Active Data - Enabling Smart Data Life Cycle Management for Large Distributed Scientific Data Sets

Anthony Simonet

THÈSE

en vue de l'obtention du grade de

Docteur de l'Université de Lyon, délivré par l'École Normale Supérieure de Lyon

Discipline : Informatique

Laboratoire de l'Informatique et du Parallélisme

École Doctorale en Informatique et Mathématiques de Lyon

présentée et soutenue publiquement le 8 juillet 2015

par Monsieur Anthony SIMONET

Active Data: Enabling Smart Data Life Cycle Management for Large Distributed Scientific Data Sets

---

Directeur the thèse : M. Gilles FEDAK

Après l'avis de :

M. Christophe CÉRIN, Professeur, Université Paris 13

M. Douglas THAIN, Associate Professor, University of Notre Dame

Devant le jury composé de :

M. Christophe CÉRIN, Professeur, Université Paris 13, Rapporteur

M. Gilles FEDAK, Chargé de Recherche Inria, LIP/Inria Rhône-Alpes, Directeur

M. Mohand-Saïd HACID, Professeur, Université Claude Bernard Lyon 1, Examinateur

M. Johan MONTAGNAT, Directeur de Recherche, Université de Nice Sophia Antipolis, Examinateur

M. Douglas THAIN, Associate Professor, University of Notre Dame, Rapporteur

Remerciements

Tout d'abord, je souhaite remercier chaleureusement Gilles Fedak pour m'avoir offert l'opportunité de vivre cette expérience, m'avoir conseillé, encouragé souvent et freiné parfois. Je me souviens avec une vive émotion de nos discussions enflammées à l'inégalée productivité et de ses intuitions qui m'auront sans doute faites gagner plusieurs années sur ma thèse.

Je remercie Christophe Cérin et Douglas Thain pour leur lecture attentive de mon manuscrit et leur rapports. Je remercie également Mohand-Saïd Hacid et Johan Montagnat d'avoir bien voulu faire parti de mon jury.

Je tiens bien sûr à remercier ma famille et en particulier mes parents, sans qui je ne serais pas ici et encore moins *là*. Vous m'avez appris à travailler par plaisir et toujours encouragé à suivre ma voie, sans jamais questionner mes choix. Merci à sœur Laura, à Bruno et à la merveilleuse Louisa qu'ils m'ont offerte. Elle va sans doute rendre la suite encore plus intéressante. Merci bien sûr à Sébastien de partager ma vie et merci à Gabrielle, André et Guillaume de m'avoir accepté dans la leur. Je n'arrive pas à imaginer comment j'aurais pu vivre cette aventure sans vous tous.

Enfin je remercie très chaleureusement le LIP, l'équipe Avalon et son Chef Christian Perez de m'avoir accueilli. Je sais que cela n'a pas été facile tous les jours et votre patiente a été appréciée justement. En plus de remerciements, je dois sans doute des excuses à ceux qui ont partagé un bureau avec moi : Daniel, Arnaud et Laurent. Je dois sans doute aussi des excuses à ceux qui ont préféré partir et n'ont donc partagé qu'une partie de l'aventure : Julien, Cristian, Adrian, George, Florent, Jonathan, Landry, Noua, Guillaume et François. Pour finir, je remercie les valeureux que je j'ai pas fait fuir : Evelyne, Laurent L., Fred. L et Fred. S, Simon, Matthieu, Vincent, Pédro, Violaine, Hélène, Jérôme, Marcos, Semen et Radu.

« C'est fou tous ces gens. J'espère que je n'en ai pas oublié. »

Abstract

In many domains, scientific progress relies more and more on our ability to exploit ever growing volumes of data. However, when data grows, so does the complexity of managing it. A key point is to deal with the complexity of data life cycle management, i.e. the various operations applied to data from their creation to their deletion: transfer, archival, replication, deletion, etc. These formerly straightforward operations become intractable when data volume grows dramatically, because of the heterogeneity of data management software on the one hand, and the complexity of the infrastructures involved on the other. In this context, cooperation between different systems becomes very complex and requires ad-hoc solutions and many human interventions.

This thesis contributes theoretical and practical tools that allow a rigorous and efficient approach to data life cycle management in large scientific applications. To this end, we study the software tools, the programming models and the infrastructures commonly used by these applications. From this analysis, we devise a set of features that must be taken into account for modeling the life cycle of data in scientific applications.

Our first contribution is a meta model that allows for the first time to represent both formally and graphically the life cycle of data distributed not only in a system, but in a set of systems, on heterogeneous infrastructures. It allows to formalize, analyze and share the life cycles, naturally exposing replication, distribution and the different data identifiers.

