated the capability of the systems to handle multiple infrastructures.

Regarding the protocols, we have categorised the level of consistency into weak (W), strong (S), and eventual (E). While weak consistency only ensures that the accesses to synchronisation variables are seen in the same order and that the set of accesses between two synchronisation points is the same in every process, strong consistency ensures that all accesses are seen by all parallel processes in the same order. However, strong consistency comes with a high cost in terms of performance. On the other hand, eventual consistency focuses on high availability, only guaranteeing that, if no new updates are made, all accesses will eventually return the last updated value. Furthermore, we have indicated the consensus algorithm into Raft (R), majority (M), and Paxos (P). Notice that consensus in a distributed system means agreeing on one result among a group of unreliable participants. Such consensus can be achieved by

a simple majority, although more complex algorithms can provide higher reliability and performance. For instance, Paxos [184] is a widely used family of consensus algorithms that make various trade-offs between assumptions about the processors, participants, and messages in a given system. On the other hand, Raft [185] is developed as a more understandable alternative to Paxos with equivalent performance and fault-tolerant guarantees.

Regarding fault tolerance, we have evaluated the resource health check mechanisms in each system. Heartbeat (H in the table) ensures that a shared resource is present at most in one place. Keepalive (K) ensures that a shared IP address is present in at least one place. More complex algorithms also exist, like Gossip [186], that spreads the information among the system in a manner similar to the spread of a rumour among office workers or a virus in a biological community.

Regarding security, we have indicated the capability of the systems to encrypt communicated data, authenticate users, and define Access Control Lists (ACL).

Finally, we have also included the maintenance and license considering the same features than in the previous cases (see Section 3.1.1 for further details).

4.2.1.2. Analysis.

In general terms, all the resource discovery and coordination tools are based on similar principles and architecture since they require a quorum to operate, implement some level of consistency, and rely on some sort of key-value store. Consul, etcd, and Zookeeper are the reference software that provide a complete solution. On the one hand, Consul is a strongly consistent data store built for service discovery that is the only alternative providing built-in mechanisms for service discovery (the other options are annotated with $\star$- in the table since they implement service discovery but not as a built-in mechanism). On the other hand, etcd is a simple yet powerful key-value store accessible through HTTP that provides features for hierarchical configuration and service discovery. Also, ZooKeeper is one of the most mature options providing robustness and feature richness, but being more complex to build and setup. Finally, re-

garding the rest of the surveyed alternatives, doozerd also provides a complete solution while Google Chubby does not provide service discovery, and Eureka and Serf provide built-in mechanisms for service discovery but are not designed for resource coordination and configuration.

More in-depth, most of the solutions use client-server architecture, provide data encryption, and feature some kind of authentication. However, there is no clear consensus regarding the client language since client APIs provide simple deployments and SDKs provide more complete environments. Furthermore, we highlight the lack of support for multiple data centres (only provided by Consul and Serf) and we believe that software within this layer will evolve to support the heterogeneity of the current applications and deployments.

Also, strong consistency is the most used consistency level, although Eureka implements only weak consistency, Serf implements only eventual Consistency, and Consul allows users to choose the consistency level (strong, weak, or eventual). There is much more heterogeneity regarding the consensus algorithm, where Consul and etcd opt for modern raft-based algorithms, and doozerd and Google Chubby for more traditional paxos-based algorithms. On its own, ZooKeeper relies on a simple majority algorithm. Similarly, regarding fault tolerance, while Consul and Serf use the Gossip algorithm (they are maintained by the same company), Google Chubby and Zookeeper rely on the Keepalive mechanism and the rest use heartbeats.

Finally, we must highlight that all frameworks (except Google Chubby) are available through different public open licenses.

4.2.2. Monitoring and Logging

Due to the increasing size of systems, networks and infrastructure, it is no longer viable to monitor the status of all the resources manually. Instead, many solutions arise to monitor a system and/or collect and analyse its logs. This section includes software designed only for resource monitoring, software designed to collect and analyse data, and full-stack frameworks that offer resource monitoring, log analytics, and interfaces to visualise the data.

4.2.2.1. Taxonomy.

Table 8 presents our taxonomy of the surveyed monitoring and logging software. Many comparisons are available for monitoring tools [215, 216] but neither compare all the features, nor all the software that we are comparing in this taxonomy. Also, many online comparisons have been made between ELK, Splunk, and Graylog [217], Nagios and Zabbix [218], and Fluend and Logstash [219]. Although we have included many of this information in our taxonomy, the readers may find more in-depth details since they are only reviewing 2 or 3 alternatives.

In our taxonomy, we first describe the architecture of each system, detailing the implementation language: Java (J in the table), C, C++, Python (P),

	Software	Features																			
		Architecture			Monitored Elements						Capabilities						Security				
		Language	Database	Web Interface	Application	Node	Network	Cloud	Container	Database	Automatic Discovery	Agent Monitoring	Agentless Monitoring	Custom Metrics	Alerts	Custom/Aggregated Graphs	Data Encryption	Authentication	Actively Maintained	License
All	DataDog [187]		M	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	-	⋆	⋆	⋆	-	⋆	⋆	8
	Dynatrace [188, 189]	J	M	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	-	⋆	⋆	⋆	-	⋆	⋆	8
	ELK Stack (Elastic) [190]	J	M	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	-	⋆	⋆	⋆	⋆	⋆	⋆	1
	Graylog [191]	J	⋆	⋆	⋆	⋆	⋆	⋆	⋆		⋆			⋆	⋆	⋆	-	⋆	⋆	3
	LogRhythm [192, 193]		⋆	⋆	⋆	⋆	⋆	⋆	-	-	/	/	/	⋆	⋆	⋆	-	⋆	⋆	8
	Nagios [194, 195]	C	M	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	-	⋆	⋆	2
	New Relic [196]		M	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	-	⋆	⋆	⋆	-	⋆	⋆	8
	Solarwinds APM Suite [197]		⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆		-	⋆	⋆	⋆	⋆	⋆	⋆	⋆	8
	Splunk [198, 199]	C++,P	O	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	-	⋆	⋆	⋆	-	⋆	⋆	1
Monitoring	Ganglia [200, 201]	C	-	⋆	-	⋆	⋆	⋆		-	⋆	⋆	-	⋆	-	⋆	-	⋆	-	5
	Icinga 2 [202]	C++	⋆	⋆	⋆	⋆	⋆	⋆	-	-	-	⋆	-	⋆	⋆	⋆	-	⋆	⋆	2
	Pandora FMS [203]	JS	M	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	-	⋆	⋆	⋆	-	⋆	⋆	2
	Sensu [204]	Go	⋆	⋆	⋆		⋆	⋆	⋆	-	⋆	⋆	-	⋆	⋆	⋆	-	⋆	⋆	6
	Zabbix [205, 206]	C	M	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	2
	Zenoss [207, 208]		M	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	-	⋆	⋆	⋆	⋆	-	⋆	⋆	2
D. Collectors	Collectd [209]	C	M,F	-	⋆	⋆	⋆	-	-	⋆	-	⋆	-	⋆	⋆	/	-	⋆	⋆	6
	Fluentd [210]	R	M	⋆	⋆	⋆	⋆	⋆	⋆	-	-	⋆	-	⋆	⋆	-	-	-	⋆	1
	Flume [211, 212]	J	-	-	⋆	⋆	⋆	-	-	-	-	⋆	-	⋆	-	/	-	-	⋆	1
	Prometheus [213]	Go	F	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	-	⋆	⋆	⋆	-	-	⋆	1
	Scribe [214]	C++	F	-	⋆	⋆	⋆	-	-	-	-	⋆	-	⋆	-	/	-	-	-	1

Legend: ⋆ Available - Not available / Not Applicable

Table 8: *Monitoring and Logging* software classified into *Resource Management* box at the *Platform* layer

JavaScript(JS), Go, or Ruby (R). The next column describes how the data is stored, distinguishing systems that use a single database ($\star$), many databases (M), a file system (F), can optionally use a database (O), or none (-). Also, we indicate whether the system has a web interface or not.

Second, we indicate which elements can be monitored or log-collected by each system; differentiating applications, machines and devices (nodes), networks, clouds, containers, and databases. Also, our taxonomy details the capabilities of each solution. For instance, we differentiate between agent and agentless monitoring. On the one hand, agent monitoring deploys a specifically designed software to gather, analyse, and process data on each host; providing insight information but locking the users to a specific platform (thus, increasing the cost of migration to new platforms without loosing the in-depth data). On the other hand, agentless monitoring relies on hardware and software integrated built-in features to centrally collect, manage, and monitor information without requiring any additional software running on the hosts. Apart from the agent and agentless monitoring, the capabilities also include automatic discovery, and the definition of custom metrics, alerts, and aggregated graphs.

Third, regarding security, we have indicated the capability of the systems to encrypt stored data and authenticate users (e.g., LDAP). Notice that we do not evaluate the encrypted communications since all the alternatives are capable of using SSL.

Finally, we have also included the maintenance and license considering the same features than in the previous cases (see Section 3.1.1 for further details).

4.2.2.2. Analysis.

In general terms, data collectors seem to be more basic tools since they do not implement neither web interfaces, nor support for clouds, containers, and databases, nor data encryption, nor authentication. Data collectors are designed to collect (and analyse) the data from various components of the system at application, device, or network level, and are often used to retrieve the logs from the different parts of the system so that they can be used to check the system's

security, run forensics, or just to monitor their status. Although discontinued in 2014 for better integration with the Hadoop ecosystem, Facebook Scribe is still a reference for aggregating log data streamed in real-time from many servers. since it was designed to be scalable, and network and resource fault-tolerant. Also, Fluentd collects and analyses event and application logs, allowing the users to homogenise them. Similarly, Apache Flume is designed to collect, aggregate, and move large amounts of log data efficiently, and it is based on a simple architecture with many fail-over and recovery mechanisms.