We then present *Active Data*, an implementation of this meta model. Once connected to existing applications, Active Data exposes the progress of data at runtime to users and programs. Active Data also keeps track of data when they pass from a system to another, creating a unique high-level view and a global namespace.

Our third contribution is a programming model that allows to execute code at each step of the data life cycle. Active Data programs have access to the complete state of data, no matter in what system or on what infrastructure they are distributed on. These programs can make local decisions based on a global knowledge and implement numerous optimizations that could not be achieved until now.

We present micro-benchmark performance evaluations and use-cases that demonstrate the expressivity of the programing model and the implementation quality. Our last contribution is a *Data surveillance framework* for the APS (*Advanced Photon Source*) implemented with Active Data and that allows scientists to monitor the progress of their data, to automatize most manual tasks, to obtain notifications and to detect and recover from many errors with no human intervention.

This work has promising perspectives in the field of Data Provenance and Open Data, while facilitating collaboration between scientists from multiple communities.

Résumé

Dans tous les domaines, le progrès scientifique repose de plus en plus sur notre capacité à exploiter des volumes toujours plus gigantesques de données. Cependant, alors que le volume des données croit, leur gestion s'en complexifie d'autant. Un point clé est de gérer la complexité de la gestion du cycle de vie des données, c'est à dire les diverses opérations qu'elles subissent entre leur création et leur disparition : le transfert, l'archivage, la réplication, la suppression, etc. Ces opérations, autrefois simples, deviennent ingérables lorsque le volume des données augmente de manière importante à cause de l'hétérogénéité des logiciels utilisés d'une part, et à cause de la complexité des infrastructures mises en œuvre d'autre part. Dans ce contexte la coopération entre différents systèmes est très compliquée et nécessite des solutions sous-optimales et de nombreuses interventions humaines.

Le but de cette thèse est de proposer des outils théoriques et pratiques pour permettre une approche rigoureuse et efficace de la gestion du cycle de vie des données dans les grandes applications scientifiques. À cette fin, nous analysons les outils logiciels, les modèles de programmation et les infrastructures distribuées communément utilisés par ces applications. De cette analyse, nous déterminons les éléments à prendre en compte pour modéliser efficacement le cycle de vie des données dans les applications scientifiques.

Notre première contribution est un méta-modèle qui permet pour la première fois de représenter formellement et graphiquement le cycle de vie de données présentes non seulement dans un système, mais également dans un assemblage de systèmes sur des infrastructures hétérogènes. Il permet de formaliser, d'analyser et de partager le cycle de vie, en exposant naturellement la réplication, la distribution et les différents identifiants des données.

Ensuite, nous présentons *Active Data*, une implémentation de ce méta-modèle. Une fois connecté à des applications existantes, Active Data expose l'état d'avancement des données au cours de l'exécution à des utilisateurs et à des programmes. Active Data garde également trace des données lorsqu'elles passent d'un système à un autre, créant un espace de nom global.

Notre troisième contribution est un modèle de programmation qui permet d'exécuter du code à chaque étape du cycle de vie des données. Les programmes écrits avec Active Data ont à tout moment accès à l'état complet des données, à la fois dans tous les systèmes et dans toutes les infrastructures sur lesquels elles sont distribuées. Ces programmes peuvent donc prendre localement des décisions basées sur une connaissance globale, et ainsi implémenter de nombreuses optimisations jusqu'alors impossibles.

Nous présentons des évaluations de performance et des cas d'utilisations qui attestent l'expressivité du modèle de programmation et la qualité de l'implémentation. Notre dernière contribution est un outil de *Surveillance des données* pour l'expérience APS (*Advanced Photon Source*) implémenté avec Active Data et qui permet aux scientifiques de suivre le progrès de leurs données, d'automatiser la plupart des tâches manuelles, d'obtenir des notifications et de détecter et corriger de nombreuses erreurs sans intervention humaine.

Ce travail présente des perspectives très intéressantes, en particulier dans les domaines de la provenance des données et de l'open data, tout en facilitant la collaboration entre les scientifiques de communautés différentes.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

We live in the era of Big Data. Over the last decade, data has become the raw material of knowledge. Increasingly, and in many fields, from physics to human sciences, scientific discoveries are fueled by our ability to acquire, filter, transfer, analyze and share phenomenal quantities of data. As such, tremendous efforts have been made by scientific and industrial communities to elaborate algorithms, hardware and computing infrastructures to enable refining data sets, distillate information and extract precious knowledge. But scientific and industrial data sets grow at an alarming rate; novel scientific instruments like CERN's LHC [1], the Large Synoptic Survey Telescope [2], the OOI Cable System [3] and large scale simulations produce Petabytes of data every year. Online services, pervasive computing, retail transactions, visitor browsing traces, billion-edge social graphs are fast and steady sources of data that have to be indexed, curated, stored and analyzed.

Managing increasingly large data sets calls for high-performance machines with a lot of storage space, memory and high-speed CPU. As such, handling today's data sets always calls for distributed approaches. Current infrastructures composed of many commodity machines are cheaper than traditional super computers but are also less reliable [4]; hardware failures are expected daily and the burden of coping with them has been transferred to the software stack. Additionally, data sets are dynamic: they can grow or shrink over time and get partially updated. As a result, data intensive experiments and applications have become too complex for many programmers: parallel and distributed computing, synchronization, hardware failures recovery are notoriously difficult problems that often produce poor design, performance, resources utilization and often erroneous results.

In addition to the difficulties coming from the infrastructures, a great deal of the big data challenge also comes from operations that are fairly trivial on reasonable-size data, but become impractical when data grow too large. Transferring large data is incredibly slow and many errors are to be expected; the indexing and the filtering of big data must be distributed because data sets cannot fit on a single machine; replication between multiple distant sites must be coordinated to avoid data loss and allow descent access time to big data; storing big data requires sophisticated systems that make up for poor disk performance, and so forth. Part of this complexity is alleviated by novel distributed storage systems, frameworks, programming models, workflow systems and tools, which abstract low-level details like failure detection, and retries. However when it comes to managing large and dynamic data sets, there is no "one-size-fits-all" solution; applications involve most of the time a fragile assemblage of software systems that were not designed to collaborate, are glued together using ad-hoc techniques and require

human intervention from time to time. When data keep growing there is a point where so many things happen at the same time that users get overwhelmed and detecting the small things that go wrong in the mass of things that go right become impractical.

We call *Data Life Cycle* all these operations that are performed on data from their creation to their deletion e.g, acquisition, transfer, filtering, analysis, replication, storage and archival; we call *Data Life Cycle Management* (DLCM) the orchestration of these operations, whether with a simple script or with a more sophisticated workflow system. DLCM is unavoidable and requires increasing efforts, which makes it the corner stone of data-intensive science. Efficient and safe DLCM will only become more decisive in the near future, and appropriate tools and techniques must be developed to alleviate it and let scientists focus on their science.

This thesis defends the idea that most of the difficulty of DLCM comes from the infrastructures on the one hand, and from the applications on the other. Removing this difficulty requires a constant flow of information between both: software must be able to react to infrastructure events on their own, while the infrastructure must adapt to application needs. Collaboration between heterogeneous systems must allow inter-system optimizations and error recovery with no user intervention. Moreover, DLCM tasks must be easy to program, for everyone. In this spirit, this thesis proposes *Active Data*, a meta model that allows for the first time to formally represent the life cycle of data that are distributed on multiple systems and heterogeneous infrastructures. From this meta model, that allows to analyze and share data life cycles, we develop the *Active Data* programing model. This novel programming model enables users to *observe* the progress of their data and to automatically execute custom code at key steps in the life cycle. It offers a unique, high-level view of all the copies of a single data item while it is present in multiple, non-collaborative systems. Using this high-level view, one can learn the state of their data at any time in the most complex applications, identify potential bottlenecks and easily program applications that manipulate large distributed data sets.

1.1 Objectives

Considering the challenges due both to heterogeneous distributed infrastructures and data life cycle management systems, this thesis aims at offering a truly data-centered formalism, runtime system and programming model that facilitates monitoring, decision making, running management tasks and overall helps preserve the quality of data assets.

We elaborate the objectives of this thesis as follows:

1. To analyze the state-of-the-art in distributed data management, workflow systems and programming models to point out their limitations and propose an approach to make distributed data management smarter and easier;
2. To provide a meta-model for representing and exposing the inner life cycle of any data management system, the end-to-end life cycle of data passing through multiple systems and a model for integrating data identifiers into a single namespace;
3. To propose an implementation of the meta-model that allows the life cycle of distributed data to be recorded and examined by users and programs;
4. To propose a programming model that allows to develop data management applications by reacting to life cycle events, using the meta-model implementation;

5. To evaluate the efficiency of the runtime system and the expressivity of the programming model with a series of benchmarks and pertinent use-cases;

6. To integrate and evaluate the runtime system in a real life data-intensive application.

1.2 Contributions

The main contributions of this thesis are presented in this section.