On the other hand, resource monitoring software provides more or less the same features as full-stack software. Resource monitoring software is built to monitor the activity, capacity, and health of any resource within a system, both on-premise and in the cloud. Typically, the implementations are low-level and light-weight, with many optimisations to minimise the data transfers and the monitoring overhead. Zabbix stands up as a reference software; providing high scalability, native agents in many platforms, auto-discovery, and configurable alerts. Also, Ganglia is a monitoring tool designed for high-performance computing systems, clusters and networks.

Full-stack frameworks provide resource monitoring and log collection, aggregation, and analytics as long as intuitive interfaces to visualise the gathered data. Dynatrace, DataDog, and New Relic lead the race [220], while the ELK stack (Elastic [221], Logstash [222, 223], and Kibana [224]) is becoming increasingly popular due to its simplicity yet robust log analysis. However, Nagios, and Splunk are still the state of the art references, providing a feature-rich and robust ecosystem.

More in-depth, all the alternatives provide monitoring/logging capabilities for applications (except Ganglia), nodes, and network. Collectd, Flume, and Scribe are the only ones not supporting clouds, and LogRythm, Icinga 2, Collectd, Flume, and Scribe do not support containers. Also, only half of the surveyed software supports monitoring/logging databases. Regarding the capabilities, software based in agents is definitely the top choice. Only Solarwinds APM Suite and Zenoss work exclusively with agentless monitoring, while Nagios

and Zabbix allow agent and agentless monitoring for different kind of services. Many alternatives also provide automatic discovery, custom metrics, and alerts.

Finally, while all the surveyed alternatives for resource monitoring and data collection are available through different public open licenses, only half of the full-stack frameworks are. For instance, LogRythm, Solarwinds APM Suite, DataDog, Dynatrace, and New Relic use private licenses and require paid plans. However, many of them offer complete guides, tutorials, and live demonstrations that can help the users to match their needs to a suitable option.

5. Communication

In the *Communication* layer, we consider any framework, library, tool, protocol or pattern that eases the communication between distributed processes. Its only purpose is to abstract the users from the infrastructure itself and the network protocols by providing a higher-level API to send and receive messages among computing resources. This layer is built directly on top of the infrastructure and, thus, models are typically low-level.

The main technical challenge of this layer is to maintain efficiency while providing a high enough abstraction so that users do not find themselves fighting against TCP or UDP protocols. Considering that this kind of models aim at advanced users, the proposals must be easy to fine-tune to obtain the expected behaviour without much performance loss.

Figure 4: Communication Layer

As shown in Figure 4, the *Communication* layer contains all the middleware which handles the information exchange in distributed environments. We consider two main paradigms: Remote Invocation and Message-Oriented. In Remote Invocation (RI), models allow the users to make program calls from one node to another via network. We distinguish between Remote Procedure

Call (RPC) and Remote Method Invocation (RMI) when the invocation is a function or an object method, respectively. First implementations date back to early 1980s, but their popularity fell, first, with TCP/IP and, then again, when HTTP became the de-facto standard for communication. The tight coupling, single language requirement, and synchronous nature of RI communications were replaced by inter-operable, loosely coupled asynchronous methods such as REST for HTTP. However, the Java RMI implementation is still around, and a new batch of modern RPC models have appeared. The new RPC models are language-agnostic and high-performant; mainly due to improvements in the serialisation/communication process (such as protocol buffers [225]), and in the synchronisation (thanks to the use of *futures* and *promises*). Google's gRPC is the most well-known framework and it is widely used in micro-services or performance-critical environments like TensorFlow (which uses it for distributed executions).

We can distinguish two paradigms inside the Message-Oriented group: Message Passing Interface (MPI), and Message Queueing (MQ). These paradigms are mostly aligned with HPC and Big Data fields. On the one hand, MPI was born from the need to standardise the proprietary protocols developed for high-speed interconnection networks (used in HPC clusters and supercomputers). MPI attempt to incur in minimal overhead, so their level of abstraction is quite low (barely above sockets). They do not provide any fault tolerance mechanism because failures (such as crashed processes or failed communications) are assumed to be fatal. On the other hand, MQ are based on inserting messages into queues and use some kind of middleware or broker to handle the queues. Also, in contrast to MPI, these queues offer intermediate storage to not require either the sender or receiver to be active during the transmission. They are loosely-coupled in time, persistent, and asynchronous, so they are normally fault-tolerant. The well-known publish-subscribe pattern is a kind of MQ where the messages are grouped by topic. In this model, applications publish messages to a given topic, and all applications subscribed to that topic receive the messages (instead of delivering it to a single receiver like in message queuing).

5.1. Taxonomy

The number of communication solutions is vast, and it is increasing to fit the myriad of different requirements of various environments and use cases. For this survey, we have chosen what we believe are the most representatives frameworks for the Distributed Computing area. Our selection is based on availability (open source), usage (measured by web presence and continuous releases), and relevance (projects using it).

	Software	Features												
		Type of Communication				Fault Tolerance				Security				
		Paradigm	Transient/Persistent	Sync/Async	Group support	Checkpointing	Replication	Resubmission	Failover	Secure Comm.	Data Encryption	API Language	Actively Maintained	License
Communication	ActiveMQ [226, 227]	MQ	P	A	-	-	⋆	⋆	⋆	⋆	⋆	CJP(5)	⋆	1
	AKKA Actors + Streams [228, 229, 230]	MP	P	A,S	⋆	⋆	⋆	⋆	⋆	⋆	⋆	J(1)	⋆	1
	Apache Qpid [231]	MQ	P	A,S	⋆	-	⋆	⋆	⋆	⋆	⋆	CJP(4)	-	1
	Cap'n Proto [232]	RI	T	A,S	-	-	-	-	-	⋆	⋆	CJP(5)	⋆	6
	Flink [233]	MQ	T	A,S	-	⋆	-	-	⋆	⋆	⋆	JP(2)	⋆	1
	gRPC [234]	RI	T	A,S	-	-	-	-	⋆	⋆	⋆	CJP(6)	⋆	1
	Java RMI [235]	RI	T	S	-	-	-	-	-	⋆	⋆	JP(2)		8
	Jgroups [236]	MP	T,P	A,S	⋆	-	⋆	⋆	⋆	⋆	⋆	J(0)	⋆	1
	Kafka [237, 238]	MQ	T	A	-		⋆	⋆	⋆	⋆	⋆	CJP(14)	⋆	1
	OpenMPI [239]	MP	T	A,S	⋆	-	-	-	-	-	-	CP(2)	⋆	5
	RabbitMQ [240]	MQ	P	A	-	-	⋆	⋆	⋆	⋆	⋆	CJP(19)	⋆	1
	Spread Toolkit [241]	MP	P	A,S	⋆	-	⋆	⋆	⋆	-	-	CJP(3)	⋆	8
	Thrift [242]	RI	T	A,S	-	-	-	-	⋆	⋆	⋆	CJP(11)	⋆	1
	ZeroMQ [243]	MQ	T,P	A,S	⋆	-	⋆	⋆	⋆	-	-	CJP(17)	⋆	4

Legend: ⋆ Available - Not available / Not Applicable

Table 9: Software classified into the *Communication* layer

Table 9 presents our communications' taxonomy. The features related to the type of communication are based on the ones listed in [244]. Moreover, to the type of communication features, we also consider fault tolerance, security, and general features such as the API language and licensing.

First, we define the principal type of communication implemented. For the communication paradigm, we differentiate between Remote Invocation (RI in the table), Message Passing (MP), and Message Queuing (MQ). Next, the communication can either be transient (T), where the messages are not stored and thus, the sender and receiver to be active at the transmission time, or persistent

(P), where messages are stored making the communication time-decoupled.

Depending on whether sending a message is blocking or not, the communication can either be asynchronous (A), or synchronous (S). For the sake of clarity, in this classification, we do not differentiate if synchronous communications are when the middleware receives the request, when the message has been delivered, or when the message has been processed. Also, notice that, nowadays, almost all communication solutions offer both synchronous and asynchronous modes.

Furthermore, we indicate whether there is support for group communication or not. We consider group communications to have the following features: messages are sent to group IDs (no need to know actual recipients), messages are delivered to all group members, and the middleware is the responsible for maintaining group memberships.

Depending on the environment, fault tolerance and security might be important concerns. As in previous sections, regarding fault tolerance, we analyse whether the software offers checkpointing, replication, re-submission, and failover (recovering after partial errors). Regarding security, we indicate whether the systems support secure communications and data encryption.

Finally, we also provide some information about the languages supported and the license. Since some solutions offer bindings for a large number of languages, we indicate support for Java (J), Python (P), and C++ (C), and, in the cases where more languages are supported, we indicate how many more are supported with a number in parenthesis. The software maintenance and license is categorised considering the same options than in the previous layers (see Section 3.1.1 for further details).

5.2. Analysis

Generally, RI and MP paradigms are transient, while MQ supports both transient and persistent communications because having a middleware allows to store (or not) the messages upon delivery easily. More in-depth, all the considered RI paradigms are transient, requiring both sender and receiver to be alive at the same time. The MP paradigms are split between MPI transient

communications and the persistent ones which have some kind of middleware (e.g., Jgroups, Spread, and AKKA). Finally, the MQ are inherently persistent as they are saved into a queue. However, systems designed for stream processing (such as Kafka or Flink) only store the data long enough to be processed; which led us to consider them as transient. Nevertheless, users can choose to save the data quite easily with different methods like high retention periods (Kafka) or checkpoints (Flink).