A meta-model for distributed data life cycle The first contribution of this thesis is the first formal data-centric meta model for representing the life cycle of distributed data sets simultaneously present in multiple, heterogeneous and non-collaborating systems. The meta model captures every operation applied to data, from end-to-end, and naturally represents the distribution of data. Finally, the meta model formally represents the flow of data between systems, keeping track of data identifiers, origin and lineage.

A implementation of the meta-model The second contribution is *Active Data*, an implementation of the meta model that offers programs the same view that the graphical model offers to users. Active Data allows users and programs to *observe* data evolving at runtime inside applications, allowing advanced optimizations. It also keeps track of data identifiers as data pass from a system to another, constructing a single logical namespace for multiple copies (or replicas) of data in different systems that would otherwise look completely unrelated. Overall, the implementation of the life cycle meta model opens the door to many optimizations related to distributing data on a platform or when recovering from failures; for example, now a system can *look* beyond its scope and learn where additional copies of lost data are located.

The Active Data programming model The third important contribution of this thesis is a programming model that is based on the life cycle meta model and its implementation. The Active Data programming model and the associated runtime system allow users of data management systems to have code automatically executed when events on data happen. It can be used to program essential data management operations, to automatically capture provenance information, to optimize inter-system collaboration and to recover from failures that are hard to detect and used to require human intervention.

Use-cases and evaluation A set of synthetic benchmarks run on the Grid'5000 experimental testbed show the performance one can expect from the implementation and a set of use-cases evaluate the expressivity of the model with typical data management situations.

Data surveillance framework The last contribution of this thesis is an application to the *Advanced Photon Source* (APS) experiment that involves multiple data management systems and infrastructures. The application constructs a *Data Surveillance Framework* for the APS that allows scientists to monitor the progress of their experiments, to automate most manual tasks, to automatically notify the community of newly

produced data sets and share them, and finally to discover errors involving multiple systems and automatically recover from them. This work has been done in collaboration with Ian Foster and Kyle Chard during a visit at the University of Chicago and Argonne National Lab.

1.3 Structure of this Manuscript

This manuscript is organized as follows. Chapter 2 reviews the literature on infrastructures, techniques, programming models and systems for large-scale distributed data management. Chapter 3 introduces our approach, Active Data, through its meta model, implementation, and its programming model. Active Data is evaluated with micro-benchmarks and synthetic use-cases in Chapter 4. Chapter 5 relates how Active Data was used to implement a Data Surveillance Framework for an e-Science experiment. We present conclusions and perspectives in Chapter 6.

1.4 Publications

The work presented in this thesis has been published in several research papers that are listed hereafter.

International Journals

- Anthony Simonet, Gilles Fedak, and Matei Ripeanu. Active Data: A Programming Model to Manage Data Life Cycle Across Heterogeneous Systems and Infrastructures. *Future Generation Computer Systems*, page 49, 2015

- Gabriel Antoniu, Julien Bigot, Christophe Blanchet, Luc Bougé, François Briant, Franck Cappello, Alexandru Costan, Frédéric Desprez, Gilles Fedak, Sylvain Gault, Kate Keahey, Bogdan Nicolae, Christian Pérez, Anthony Simonet, Frédéric Suter, Bing Tang, and Raphael Terreux. Scalable data management for mapreduce-based data-intensive applications: a view for cloud and hybrid infrastructures. *International Journal of Cloud Computing*, 2(2):150–170, 2013

International Conferences

- Anthony Simonet, Kyle Chard, Gilles Fedak, and Ian Foster. Using active data to provide smart data surveillance to e-science users. In *Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed and Network-based Processing*, PDP 2015. IEEE, 2015

- Gabriel Antoniu, Julien Bigot, Christophe Blanchet, Luc Bougé, François Briant, Franck Cappello, Alexandru Costan, Frédéric Desprez, Gilles Fedak, Sylvain Gault, Kate Keahey, Bogdan Nicolae, Christian Pérez, Anthony Simonet, Frédéric Suter, Bing Tang, and Raphael Terreux. Towards Scalable Data Management for Map-Reduce-based Data-Intensive Applications on Cloud and Hybrid Infrastructures. In *1st International IBM Cloud Academy Conference - ICA CON 2012*, Research Triangle Park, North Carolina, 2012

Workshop

- Anthony Simonet, Gilles Fedak, Matei Ripeanu, and Samer Al-Kiswany. Active data: a data-centric approach to data life-cycle management. In *Proceedings of the 8th Parallel Data Storage Workshop*, pages 39–44. ACM, 2013

National Conference

- Anthony Simonet. Active data: Un modèle pour représenter et programmer le cycle de vie des données distribuées. In *ComPAS'2014*, 2014

Research Report

- Anthony Simonet, Gilles Fedak, and Matei Ripeanu. Active Data: A Programming Model to Manage Data Life Cycle Across Heterogeneous Systems and Infrastructures. Research Report RR-8062, Inria - Research Centre Grenoble – Rhône-Alpes ; ENS Lyon ; University of British Columbia, 2015

2 Large scale, Data Intensive Distributed Computing

2.1 Data Intensive Applications

While very large data sets are at the heart of many scientific events and are subject to an increasing number of research projects, the very definition of *Big Data* is still hard to capture because of the lack of consensus on the matter. Beyond what is big data and what is not, this first section highlights research challenges the computer science community and other scientific communities are facing together.