Regarding synchronous communication, RI solutions were synchronous by nature. However, this has changed with modern RI solutions such as gRPC and Thrift. Also, the MPI standard incorporates methods for asynchronous support.

Generally, HPC-oriented communications (like MPI) do not offer neither fault tolerance nor security because they typically run in closed, secure environments. On the other hand, MQ are used in more unreliable environments, and thus, provide different mechanisms to handle failures and security. Most transient-communication solutions do not offer fault tolerance because the communication is expected to fail if the message can not be delivered (as in MPI). On the other hand, most persistent-communications frameworks (like MQ-based models) are robust to failure because they have a middleware that can keep track of failed submissions.

6. Infrastructure

When we talk about infrastructure, we no longer talk about computing nodes themselves but about the infrastructure software that is capable of managing clusters, clouds, virtual machines or containers. As shown in Figure 5, there is a clear division between software developed to administrate and manage batch systems (such as supercomputers), and software developed for interactive systems (such as cloud platforms or container providers). Next subsections provide further information about both approaches by defining, comparing, and categorising the latest software.

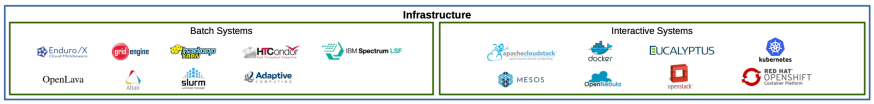

Figure 5: Infrastructure Layer

6.1. Batch systems

Batch systems are in charge of scheduling jobs that can run without user interaction, require a fixed amount of resources, and have a hard timeout among the resources they manage. Since these systems are typically for HPC facilities (i.e., supercomputers), they are designed to administrate a fixed number of computational resources with homogeneous capabilities. To support modern HPC facilities with heterogeneous resources, batch systems can include constraints or queue configurations. Also, in an attempt to make the HPC infrastructure more flexible, some alternatives include job elasticity to vary the amount of resources assigned to a job at execution time.

6.1.1. Taxonomy

Table 10 presents our taxonomy for the batch systems. First, we describe their architecture, detailing whether the implementation language is Java (J in

Software		Features													
		Architecture			Configuration			Limits		F.T.	Security				
		Language	OS	File System	Heter. Resources	Job priority	Group priority	Max. nodes	Max. jobs	Job Checkpointing	Authentication	Encryption	Maintainer	Actively Maintained	License
Batch systems	Enduro/X [245]	C,C++	U	P	⋆	-	-			⋆	OS	M	Maximax Ltd	⋆	2
	GridEngine [246]	C	W,U	A	⋆	⋆	⋆	10k	300k	⋆	M	⋆	Univa	⋆	8
	Hadoop Yarn [247, 248]	J	W,U	H	⋆	⋆	-	10k	500k		M	⋆	Apache SF	⋆	1
	HT Condor [249, 250]	C++	W,U	N	⋆	⋆	⋆	10k	100k	⋆	M	M	UW-Madison	⋆	1
	IBM Spectrum LSF [251]	C,C++	W,U	A	⋆	⋆	⋆	9k	4M	⋆	M	⋆	IBM	⋆	8
	OpenLava [252]	C,C++	L	N	⋆	⋆	⋆			⋆	OS	-	Teraproc	-	2
	PBS Pro [253, 254]	C,P	W,L	P	⋆	⋆	⋆	50k	1M	⋆	OS	-	Altair	⋆	2
	SLURM [255, 256]	C	U	A	⋆	⋆	⋆	120k	100k	⋆	M	-	SchedMD	⋆	2
	Torque [257]	C	U	A	⋆	⋆	⋆			⋆	OS	-	Adaptive Computing	⋆	8

Legend: ⋆ Available - Not available / Not Applicable

Table 10: *Batch systems* classified into the *Infrastructure* layer

the table), Python (P), C++, or C. We also differentiate between the supported underlying operating systems - Windows (W), Linux (L), or Unix-Like (U) - and file systems - NFS (N), Posix (P), HDFS (H), or Any (A) -.

Next, we point out configurable features that system administrators might require; such as support for heterogeneous resources, job priority, and group priority. Also, we provide information about the system limits; stating the maximum number of nodes and jobs that each system has proven to work with.

Regarding fault tolerance, we indicate the support for job checkpointing. Also, regarding security, we categorise the user authentication between operating system (OS) and many (M), and the stored data encryption between many (M), Yes ($\star$) or None (-).

Finally, we have included the maintainer since batch systems are critical for the software stack. Thus, reliability and support from the maintainer might be critical when choosing between the different options. Moreover, we indicate whether the software is under active development and its license considering the same options than in the previous layers (see Section 3.1.1 for further details).

6.1.2. Analysis

Although every system has its own set of user commands, they all provide more or less the same functionalities to the end-user; the only exception being some advanced features such as environment copy, project allocations, or generic resources. Hence, choosing long-term well-supported software is a key points when looking for a batch system. For instance, SLURM is a free and open-source job scheduler used on about the 60% of the TOP500 supercomputers. IBM Spectrum LSF, Torque, and PBS are also widely used options with constant updates and support. Also, HT Condor and Hadoop Yarn are specialised alternatives (e.g., scavenging resources from unused nodes or managing Hadoop clusters) with active user communities.

However, system administrators will find significant differences between systems. Regarding the OS support, all the alternatives are available for Unix-Like systems. Moreover, GirdEngine, Hadoop Yarn, HT Condor, IBM Spectrum

LSF, and PBS Pro also support Windows nodes. As expected, notice that all the alternatives work with NFS or POSIX file systems expect Hadoop Yarn that relies on HDFS. On the other hand, regarding the configuration options and fault tolerance, all the systems support heterogeneous resources, job priority (except Enduro/X), group priority (except Enduro/X and Hadoop Yarn), and job checkpointing (except Hadoop Yarn). Finally, regarding security, all the systems support user authentication through different methods. Although batch systems are usually installed in secure clusters, Enduro/X, GridEngine, Hadoop Yarn, HT Condor, and IBM Spectrum LSF provide data encryption.

Related to the relevance of choosing long-term well-supported software, our taxonomy includes the tested limits. SLURM is the only system that has been proven to work with more than 100k nodes. PBS Pro has been tested with 50k nodes, and GridEngine, Hadoop Yarn, HT Condor and IBM Spectrum LSF are far behind with around 10k nodes. On the other hand, IBM Spectrum LSF and PBS Pro are the only systems that have been proven to handle more than 1 million jobs. Next, Hadoop Yarn has been tested with 500k jobs, Grid Engine with 300k jobs, and HT Condor and SLURM with 100k jobs. Unluckily, we have not been able to find reliable information regarding the limit of nodes nor the limit of jobs for Enduro/X, OpenLava, or Torque.

Finally, all the alternatives except GridEngine and Torque are available through different public open licenses. However, the maintainers offer paid plans for installation and support that are highly recommended for large clusters.

6.2. Interactive systems

In contrast to batch systems, interactive systems are designed for on-demand availability of computing and storage resources; freeing the user from directly managing them. Since these systems are designed for clouds and container platforms, they are required to (1) integrate and free resources from the system, and (2) dynamically adapt the resources assigned to a running job to fulfil its requirements. Typically, these systems handle heterogeneous resources in one or many data centres from one or many organisations.

6.2.1. Taxonomy

Previous work has been published around interactive systems that already analyses many of the features of our taxonomy and discusses some of the alternatives. For instance, [273] provides an in-depth comparison about OpenStack and OpenNebula. Also, the principal investigator and the chief architect discuss the OpenNebula project in [274]. Furthermore, there are online comparisons about Kubernetes and Docker Swarm [275], or Kubernetes and Mesos [276] that were useful when retrieving information for our taxonomy.

Table 11 presents our taxonomy of the surveyed interactive systems. The first two columns classify whether the different technologies work with virtual-machines or containers. Next column defines the supported virtualisation formats; distinguishing between raw (R in the table), compressed (C), Docker (D) or Appc (A). Also, we indicate the number of available hypervisors (e.g., KVM, Xen, Qemu, vSphere, Hyper-V, bare-metal) and the implementation language, differentiating between Java (J), Go (G), Python (P), C (C), and C++ (C++).

Regarding the user interaction, we define the different interfaces for each system: web (W), client (C), REST (R), EC2 (E), and HTTP (H). Moreover, we also include the language of the available libraries distinguishing between Java (J), Python (P), Ruby (R), Go (G), and C++.

	Software	Features																				
		Virtualisation				Code	Interaction		Elasticity			Mgmt.		Application								
		VM	Container	Format	Hypervisor	Language	Interface	Libraries	Automatic Scaling	Bursting	Scalability	Accounting	User quotas	Load Balancing	Object Storage	Live Migration	Rolling Update	Self-healing	Rollbacks	Actively Maintained	License
Interactive sys.	CloudStack [258, 259]	⋆	-	R,C	7	J	W,R,E	-	⋆	-		⋆	⋆	⋆	-	⋆	-	-	-	⋆	1
	Docker Swarm [260, 261]	-	⋆	D	-	G	C,H	G,P	-	-	1kn	-	-	⋆	-	-	⋆	⋆	-	⋆	1
	Eucalyptus [262, 263]	⋆	-	R	1	J,C	W,C,E	-	⋆	-		-	⋆	⋆	⋆	⋆	-	-	-	-	3
	Kubernetes [264, 265]	-	⋆	D	-	G	W,C,R	G,P	⋆	-	5kn	-	⋆	⋆	-	⋆	⋆	⋆	⋆-	⋆	1
	Mesos [266, 267]	-	⋆	D,A	-	C++	W,H	J,C++	⋆	-	50kn	-	⋆	-	-	-	⋆	⋆	-	⋆	1
	OpenNebula [268, 269]	⋆	-	R,C	3	C++	W,C,R	R,J	⋆	⋆		⋆	⋆	-	-	⋆	-	-	-	⋆	1
	OpenStack [270, 271]	⋆	⋆	R,C	6	P	W,C,R	P	⋆	-	120kc	⋆	⋆	⋆	⋆	⋆	-	-	-	⋆	1
	RedHat OpenShift [272]	-	⋆	D	-	G	R,C,W	-	⋆	⋆	1kn	-	⋆	⋆	-	⋆	⋆	⋆	⋆	⋆	1

Legend: ⋆ Available - Not available / Not Applicable

Table 11: *Interactive systems* classified into the *Infrastructure* layer

Regarding the elasticity, we indicate whether the systems support automatic scaling and bursting. Also, we indicate the maximum number of nodes or cores that the system has proven to manage. Furthermore, we indicate whether the systems provide accounting and user quotas since these might be interesting features for system administrators. Similarly, we indicate whether the systems support load balancing, object storage, live migration, rolling updates, self-healing techniques, and rollbacks. Finally, we detail the maintenance and license considering the same options than in the previous layers (see Section 3.1.1 for further details).

6.2.2. Analysis

OpenStack and OpenNebula are the reference software for managing cloud computing infrastructures based on virtual machines. Both solutions are deployed as Infrastructure as a Service (IaaS), supporting multiple, heterogeneous, and distributed data centres and offering private, public, and hybrid clouds. Although both are free and open-source, OpenStack is the only option to handle virtual machines and containers simultaneously. We highlight that all the surveyed alternatives using virtual machines have support for raw and compressed formats through different hypervisors except Eucalyptus.

On the other hand, Docker is the reference container technology offering Platform as a Service (PaaS) products that rely on the OS-level virtualisation to deliver software in packages (also known as containers). While Docker Swarm is the native mode to manage clusters of Docker Engines, Kubernetes is an open-source container-orchestration system to provide automatic deployment and scaling of applications running with containers in a cluster.

More in-depth, regardless of the virtualisation, the systems offer many interfaces; the most common ones being client, REST, or web interfaces. Also, regarding elasticity, all the alternatives except Docker Swarm provide automatic scaling. However, only OpenNebula and RedHat OpenShift provide cloud bursting. Furthermore, regarding user management, all the surveyed systems (except Docker Swarm) provide user quotas, but only Apache CloudStack, OpenNebula,

and OpenStack provide user accounting. Also, regarding application features, most of the systems provide load balancing and live migration. On the other hand, rolling updates, self-healing, and rollbacks are more common in container platforms than virtual machine platforms. Also, Kubernetes and RedHat OpenShift are the richest options; providing all the application management features except Object storage.

Finally, we must highlight that all the alternatives are available through different public open licenses. However, it is recommended to check the paid plans offered by the maintainers for large clusters and continuous support.

7. Conclusion

This paper proposes a layered top-bottom classification of the distributed computing software based on the abstraction level. For each layer, we define the general background, discuss its technical challenges, and build a taxonomy to easily analyse the latest programming languages, programming models, frameworks, libraries, and tools. In total, we review more than 150 different technologies; classifying them considering their core business, although many alternatives can offer functionalities from other layers.

As shown in Figure 6, on the very top of our classification, we define the *Application Development* layer that includes high-level abstraction frameworks that provide all-in-one solutions to develop distributed applications. Often, they rely on huge stacks and are continuously upgraded to support the systems' heterogeneity. Considering their purpose, the different alternatives are classified into Task-based Workflows, Dataflows, and Graph Processing frameworks.

Next, the *Platform* layer includes less general solutions that resolve a single computing challenge: data sharing or resource management. The different alternatives include high-end tools and frameworks that provide a single and homogeneous solution for many underlying infrastructures.

The third layer of our classification is *Communication*. The software within this layer eases the communication between distributed processes (including Re-

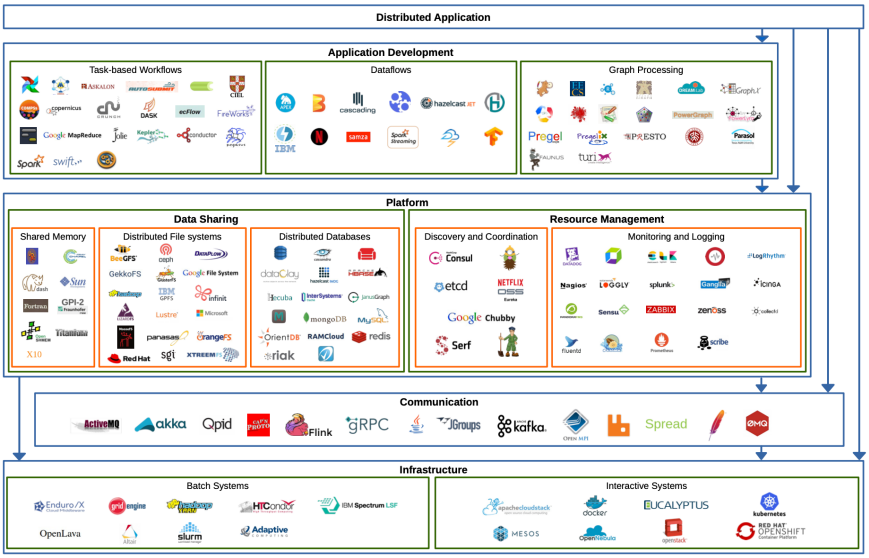

Figure 6: Complete software classification

mote Invocation, Message Passing, and Message Queuing) and its only purpose is to abstract the users from the infrastructure itself and the network protocols by providing a higher-level API to communicate data values and control messages among resources.

Finally, on the bottom of our classification, the *Infrastructure* layer manages clusters, clouds, virtual machines or containers. The software within this layer is classified into batch and interactive systems to differentiate between systems in charge of scheduling jobs with hard timeouts among a fixed set of resources (such as supercomputers), and systems designed for on-demand availability of computing and storage resources (such as cloud platforms or container providers).

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

This work is partly supported by the Spanish Ministry of Science, Innovation, and Universities through the Severo Ochoa Program (SEV-2015-0493) and the TIN2015-65316-P project. It is also supported by the Generalitat de Catalunya under contracts 2014-SGR-1051 and 2014-SGR-1272. Cristian Ramon-Cortes pre-doctoral contract is financed by the Spanish Ministry of Science, Innovation, and Universities under the contract BES-2016-076791.

References

[1] K. Asanovic, et al., The landscape of parallel computing research: A view from berkeley, Technical Report UCB/EECS-2006-183 2.

[2] I. Foster, C. Kesselman, The Grid 2: Blueprint for a new computing infrastructure, Elsevier, 2003.

[3] K. Krauter, R. Buyya, M. Maheswaran, A taxonomy and survey of grid resource management systems for distributed computing, Software: Practice and Experience 32 (2) (2002) 135–164.

[4] V. Kumar, et al., Introduction to parallel computing: design and analysis of algorithms, Vol. 400, Benjamin/Cummings Redwood City, 1994.

[5] K. Asanovic, et al., A view of the Parallel Computing Landscape, Communications of the ACM 52 (10) (2009) 56–67.

[6] S. Kaisler, et al., Big Data: Issues and Challenges Moving Forward, in: 46th Hawaii International Conference on System Sciences, IEEE, 2013, pp. 995–1004.

[7] S. Sagiroglu, D. Sinanc, Big data: A review, in: International Conference on Collaboration Technologies and Systems (CTS), IEEE, 2013, pp. 42–47.

[8] P. Russom, et al., Big data analytics, TDWI best practices report, fourth quarter 19.

[9] J. Dongarra, et al., The international Exascale Software Project roadmap, The International Journal of High Performance Computing Applications 25 (1) (2011) 3–60.

[10] D. A. Reed, J. Dongarra, Exascale Computing and Big Data, Commun. ACM 58 (7) (2015) 5668.

[11] E. Deelman, Big Data Analytics and High Performance Computing Convergence Through Workflows and Virtualization (2016).

[12] S. Cano-Lores, F. Isaila, J. Carretero, Data-Aware Support for Hybrid HPC and Big Data Applications, in: 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2017, pp. 719–722.

[13] C. Hsu, G. Fox, G. Min, S. Sharma, Advances in big data programming, system software and HPC convergence, The Journal of Supercomputing 75 (2019) 489–493. doi:10.1007/s11227-018-2706-x.

[14] G. Fox, et al., Big Data, Simulations and HPC Convergence, in: T. Rabl, et al. (Eds.), Big Data Benchmarking, Springer, Cham, 2016, pp. 3–17.

[15] M. Zaharia, et al., Spark: Cluster computing with working sets., HotCloud 10 (10-10) (2010) 95.

[16] A. Toshniwal, et al., Storm@ twitter, in: Proceedings of the 2014 ACM SIGMOD international conference on Management of data, ACM, 2014, pp. 147–156.

[17] M. Abadi, et al., Tensorflow: Large-scale machine learning on heterogeneous distributed systems, in: arXiv preprint:1603.04467, arXiv, 2016, pp. 1–19.

[18] J. Liu, et al., A Survey of Data-Intensive Scientific Workflow Management, Journal of Grid Computing 13 (4) (2015) 457–493.

[19] B. P. Rimal, E. Choi, I. Lumb, A Taxonomy and Survey of Cloud Computing Systems, in: 2009 Fifth International Joint Conference on INC, IMS and IDC, 2009, pp. 44–51.

[20] C. Kacfah Emani, N. Cullot, C. Nicolle, Understandable Big Data: A survey, Computer Science Review 17 (2015) 70–81.

[21] The Apache Software Foundation, Apache Airflow, `http://airflow.apache.org`, accessed 2 October 2019 (2019).