2.1.1 Big Data

The definition of big data is still being actively discussed [12–14]. The "3 Vs" are however part of many definitions. They are *i)* Volume: big data implies large volumes of data; although the line at which big data starts is not clearly drawn, it seems to be somewhere between a few Terabytes a few Petabytes of data. *ii)* Velocity: the speed at which data is generated or acquired is extremely fast and requires short cycles to produce interpretable outcomes. *iii)* Variety: data comes in many forms, either structured, semi-structured or unstructured (e.g, image and video files).

A fourth V for *Value* appeared later in the literature and is now regarded as an essential component of big data. Big data have the potential for very high value, but its extraction requires considerable work ("huge value but very low density" [12]). Big data can be regarded as raw material that has no value on its own and gets value after being transformed into refined information. One could say that industries that get a lot of data "for free" (click and visitor trackers, user ratings and comments, social media integration, search engines keywords etc.) but do not exploit it are sitting on a gold mine. The same is true for science where discoveries depend more and more on the exploitation of huge data sets produced by large scale simulations, new scientific instruments and sensor networks.

We now understand that there can be no empirical definition of big data, and that big data is not necessarily *large*; instead it seems to start with a matter of scale. In [15], Manovich explains how yesterday's big data can today fit on a single laptop computer. Thus, data may simply become *big data* when it is large and fast enough so that traditional computer systems cannot process it anymore. Or simply put, data is big data when it is thrown at us so fast that our methods and systems cannot keep up.

2.1.2 The Origins of Data

A first source of very large data sets is the Internet and pervasive online services that users all over access on multiple devices from all over the world. Big data is constituted from the data they produce and submit willingly (social network entries, customer reviews, shared pictures and videos, files stored by online services, emails) and data they create inadvertently (search queries, browsing habits, online advertisement, leaving breadcrumbs on visited websites that seem to have nothing in common.

Internet companies and social networks in particular create billion-edge graphs that need be be analyzed to extract sense and restitute it to engaged users.

Moreover, new sources of big data are appearing, like the Internet of Things (IoT) [16, 17]. IoT designates a paradigm where a huge number of ubiquitous sensor-enabled devices (home appliances, smartphones, medical devices, environmental sensors) record data about their environment and user interactions and daily life; the recorded data is sent to remote services that are in charge of the heavy computation. These devices may record data about virtually anything from air quality, to seismology, to traffic and parking spots, to medical data and user whereabouts. This *everything connected* fashion creates a highly dynamic and distributed production of big data that match the four Vs: huge volumes are produced fast by heterogenous devices, hence various formats and they hold high value at low density.

2.1.3 Challenges

Managing multi-petabyte data sets naturally calls for distribution and deduplication. In this context, maintaining the quality of service users are used to (quick access, high bandwidth, rare data loss, etc.) is challenging. We need new algorithms to filter, transfer, split, store, index, version and deduplicate data that scale to thousands of machines.

Thus, big data also calls for new infrastructures that fit big data properties; infrastructures for big data must be always on to keep-up with incoming data; they must be elastic to accommodate variations in data sizes; their compute and I/O performance must satisfy big data workloads. At the same time adding disks and computers to a system automatically increases the number of faults that happen daily. We have reached a point where hardware faults are just expected and where the software stack needs to cope with them.

Big data involves more complicated life cycles. Data often arrive faster than it can be processed; at the same time, the value of data depend on the speed at which it is exploited. This means decisions about which data to keep and which data to discard, filtering and many life cycle operations must be made as fast as possible, despite their volume. New programming models are also necessary to help programers and data scientists focus on analytical code and program applications as easily as for traditional data sets.

Many large data sets involve several communities that can collaborate to extract information from data or seek different results. Geographically dispersed communities require major advanced in networks to move these data sets and social tools to support information sharing and lineage tracking. These tools must be optimized to cope with the size and variety of big data (structured, semi-structured and unstructured).

2.2 Cyber-Infrastructures for Big Data

The emergence of big data science has driven the need for larger storage and computing infrastructures. In this section, we introduce the most common cyber-infrastructures found in companies and research institute to acquire, store and process large data sets.

2.2.1 Data Centers

The emergence of big data applications has lead companies and public institutions to invest in large computing clusters specially dedicated to storing and exploiting large volumes of data. Data centers are composed of multicore machines grouped in clusters; clusters are closely located and connected with a high performance Local Area Network (LAN). All machines in a cluster—also called *nodes*—share the same hardware and most of the time run the same operating system. Big data has driven the need for larger data centers that can fit up to hundreds of thousands of nodes. As the number of nodes increases, even high-end hardware is subject to daily failures of some of the nodes, which made the case for using cheaper hardware, leading to what is now commonly know as commodity clusters [18, 19]. Google clusters [4] form a good example of very large data centers composed of commodity computers and of software robust to many hardware failures.

Data centers offer premium facilities to companies and universities, but their high cost make them unaffordable to the smaller ones. Managing a data center and running jobs require specialized software to monitor machines and to accommodate the usage of several users. This software layer, often called *batch scheduler* is a critical middleware to guarantee the availability and the security of the platform. Thus, hardware and software maintenance requires trained administrators that adds up to operating costs.

2.2.2 Data Grids

Data science had driven the need to share and transport large data sets over distant location, leading to the development of worldwide *Data Grids* [20, 21]. A data grid is an integrated architecture that coordinates access to heterogeneous resources from distant data centers into a *Virtual Organization*. A data grid federates storage, compute resources and services from distant sites managed by different institutions and connected by a Wide Area Network (WAN). Services store, index, cache, replicate, transfer and serve data transparently for user's applications [22]. Replication, which is used for fault tolerance and I/O performance [23] (improving bandwidth and locality), hides the different storage layers and naming conventions of each sites so users only see a unique global namespace. Data grids thus rely on traditional institutional grid infrastructures—originally defined in [24], [25] and [26]—on top of which they build a set of data-oriented services.

The *EU Data Grid Project* [27,28] has been one of the first implementations of data grid; it was built to support I/O intensive experiments High-Energy Physics, Earth Observation and Biology. CERN's *Worldwide LHC Computing Grid* [29] supports experiments with the 30 Petabytes of data generated by the Large Hadron Collider annually; it currently includes 170 sites in 41 countries.

2.2.3 Clouds Platforms

Cloud computing is a paradigm in which virtually-unlimited resources are accessed on-demand, in a scalable way, and abstracted with virtualization. In clouds, resources are remote, administered by third parties and available almost as a "public utility", quoting Foster et al. in [30]. Cloud computing features several layers of abstractions on the resources they provide, enabling virtually any application to run [31]; *Infrastructure-as-a-Service* (IaaS) provides users with virtual machines and a virtual network on which they can run any application; *Platform-as-a-Service* (PaaS) provides users with abstractions (libraries and software components) to program applications and an execution platform to run them transparently; *Software-as-a-Service* (SaaS) provides ready-to-use web applications and services.

Data Grids require scientists to join into important multi-year projects and expensive administrative, development and funding efforts. Many institutions and research groups cannot federate a large enough community to get involved in a grid project; others could simply not afford it. To these scientists, cloud computing [32–34] has given access to huge computing and storage resources at an affordable cost (with no initial investment necessary). With their on demand fashion, flexible computing resources and data-oriented services available in *pay-as-you-go*, clouds have arguably a lot in common with the idea people used to have of the grid [30].

2.2.4 Desktop Grids

Desktop grids are an other solution for scientists with big resource needs but less money at the cost of more heterogeneity and a more complex fault model. Desktop grids aggregate the computing power and/or storage resources of idle desktop computers. The best illustration of a successful desktop grid is probably the SETI@home project [35] that performs at more than 1,800 TeraFLOPS [36] using the *BOINC* middleware [37]. The resources a desktop grid aggregates can belong to a single organization, like a university or a company; they can also be volunteered by individuals, which is why this approach is also referred to as *volunteer computing*. *BonjourGrid* [38] is a system that allows the deployment and the orchestration of several desktop grid middleware.

Desktop grids have traditionally been limited to Bag of Task (BoT) applications that are compute intensive, require few input data and produce small output data and with low quality of service requirements. Current research demonstrate successful attempts at implementing more sophisticated applications and programming models like the *Moon* [39] MapReduce implementation.