[22] C. Vecchiola, X. Chu, R. Buyya, Aneka: A software platform for .NET-based cloud computing, High Speed and Large Scale Scientific Computing 18 (2009) 267–295.

[23] T. Fahringer, et al., Askalon: A grid application development and computing environment, in: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing, IEEE, 2005, pp. 122–131.

[24] D. Manubens-Gil, et al., Seamless management of ensemble climate prediction experiments on HPC platforms, in: 2016 International Conference on High Performance Computing Simulation (HPCS), 2016, pp. 895–900.

[25] Ask Solem, Celery, `http://www.celeryproject.org`, accessed 2 October 2019 (2019).

[26] D. G. Murray, et al., CIEL: a universal execution engine for distributed data-flow computing, in: Proceedings of the 8th ACM/USENIX Symposium on Networked Systems Design and Implementation, 2011, pp. 113–126.

[27] Barcelona Supercomputing Center (BSC), COMP Superscalar (COMPSs), `https://compss.bsc.es`, accessed 2 October 2019 (2019).

[28] S. Pronk, et al., Copernicus: A new paradigm for parallel adaptive molecular dynamics, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2011, pp. 60:1–60:10.

[29] The Apache Software Foundation, Apache Crunch, `https://crunch.apache.org`, accessed 2 October 2019 (2013).

[30] NumFOCUS, Dask: Library for dynamic task scheduling, `http://dask.pydata.org`, accessed 2 October 2019 (2019).

[31] ECMWF, EcFlow, `https://confluence.ecmwf.int/display/ECFLOW`, accessed 2 October 2019 (2019).

[32] J. Anubhav, et al., FireWorks: a dynamic workflow system designed for high-throughput applications, Concurrency and computation: practice and experience 27.

[33] E. Afgan, et al., The galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2016 update, in: Nucleic acids research, 2016, p. gkw343.

[34] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, in: Proc. of the 6th Conf. on Symposium on Operating Systems Design and Implementation - Volume 6, OSDI'04, USENIX Association, USA, 2004, p. 10.

[35] F. Montesi, et al., Jolie: a Java orchestration language interpreter engine, Electronic Notes in Theoretical Computer Science 181 (2007) 19–33.

[36] I. Altintas, et al., Kepler: an extensible system for design and execution of scientific workflows, in: Proceedings. 16th International Conference on Scientific and Statistical Database Management, IEEE, 2004, pp. 423–424.

[37] Netflix, Netflix Conductor, `https://netflix.github.io/conductor`, accessed 2 October 2019 (2019).

[38] E. Deelman, et al., Pegasus, a workflow management system for science automation, Future Generation Computer Systems 46 (2015) 17–35.

[39] M. Wilde, et al., Swift: A language for distributed parallel scripting, Parallel Computing 37(9) (2011) 633–652.

[40] D. Hull, et al., Taverna: a tool for building and running workflows of services, Nucleic Acids Research 34 (Web Server issue) (2006) W729–W732.

[41] The Apache Software Foundation, Apache License, version 2.0, `https://www.apache.org/licenses/LICENSE-2.0`, accessed 2 October 2019 (2019).

[42] Free Software Foundation (FSF), GNU General Public License, version 2, `https://www.gnu.org/licenses/old-licenses/gpl-2.0.html`, accessed 2 October 2019 (2017).

[43] Free Software Foundation (FSF), GNU General Public License, version 3, `https://www.gnu.org/licenses/gpl-3.0.en.html`, accessed 2 October 2019 (2016).

[44] Free Software Foundation (FSF), GNU Lesser General Public License, version 2.1, `https://www.gnu.org/licenses/old-licenses/lgpl-2.1.html`, accessed 2 October 2019 (2018).

[45] The Linux Information Project, BSD License, `http://www.linfo.org/bsdlicense`, accessed 2 October 2019 (2005).

[46] Opensource.org, MIT License, `https://opensource.org/licenses/MIT`, accessed 2 October 2019 (2019).

[47] Opensource.org, Academic Free License version 3.0, `https://opensource.org/licenses/AFL-3.0`, accessed 2 October 2019 (2005).

[48] Mozilla Foundation, Mozilla Public License Version 2.0, https://www.mozilla.org/en-US/MPL/2.0, accessed 2 October 2019 (2019).

[49] Eclipse Foundation Inc., Eclipse Public License v1.0, https://www.eclipse.org/legal/epl-v10.html, accessed 2 October 2019 (2019).

[50] The Apache Software Foundation, Apache Apex, https://apex.apache.org, accessed 2 October 2019 (2019).

[51] The Apache Software Foundation, Apache Beam, https://beam.apache.org, accessed 2 October 2019 (2019).

[52] Cascading Maintainers, Cascading, http://www.cascading.org, accessed 2 October 2019 (2018).

[53] The Apache Software Foundation, Apache Gearpump, https://gearpump.apache.org, accessed 2 October 2019 (2019).

[54] Hazelcast Inc., Hazelcast Jet, https://jet.hazelcast.org, accessed 2 October 2019 (2019).

[55] S. Kulkarni, et al., Twitter heron: Stream processing at scale, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, ACM, 2015, pp. 239–250.

[56] M. Hirzel, et al., IBM streams processing language: Analyzing big data in motion, IBM Journal of Research and Development 57 (3/4) (2013) 7–11.

[57] B. Schmaus, et al., Netflix Blog: Stream processing with Mantis, https://medium.com/netflix-techblog/stream-processing-with-mantis-78af913f51a6, accessed 2 October 2019 (2016).

[58] The Apache Software Foundation, Apache Samza, http://samza.apache.org, accessed 2 October 2019 (2019).

[59] M. Zaharia, et al., Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters, HotCloud 12 (2012) 10–16.

[60] The Apache Software Foundation, Apache Hama, https://hama.apache.org, accessed 2 October 2019 (2016).

[61] A. Buluç, J. R. Gilbert, The Combinatorial BLAS: Design, implementation, and applications, The International Journal of High Performance Computing Applications 25 (4) (2011) 496–509.

[62] A. Azad, A. Buluç, J. R. Gilbert, Combinatorial BLAS (CombBLAS), https://people.eecs.berkeley.edu/~aydin/CombBLAS/html/index.html, accessed 2 October 2019 (2018).

[63] V. Amelkin, A. Buluç, J. R. Gilbert, Knowledge Discovery Toolbox (KDT), http://kdt.sourceforge.net, accessed 2 October 2019 (2013).

[64] Micro Focus, Distributed R, https://marketplace.microfocus.com/vertica/content/distributed-r, accessed 2 October 2019 (2019).

[65] The Apache Software Foundation, Giraph, http://giraph.apache.org, accessed 2 October 2019 (2019).

[66] Y. Simmhan, et al., GoFFish: A Sub-graph Centric Framework for Large-Scale Graph Analytics, in: Euro-Par 2014 Parallel Processing, Springer, 2014, pp. 451–462.

[67] DREAM:Lab, GoFFish, http://dream-lab.cds.iisc.ac.in/projects/goffish, accessed 2 October 2019 (2017).

[68] R. S. Xin, et al., Graphx: A resilient distributed graph system on spark, in: First International Workshop on Graph Data Management Experiences and Systems, ACM, 2013, pp. 1–6.

[69] The Apache Software Foundation, Apache Spark - GraphX, https://spark.apache.org/graphx, accessed 2 October 2019 (2018).

[70] B. Shao, H. Wang, Y. Li, Trinity: A distributed graph engine on a memory cloud, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, ACM, 2013, pp. 505–516.

[71] Microsoft, Microsoft Trinity Project, https://www.microsoft.com/en-us/research/project/trinity, accessed 2 October 2019 (2019).

[72] Microsoft, Graph Engine, https://www.graphengine.io, accessed 2 October 2019 (2017).

[73] S. Salihoglu, J. Widom, GPS: a graph processing system, in: Proceedings of the 25th International Conference on Scientific and Statistical Database Management, ACM, 2013, pp. 1–12.

[74] J. Widom, GPS: Graph Processing System, http://infolab.stanford.edu/gps, accessed 2 October 2019 (2014).

[75] P. Wang, et al., Replication-based fault-tolerance for large-scale graph processing, in: 2014 44th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN), IEEE, 2014, pp. 562–573.

[76] Shanghai Jiao Tong University, Imitator, http://ipads.se.sjtu.edu.cn/projects/imitator.html, accessed 2 October 2019 (2014).

[77] D. Gregor, A. Lumsdaine, The parallel BGL: A generic library for distributed graph computations, Parallel Object-Oriented Scientific Computing (POOSC) 2 (2005) 1–18.

[78] N. Edmonds, D. Gregor, A. Lumsdaine, Parallel Boost Graph Library, http://www.boost.org/doc/libs/1_53_0/libs/graph_parallel/doc/html/index.html, accessed 2 October 2019 (2009).

[79] J. E. Gonzalez, et al., PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs, in: Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), USENIX, 2012, pp. 17–30.

[80] R. Chen, et al., PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs, in: Proceedings of the Tenth European Conference on Computer Systems, ACM, 2015, pp. 1:1–1:15.

[81] R. Chen, PowerLyra, http://ipads.se.sjtu.edu.cn/projects/powerlyra.html, accessed 2 October 2019 (2013).

[82] G. Malewicz, et al., Pregel: a system for large-scale graph processing, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, ACM, 2010, pp. 135–146.

[83] Y. Bu, et al., Pregelix: Big(Ger) Graph Analytics on a Dataflow Engine, Proc. VLDB Endow. 8 (2) (2014) 161–172.

[84] Pregelix Team, Pregelix, http://pregelix.ics.uci.edu, accessed 2 October 2019 (2014).

[85] S. Venkataraman, et al., Presto: Distributed Machine Learning and Graph Processing with Sparse Matrices, in: Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys 2013), ACM, 2013, pp. 197–210.

[86] J. Xue, et al., Processing concurrent graph analytics with decoupled computation model, IEEE Transactions on Computers 66 (5) (2017) 876–890.

[87] M. Zandifar, et al., The STAPL skeleton framework, in: International Workshop on Languages and Compilers for Parallel Computing, Springer, 2014, pp. 176–190.

[88] Parasol Laboratory, STAPL: Standard Template Adaptive Parallel Library (Parasol), https://parasol.tamu.edu/groups/rwergergroup/research/stapl, accessed 2 October 2019 (2017).

[89] DataStax Inc., Titan Hadoop (Faunus), https://github.com/thinkaurelius/faunus, accessed 2 October 2019 (2015).

[90] Y. Low, et al., Distributed GraphLab: a framework for machine learning and data mining in the cloud, Proceedings of the VLDB Endowment 5 (8) (2012) 716–727.

[91] Turi, Turi Create, `https://turi.com`, accessed 2 October 2019 (2018).

[92] N. Doekemeijer, A. L. Varbanescu, A survey of parallel graph processing frameworks, Tech. rep., Technical Report PDS-2014-003. Delft University of Technology (2014).

[93] L. G. Valiant, A Bridging Model for Parallel Computation, Commun. ACM 33 (8) (1990) 103–111.

[94] T. El-Ghazawi, et al., Unified Parallel C, `http://upc.gwu.edu`, accessed 2 October 2019 (2005).

[95] U. Consortium, UPC Language Specifications V1.2, Tech. rep., UPC Consortium (5 2005). `doi:10.2172/862127`.

[96] C. Coarfa, et al., An evaluation of global address space languages: co-array fortran and unified parallel C, in: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, ACM, 2005, pp. 36–47.

[97] B. L. Chamberlain, D. Callahan, H. P. Zima, Parallel Programmability and the Chapel Language, The International Journal of High Performance Computing Applications 21 (3) (2007) 291–312.

[98] CRAY, The Chapel Parallel Programming Language, `https://chapel-lang.org`, accessed 2 October 2019 (2019).

[99] K. Fürlinger, T. Fuchs, R. Kowalewski, DASH: a C++ PGAS library for distributed data structures and parallel algorithms, in: 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart

City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), Ieee, 2016, pp. 983–990.

[100] Dash Team, DASH, http://www.dash-project.org, accessed 2 October 2019 (2018).

[101] E. Allen, et al., The Fortress language specification, Sun Microsystems 139 (140) (2005) 116.

[102] R. W. Numrich, J. Reid, Co-array Fortran for Parallel Programming, SIGPLAN Fortran Forum 17 (2) (1998) 1–31.

[103] GPI-2, GPI-2: Programming Next Generation Supercomputers, http://www.gpi-site.com, accessed 2 October 2019 (2019).

[104] B. Chapman, et al., Introducing OpenSHMEM: SHMEM for the PGAS Community, in: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model, Association for Computing Machinery, 2010, pp. 1–3.

[105] Silicon Graphics International Corp., OpenSHMEM, http://www.openshmem.org, accessed 12 February 2020 (2019).

[106] K. Yelick, et al., Titanium: a high-performance Java dialect, Concurrency and Computation: Practice and Experience 10 (11-13) (1998) 825–836.

[107] P. N. Hilfinger, et al., Titanium language reference manual (2006).

[108] Computer Science Division, University of California at Berkeley, Titanium, http://titanium.cs.berkeley.edu, accessed 2 October 2019 (2014).

[109] P. Charles, et al., X10: An Object-oriented Approach to Non-uniform Cluster Computing, SIGPLAN Not. 40 (10) (2005) 519–538.

[110] V. Saraswat, et al., X10 Language Specification - Version 2.6.2 (2019).

[111] IBM, The X10 Parallel Programming Language, http://x10-lang.org, accessed 2 October 2019 (2018).

[112] PGAS org, PGAS: Partitioned Global Address Space, http://www.pgas.org, accessed 2 October 2019 (2016).

[113] O. Tardieu, The apgas library: Resilient parallel and distributed programming in java 8, in: Proceedings of the ACM SIGPLAN Workshop on X10, X10 2015, ACM, 2015, pp. 25–26.

[114] J. Breitbart, M. Schmidtobreick, V. Heuveline, Evaluation of the Global Address Space Programming Interface (GASPI), in: 2014 IEEE International Parallel Distributed Processing Symposium Workshops, 2014, pp. 717–726.

[115] GASPI-Forum, GASPI: Global Address Space Programming Interface, http://www.gaspi.de, accessed 2 October 2019 (2019).

[116] T. Alrutz, et al., GASPI – A Partitioned Global Address Space Programming Interface, in: Facing the Multicore-Challenge III: Aspects of New Paradigms and Technologies in Parallel Computing, Springer, 2013, pp. 135–136.

[117] Mellanox Technologies, What is RDMA?, https://community.mellanox.com/s/article/what-is-rdma-x, accessed 12 February 2020 (2019).

[118] D. Bonachea, P. H. Hargrove, GASNet-EX: A High-Performance, Portable Communication Library for Exascale, in: International Workshop on Languages and Compilers for Parallel Computing, Springer, 2018, pp. 138–158.

[119] Berkeley Lab., GASNet, https://gasnet.lbl.gov, accessed 12 February 2020 (2020).

[120] J. Heichler, An introduction to BeeGFS, Tech. rep., BeeGFS (2014).
URL https://www.beegfs.io/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf

[121] ThinkParQ and Fraunhofer, BeeGFS, https://www.beegfs.io, accessed 2 October 2019 (2019).

[122] S. A. Weil, et al., Ceph: A Scalable, High-performance Distributed File System, in: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI'06, USENIX Association, USA, 2006, p. 307320.

[123] DataPlow Inc., DataPlow Nasan File System, http://www.dataplow.com/Products.htm#Nasan, accessed 2 October 2019 (2019).

[124] M.-A. Vef, et al., GekkoFS - A temporary distributed file system for HPC applications, in: 2018 IEEE International Conference on Cluster Computing (CLUSTER), IEEE, 2018, pp. 319–324.

[125] NGIO project. BSC in collaboration with JGU, GekkoFS, https://github.com/NGIOproject/GekkoFS, accessed 22 May 2020 (2020).

[126] E. B. Boyer, M. C. Broomfield, T. A. Perrotti, Glusterfs one storage server to rule them all, Tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States) (2012).
URL https://www.osti.gov/biblio/1048672

[127] A. Davies, A. Orsaria, Scale out with GlusterFS, Linux J. 2013 (235).

[128] S. Ghemawat, H. Gobioff, S. Leung, The Google File System, SIGOPS Oper. Syst. Rev. 37 (5) (2003) 29–43.

[129] K. Shvachko, et al., The hadoop distributed file system, in: 2010 IEEE 26th symposium on Mass storage systems and technologies (MSST), IEEE, 2010, pp. 1–10.

[130] F. Schmuck, R. Haskin, GPFS: A Shared-Disk File System for Large Computing Clusters, in: Proceedings of the 1st USENIX Conference on File and Storage Technologies, FAST'02, USENIX Association, USA, 2002, p. 16.

[131] Infinit International Inc., Infinit Storage Platform, `https://infinit.sh/reference`, accessed 2 October 2019 (2015).

[132] LizardFS Inc., LizardFS, `https://lizardfs.com`, accessed 2 October 2019 (2019).

[133] S. Faibish, et al., Lustre File System, uS Patent 9,779,108 (3 2017).

[134] A. D'amato, et al., Cluster shared volumes, uS Patent 7,840,730 (11 2010).

[135] Core Technology Sp. z o.o., MooseFS, `https://moosefs.com`, accessed 2 October 2019 (2019).

[136] D. Nagle, D. Serenyi, A. Matthews, The Panasas ActiveScale Storage Cluster: Delivering Scalable High Bandwidth Storage, in: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, IEEE, 2004, pp. 53–.

[137] P. Inc, Panasas ActiveStor Architecture Overview, Tech. rep., White paper (2017).
URL `http://performance.panasas.com/wp-architecture-hp-thanks.html`

[138] P. H. Carns, et al., PVFS: A parallel file system for Linux clusters, in: Proceedings of the 4th Annual Linux Showcase and Conference – Volume 4, ALS'00, USENIX Association, 2000, pp. 28–29.

[139] OrangeFS.org, The OrangeFS Project, `http://www.orangefs.org`, accessed 2 October 2019 (2018).

[140] S. Whitehouse, The GFS2 filesystem, in: Proceedings of the Linux Symposium, Citeseer, 2007, pp. 253–259.

[141] R. Inc., Red Hat Global File System, Tech. rep., RedHat (2004).
URL https://listman.redhat.com/whitepapers/rha/gfs/GFS_INS0032US.pdf

[142] L. Shepard, E. Eppe, SGI® InfiniteStorage Shared Filesystem CXFS: A High-Performance, Multi-OS Filesystem from SGI, Tech. rep., White paper 2691 (2003).
URL https://jarrang.com/client/sgi/storage/StorageCD/Collateral/DataSheets/GeneralAndWhitepapers/SGIInfiniteStorageSharedFilesystemCXFSwhitepaper.pdf

[143] J. Stender, M. Berlin, A. Reinefeld, XtreemFS: A file system for the cloud, in: Data intensive storage services for cloud environments, IGI Global, 2013, pp. 267–285.

[144] Amazon Web Services Inc., DynamoDB, https://aws.amazon.com/es/dynamodb, accessed 4 December 2019 (2019).