2.2.5 Hybrid Infrastructures

There are big incentives to run data and I/O intensive scientific applications on hybrid infrastructures, i.e. infrastructures that federate the resources of other infrastructures. Combining local and remote resources like a grid or a public cloud is a use case that allows to meet QoS constraints at a controlled cost. When using desktop grids, there is obviously a tradeoff to make between performance, quality of service and cost that is illustrated by *SpeQuloS* [40]. SpeQuloS proposes to use volunteered hosts as much as possible, while using paid cloud instances when necessary to meet some QoS constraints for Bag of Task applications. Another frequent scenario is to have a desktop grid scheduler service running on a private cluster to coordinate volunteered nodes, while storing executables and scientific data in the cloud for better availability, lower latency

Platform type	Administration	Initial cost	Usage cost	Homogeneity
Data center	Institution	+++	++	Yes
Data Grid	Institution (locally) & Grid project (globally)	+++	++	No
Cloud	Third party	–	++	Yes
Desktop Grid	DG project	+	–	No
Hybrid	Depends on the platforms used			No

Table 2.1: Comparison of distributed infrastructures in terms of administration effort, cost and homogeneity.

and resilience. Table 2.1 illustrates this tradeoff and compares infrastructures in terms of administration effort (who manages the platform), initial and usage cost from "–" (near zero) to "+++" (very expensive) and homogeneity of resources.

While hybrid infrastructures are interesting for their flexibility and cost, implementing them require unified interfaces and tools for accessing resources. Currently, there seems to be no such interface and users tend to use ad-hoc solutions that produce non-reusable code and heavy maintenance efforts.

2.3 Data Management Systems

Large volumes of data automatically require to distribute storage and management tasks to many disks and machines. Many strategies for distributed storage have been studied, considering performance, fault tolerance needs, the type of queries to support, the size of updates to optimize and much more. In this section we discuss research in parallel and distributed data management and storage systems, and how they compromise to optimize reliability, access time, concurrency, consistency and querying.

2.3.1 Distributed Filesystems and Distributed Datastores

Big data science is new and often supported by scientists that are not computing experts. To facilitate access to data sets, many sophisticated distributed storage systems feature a POSIX filesystem interface. They allow the storage system to be used like any local filesystem while offering large storage aggregated from multiple servers, usually through a network [41]. They allow users to use remote storage space transparently: the true location of files is hidden and users are presented a unique global namespace that can be used directly by applications; this feature is called *naming virtualization*. Distributed filesystems can also transparently implement replication for performance or fault tolerance with various placement strategies.

Distributed filesystems face two challenges in big data applications. First, some client applications need to handle many small files, leading to the file-count problem [42]: the space for storing metadata can exceed the space used by the actual data, and the sum of parallel operations can exceed the load of I/O operations. Second, distributed filesystems often have to handle unusually large files; they can come either as a mean of working around the file-count problem, packing small files into large blobs of unstructured data, or because of particular application needs.

Scalability can be improved by separating metadata operations that are related to the namespace (open, rename) from I/O operations (read, write). This is what *Ceph* [43], *GFS* [44] and *HDFS* [45] do; they replace traditional disks with Object Storage Devices (OSD) that comprise a disk, a controller and network; OSD are able to serve I/O operations directly, leaving namespace management to a *master*. In addition, GFS and HDFS now have distributed masters able to manage up to hundreds of millions of files each. Ceph and *Lustre* [46] removed the need for a master for some operations; replacing inode and object lists by hash functions, they allow agents to discover the physical location of a data block without having to query a centralized catalog; with this architecture, clients can negotiate operations directly with the OSD that host the object they want to access.

All these systems now seem to agree that while the number of machines and disks grow with computing infrastructures, "component failures are the norm, not the exception" [43, 44]. Thus, they tend not to rely on traditional protections like RAID, but to consider instead every disk as unreliable. Replication, used by both GFS and HDFS allows to implement load balancing and reliability. Other techniques for building reliable storage from unreliable hardware requires significantly more complex code for storage systems. Replication, load balancing and consistency checks are hidden from the user, so they can use the distributed file system in a similar way as they use a local filesystem. *Chirp* [47] is an other distributed filesystem that offers a POSIX compatible interface and is suited for clusters and grids, and support Wide Area Networks. Another interesting feature of Chirp is that it runs in user space, allowing to deploy it on top of desktop grids, which makes it suitable for hybrid infrastructures that comprise desktop grids.

The *FUSE* [48] (Filesystem in Userspace) project has helped developers of distributed datastore offer a POSIX-compliant interface with few effort. FUSE is a Linux kernel module that allows programs to emulate filesystems in user space with no administrative privileges. Thanks to this feature, it is well suited for deployment of complex storage systems on hybrid infrastructures that comprise desktop grids where programs run with no privilege. FUSE is used by Ceph and Chirp to provide their POSIX interface.