[145] A. Lakshman, P. Malik, Cassandra: a decentralized structured storage system, ACM SIGOPS Operating Systems Review 44 (2) (2010) 35–40.

[146] J. C. Anderson, J. Lehnardt, N. Slater, CouchDB: the definitive guide, O'Reilly Media Inc., 2010.

[147] The Apache Software Foundation, Apache CouchDB, https://couchdb.apache.org, accessed 4 December 2019 (2019).

[148] J. Martí, et al., Dataclay: A distributed data store for effective inter-player data sharing, Journal of Systems and Software 131 (2017) 129–145.

[149] Barcelona Supercomputing Center (BSC), dataClay, https://www.bsc.es/research-and-development/software-and-apps/software-list/dataclay, accessed 4 December 2019 (2019).

[150] Hazelcast Inc., Hazelcast IMDG, https://hazelcast.com/products/imdg, accessed 4 December 2019 (2019).

[151] M. N. Vora, Hadoop-HBase for large-scale data, in: Proceedings of 2011 International Conference on Computer Science and Network Technology, Vol. 1, IEEE, 2011, pp. 601–605.

[152] The Apache Software Foundation, Apache HBase, https://hbase.apache.org, accessed 4 December 2019 (2019).

[153] G. Alomar, Y. Becerra, J. Torres, Hecuba: Nosql made easy, in: BSC Doctoral Symposium (2nd: 2015: Barcelona), Barcelona Supercomputing Center, 2015, pp. 136–137.

[154] E. Tejedor, et al., PyCOMPSs: Parallel computational workflows in Python, The International Journal of High Performance Computing Applications (IJHPCA) 31 (1) (2017) 66–82.

[155] Barcelona Supercomputing Center (BSC), Hecuba, https://github.com/bsc-dd/hecuba, accessed 4 December 2019 (2019).

[156] InterSystems Corporation, Intersystems Cache, https://www.intersystems.com/products/cache, accessed 4 December 2019 (2019).

[157] JanusGraph Authors, JanusGraph, https://janusgraph.org, accessed 4 December 2019 (2019).

[158] Thinkaurelius, Titan: Distributed Graph Database, http://titan.thinkaurelius.com, accessed 11 May 2020 (2015).

[159] Dormando, Memcached, https://memcached.org, accessed 4 December 2019 (2018).

[160] K. Banker, MongoDB in action, Manning Publications Co., 2011.

[161] MongoDB Inc., MongoDB: The most popular database for modern apps, https://www.mongodb.com, accessed 4 December 2019 (2019).

[162] S. Suehring, MySQL bible, John Wiley & Sons Inc., 2002.

[163] Oracle Corporation, MySQL, https://www.mysql.com, accessed 4 December 2019 (2019).

[164] C. Tesoriero, Getting Started with OrientDB, Packt Publishing Ltd, 2013.

[165] Callidus Software Inc., OrientDB: The database designed for the modern world, https://orientdb.com, accessed 4 December 2019 (2019).

[166] J. Ousterhout, et al., The case for RAMClouds: scalable high-performance storage entirely in DRAM, ACM SIGOPS Operating Systems Review 43 (4) (2010) 92–105.

[167] J. Ousterhout, RAMCloud Project, https://ramcloud.atlassian.net/wiki/spaces/RAM/overview, accessed 22 May 2020 (2019).

[168] T. Macedo, F. Oliveira, Redis Cookbook: Practical Techniques for Fast Data Manipulation, O'Reilly Media Inc., 2011.

[169] RedisLabs, Redis, https://redis.io, accessed 4 December 2019 (2019).

[170] Basho Technologies, Riak, https://riak.com/riak, accessed 4 December 2019 (2019).

[171] OpenLink Software, Virtuoso: Data-driven agility without compromise, https://virtuoso.openlinksw.com, accessed 4 December 2019 (2019).

[172] HashiCorp, Consul, https://www.consul.io, accessed 4 December 2019 (2019).

[173] K. Rarick, Introducing Doozerd, https://xph.us/2011/04/13/introducing-doozer.html, accessed 4 December 2019 (2011).

[174] The etcd authors, etcd: A distributed, reliable key-value store for the most critical data of a distributed system, https://etcd.io, accessed 4 December 2019 (2019).

[175] Netflix, Netflix Eureka - GitHub repository, https://github.com/Netflix/eureka, accessed 4 December 2019 (2019).

[176] M. Burrows, The chubby lock service for loosely-coupled distributed systems, in: Proceedings of the 7th symposium on Operating systems design and implementation, USENIX Association, 2006, pp. 335–350.

[177] Ameya, Chubby: A lock service for distributed coordination, In https://medium.com/coinmonks/chubby-a-centralized-lock-service-for-distributed-applications-390571273052, accessed 4 December 2019 (2018).

[178] HashiCorp, Serf, https://www.serf.io, accessed 4 December 2019 (2019).

[179] P. Hunt, et al., ZooKeeper: Wait-free Coordination for Internet-scale Systems, in: USENIX annual technical conference, Vol. 8, Boston, MA, USA, 2010, pp. 1–14.

[180] The Apache Software Foundation, Apache ZooKeeper, https://zookeeper.apache.org, accessed 4 December 2019 (2019).

[181] HashiCorp, Serf vs. ZooKeeper, doozerd, etcd, https://www.serf.io/intro/vs-zookeeper.html, accessed 4 December 2019 (2019).

[182] I. Glushkov, Comparing ZooKeeper and Consul, https://es.slideshare.net/IvanGlushkov/zookeeper-vs-consul-41882991, accessed 4 December 2019 (2014).

[183] Farcic, V., Service Discovery: Zookeeper vs etcd vs Consul, https://technologyconversations.com/2015/09/08/service-discovery-zookeeper-vs-etcd-vs-consul, accessed 4 December 2019 (2015).

[184] L. Lamport, et al., Paxos made simple, ACM Sigact News 32 (4) (2001) 18–25.

[185] D. Ongaro, J. Ousterhout, In search of an understandable consensus algorithm, in: 2014 USENIX Annual Technical Conference (USENIX ATC 14), 2014, pp. 305–319.

[186] K. Birman, The promise, and limitations, of gossip protocols, ACM SIGOPS Operating Systems Review 41 (5) (2007) 8–13.

[187] Datadog, Datadog: Cloud Monitoring as a Service, https://www.datadoghq.com, accessed 4 December 2019 (2019).

[188] F. Willnecker, A. Brunnert, W. Gottesheim, H. Krcmar, Using Dynatrace Monitoring Data for Generating Performance Models of Java EE Applications, in: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, Association for Computing Machinery, 2015, p. 103104. doi:10.1145/2668930.2688061.
URL https://doi.org/10.1145/2668930.2688061

[189] Dynatrace, Dynatrace: The Leader in Cloud Monitoring, https://www.dynatrace.com, accessed 21 May 2021 (2021).

[190] Elasticsearch B.V., ELK Stack: Elasticsearch, Logstash, Kibana, https://www.elastic.co/what-is/elk-stack, accessed 4 December 2019 (2019).

[191] Graylog, Graylog, https://www.graylog.org, accessed 4 December 2019 (2019).

[192] P. Villella, C. Petersen, Log collection, structuring and processing, uS Patent 7,653,633 (jan 2010).

[193] LogRhythm Inc., LogRhythm: The Security Intelligence Company, https://logrhythm.com, accessed 4 December 2019 (2019).

[194] W. Barth, Nagios: System and network monitoring, No Starch Press, 2008.

[195] Nagios Enterprises, LLC, Nagios - The Industry Standard In IT Infrastructure Monitoring, https://www.nagios.org, accessed 4 December 2019 (2019).

[196] New Relic, New Relic, https://newrelic.com, accessed 21 May 2021 (2021).

[197] Solarwinds, Log Management by Loggly, https://www.loggly.com, accessed 4 December 2019 (2019).

[198] D. Carasso, Exploring splunk, CITO Research New York, USA, 2012.

[199] Splunk Inc., Splunk: SIEM, AIOps, Application Management, Log Management, Machine Learning, and Compliance, https://www.splunk.com, accessed 4 December 2019 (2019).

[200] M. L. Massie, B. N. Chun, D. E. Culler, The ganglia distributed monitoring system: design, implementation, and experience, Parallel Computing 30 (7) (2004) 817–840.

[201] Ganglia Project, Ganglia Monitoring System, http://ganglia.info, accessed 2 October 2019 (2018).

[202] Icinga GmbH, Icinga, https://icinga.com, accessed 4 December 2019 (2019).

[203] Pandora FMS, Pandora FMS: The flexible monitoring software for large business, https://pandorafms.com, accessed 4 December 2019 (2019).

[204] Sensu Inc., Sensu, https://sensu.io, accessed 4 December 2019 (2019).

[205] R. Olups, Zabbix Network Monitoring, Packt Publishing Ltd, 2016.

[206] Zabbix LLC., Zabbix: The Enterprise-Class Open Source Network Monitoring Solution, https://www.zabbix.com, accessed 4 December 2019 (2019).

[207] M. Badger, Zenoss core network and system monitoring, Packt Publishing Ltd, 2008.

[208] Zenoss Inc., Zenoss: Intelligent Application and Service Monitoring + AIOps, https://www.zenoss.com, accessed 4 December 2019 (2019).

[209] F. e. a. Forster, Collectd, https://collectd.org, accessed 4 December 2019 (2019).

[210] Fluentd Project, Fluentd: Open Source Data Collector and Unified Logging Layer, https://www.fluentd.org, accessed 4 December 2019 (2019).

[211] S. Hoffman, Apache Flume: Distributed Log Collection for Hadoop, Packt Publishing Ltd, 2013.

[212] The Apache Software Foundation, Apache Flume, https://flume.apache.org, accessed 4 December 2019 (2019).

[213] Prometheus Authors, Prometheus: From metrics to insight, https://prometheus.io, accessed 4 December 2019 (2019).

[214] Facebook, Scribe: Transporting petabytes per hour via a distributed, buffered queueing system, https://engineering.fb.com/data-infrastructure/scribe, accessed 4 December 2019 (2019).

[215] Opentica, Open Source Monitoring Tools, http://opentica.com/en/2016/02/02/open-source-monitoring-tools, accessed 4 December 2019 (2016).

[216] L. Kufel, Tools for distributed systems monitoring, Foundations of Computing and Decision Sciences 41 (4) (2016) 237–260.

[217] R. Bhargava, Best of 2018: Log Monitoring and Analysis: Comparing ELK, Splunk and Graylog, https://devops.com/log-monitoring-and-analysis-comparing-elk-splunk-and-graylog, accessed 4 December 2019 (2018).

[218] T. Keary, Nagios vs Zabbix Compared Which Is Better for Network Monitoring?, https://www.comparitech.com/net-admin/nagios-vs-zabbix, accessed 4 December 2019 (2018).

[219] N. Peri, Fluentd vs. Logstash: A Comparison of Log Collectors, https://logz.io/blog/fluentd-logstash, accessed 4 December 2019 (2015).

[220] Charley Rich, Federico De Silva, Gartner Magic Quadrant for Application Performance Monitoring, https://www.gartner.com/en/documents/3983892/magic-quadrant-for-application-performance-monitoring, accessed 21 May 2021 (2020).

[221] Elasticsearch B.V., Elasticsearch: The Official Distributed Search and Analytics, https://www.elastic.co/products/elasticsearch, accessed 4 December 2019 (2019).

[222] J. Turnbull, The Logstash Book, James Turnbull, 2013.

[223] Elasticsearch B.V., Logstash: Collect, Parse, Transform Logs, https://www.elastic.co/products/logstash, accessed 4 December 2019 (2019).

[224] Elasticsearch B.V., Kibana: Explore, Visualize, Discover Data, https://www.elastic.co/products/kibana, accessed 4 December 2019 (2019).

[225] Google, Protocol Buffers, https://developers.google.com/protocol-buffers, accessed 3 December 2019 (2019).

[226] B. Snyder, D. Bosnanac, R. Davies, ActiveMQ in action, Vol. 47, Manning Greenwich Conn., 2011.

[227] The Apache Software Foundation, Apache ActiveMQ, https://activemq.apache.org, accessed 3 December 2019 (2019).

[228] M. Gupta, Akka essentials, Packt Publishing Ltd, 2012.

[229] Lightbend Inc., AKKA Documentation - Classic Actors, https://doc.akka.io/docs/akka/current/actors.html, accessed 3 December 2019 (2019).

[230] Lightbend Inc., AKKA Documentation - Streams, https://doc.akka.io/docs/akka/current/stream/index.html, accessed 3 December 2019 (2019).

[231] The Apache Software Foundation, Apache Qpid, https://qpid.apache.org, accessed 3 December 2019 (2015).

[232] Sandstorm, Cap'n proto: Introduction, https://capnproto.org, accessed 3 December 2019 (2013).

[233] P. Carbone, et al., Apache flink: Stream and batch processing in a single engine, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 36 (4).

[234] gRPC, gRPC Motivation and Design Principles, https://grpc.io/blog/principles, accessed 3 December 2019 (2015).

[235] Oracle, Java Remote Method Invocation Specification, https://docs.oracle.com/javase/9/docs/specs/rmi, accessed 3 December 2019 (2017).

[236] B. Ban, et al., JGroups, a toolkit for reliable multicast communication (2002).

[237] J. Kreps, N. Narkhede, J. Rao, et al., Kafka: A distributed messaging system for log processing, in: Proceedings of the NetDB, 2011, pp. 1–7.

[238] The Apache Software Foundation, Apache Kafka, https://kafka.apache.org, accessed 3 December 2019 (2017).

[239] E. Gabriel, et al., Open MPI: Goals, concept, and design of a next generation MPI implementation, in: European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, Springer, 2004, pp. 97–104.

[240] A. Videla, J. J. Williams, RabbitMQ in action: distributed messaging for everyone, Manning, 2012.

[241] L. Spread Concepts, The spread toolkit (2006).

[242] A. Prunicki, Apache Thrift, Tech. rep., Object Computing, Inc. (2009).

[243] P. Hintjens, ZeroMQ: messaging for many applications, O'Reilly Media Inc., 2013.

[244] A. S. Tanenbaum, M. Van Steen, Distributed systems: principles and paradigms, Prentice-Hall, 2007.

[245] Mavimax, Enduro/X Middleware Platform for Distributed Transaction Processing, `https://www.endurox.org`, accessed 3 December 2019 (2015).

[246] W. Gentzsch, Sun grid engine: Towards creating a compute power grid, in: Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on, IEEE, 2001, pp. 35–36.

[247] V. K. Vavilapalli, et al., Apache hadoop yarn: Yet another resource negotiator, in: Proceedings of the 4th annual Symposium on Cloud Computing, ACM, 2013, p. 5.

[248] The Apache Software Foundation, Apache Hadoop YARN, `https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`, accessed 3 December 2019 (2019).

[249] D. Thain, T. Tannenbaum, M. Livny, Distributed computing in practice: the Condor experience, Concurrency and computation: practice and experience 17 (2-4) (2005) 323–356.

[250] University of Wisconsin-Madison - Computer Sciences Department, HTCondor - High Troughput Computing, `https://research.cs.wisc.edu/htcondor`, accessed 3 December 2019 (2019).

[251] IBM, IBM LSF, `https://www.ibm.com/support/knowledgecenter/en/SSETD4/product_welcome_platform_lsf.html`, accessed 3 December 2019 (2016).

[252] P. Joshi, M. R. Babu, Openlava: An open source scheduler for high performance computing, in: 2016 International Conference on Research Advances in Integrated Navigation Systems (RAINS), 2016, pp. 1–3.

[253] R. L. Henderson, Job scheduling under the portable batch system, in: Workshop on Job Scheduling Strategies for Parallel Processing, Springer, 1995, pp. 279–294.

[254] Altair Engineering Inc., PBS Professional - Open Source Project, `https://www.pbspro.org`, accessed 3 December 2019 (2019).

[255] A. B. Yoo, M. A. Jette, M. Grondona, Slurm: Simple linux utility for resource management, in: Workshop on Job Scheduling Strategies for Parallel Processing, Springer, 2003, pp. 44–60.

[256] Slurm Team, Slurm Workload Manager, `https://slurm.schedmd.com`, accessed 3 December 2019 (2019).

[257] Adaptive Computing Inc., TORQUE Resource Manager, `https://www.adaptivecomputing.com/products/torque`, accessed 3 December 2019 (2019).

[258] R. Kumar, et al., Apache cloudstack: Open source infrastructure as a service cloud computing platform, Proceedings of the International Journal of advancement in Engineering technology, Management and Applied Science 111 (2014) 116.

[259] The Apache Software Foundation, Apache CloudStack - Open Source Cloud Computing, `https://cloudstack.apache.org`, accessed 3 December 2019 (2017).

[260] N. Naik, Building a virtual system of systems using docker swarm in multiple clouds, in: 2016 IEEE International Symposium on Systems Engineering (ISSE), 2016, pp. 1–3.

[261] Docker Inc., Swarm Mode Overview, https://docs.docker.com/engine/swarm, accessed 3 December 2019 (2019).

[262] D. Nurmi, et al., The eucalyptus open-source cloud-computing system, in: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, IEEE Computer Society, 2009, pp. 124–131.

[263] Appscale Systems, Eucalyptus, https://www.eucalyptus.cloud, accessed 3 December 2019 (2018).

[264] K. Hightower, B. Burns, J. Beda, Kubernetes: up and running: dive into the future of infrastructure, O'Reilly Media Inc., 2017.

[265] The Linux Foundation, Kubernetes, https://kubernetes.io, accessed 3 December 2019 (2019).

[266] B. Hindman, et al., Mesos: A platform for fine-grained resource sharing in the data center, in: NSDI, Vol. 11, 2011, pp. 22–22.

[267] The Apache Software Foundation, Apache Mesos, http://mesos.apache.org, accessed 3 December 2019 (2018).

[268] G. Toraldo, Opennebula 3 cloud computing, Packt Publishing Ltd, 2012.

[269] OpenNebula Project (OpenNebula.org), OpenNebula, https://opennebula.org, accessed 3 December 2019 (2019).

[270] O. Sefraoui, M. Aissaoui, M. Eleuldj, OpenStack: toward an open-source solution for cloud computing, International Journal of Computer Applications 55 (3) (2012) 38–42.

[271] OpenStack Foundation, OpenStack, https://www.openstack.org, accessed 3 December 2019 (2019).

[272] Red Hat Inc., RedHat OpenShift, https://www.openshift.com, accessed 3 December 2019 (2019).

[273] X. Wen, et al., Comparison of open-source cloud management platforms: OpenStack and OpenNebula, in: 2012 9th International Conference on Fuzzy Systems and Knowledge Discovery, 2012, pp. 2457–2461.

[274] D. Milojičić, I. M. Llorente, R. S. Montero, Opennebula: A cloud management tool, IEEE Internet Computing 15 (2) (2011) 11–14.

[275] Platform 9, Kubernetes and Docker Swarm Compared, `https://platform9.com/blog/kubernetes-docker-swarm-compared`, accessed 3 December 2019 (2017).

[276] Platform 9, Kubernetes and Mesos Compared, `https://platform9.com/blog/compare-kubernetes-vs-mesos`, accessed 3 December 2019 (2016).