However, offering a POSIX-compliant interface often comes in the way of performance; this is why many storage systems, while keeping a semantic close to file systems, do not implement standard interfaces. These storage systems are commonly called *distributed datastores*. Distributed datastores already cited include GFS and HDFS. *BlobSeer* [49] is another distributed datastore that is optimized for large unstructured blobs and write-once/read-many applications using versioning as primary concurrency control.

2.3.2 In-Memory Datastores

In many big data applications, short-lived files are used for inter-process communication (the result of a process is used as input to another process), caching and checkpointing. While these files can be lost and regenerated, their access time is critical to applications performance. In-memory storage is a good compromise for this class of files; the bandwidth of memory is orders of magnitude larger than disks and the latency of memory is orders of magnitude faster. At the same time, volatility is not a problem since the files must only live for the duration of the application.

MosaStore [50] is an in-memory "versatile storage system" backed up by disk storage

aggregated with *GPFS* [51]. MosaStore aggregates underutilized resources of cluster or desktop nodes; it adapts to specific application needs by satisfying per-file constraints at deployment time. These constraint are expressed in terms of access patterns (broadcast, scatter, reduce, etc.) and in turn MosaStore optimizes file placement and replication [52]. A prediction mechanism was also studied in [53] to automatically adapt the storage configuration of I/O intensive workflows.

RAMClouds [54] also proposes to store flies in RAM for performance but attempts to extend the application scope of in-memory distributed datastores by asynchronously persisting data to disks. *MemCached* [55] uses a hash table design where each aggregated node is a bucket. A single machine reads client requests, performs the lookup and directs clients to the machine that will handle the read or write request; this design allows Memcached to scale up to dozens of nodes with constant complexity under heavy write load.

2.3.3 Databases for Big Data

The large volumes of big data does not fit well in traditional relational databases because data is often semi-structured or unstructured and supporting a schema makes data grow even larger. To support big data, database management systems have evolved in two ways: relational databases can now be distributed and systems storing data in an unstructured way have appeared.

Just like distributed filesystems, distributed databases had to relax some properties—data model and consistency, for example—to horizontally scale to the need of current data-intensive applications.

2.3.3.1 Distributed Relational Databases

Driven by industrial needs on one hand, and the ease to parallelize SQL queries on the other, traditional relational databases evolved to a distributed paradigme in the late 1980's [56]. Parallelism allowed Relational Database Management Systems (RDBMS) to handle larger volumes of data, to serve numerous concurrent queries [57] and transactions while maintaining ACID properties (*Atomicity, Consistency, Isolation and Durability*) [58]. Distributed databases comprise several Database Management Systems instances controlled by a coordinator. The distributed DBMS instances can be used for performance—executing queries in parallel, redirecting queries to less loaded instances—or availability and reliability through replication [59]. The effective distribution of data objects amongst the DBMS and the strategies to redirect queries attempt to optimize one or several of these aspects.

However, unstructured data sets cannot fit in a relational database that requires a strict schema. Importing even semi-structured data sets proves to be unpractical due to the lengthy conversion and disk space overhead [15] induced by the relational model.

2.3.3.2 NoSql Databases

NoSql databases have appeared only recently to store unstructured big data for which the relational model is inefficient and burdensome. Document, object and key-value DBMS implement storage, transaction and query models that allow complex and interlinked data structures to be accessed in a natural way by data-intensive applications while enabling horizontal scalability. However, this shift comes at the price of by re-

laxing ACID properties for the less restrictive BASE (*Basic Availability, Soft state, Eventual consistency*) [60].

While document databases allow to store arbitrary documents, either structured or unstructured, the trend in big data applications seems to be large unstructured data sets. Often coded as XML or JSON, these documents can be media files (sound, image, video), log files, text documents (blog entries, Tweets) and streams. *MongoBD* [61] (open-source) and *CouchDB* [62] (commercial) are two examples of database systems storing arbitrary documents in JSON and in which additional nodes can be added on demand to to satisfy the need of data-intensive applications [60].

Key-value store are also increasingly attractive thanks to their *hash-map-like* pattern where values can often be arbitrary byte strings. *Redis* [63], *Voldemort* [64] (open-source) and *DynamoDB* [65] (commercial) are good examples of this trend.

Google's Bigtable [66] is a distributed datastore that could be qualified of "hybrid"; it looks like a distributed database management system in the way it supports structured data. Bigtable organizes data in rows and tables but does not support relations. These properties, paired with versioning allow for a efficient management of very large data sets with many small updates. However, the data model similar to a three-dimensional map indexed by row, column and timestamp make it look more like a key-value store. *Hbase* [67], maintained by the *Apache Software Foundation* is an open-source project that provides BigTable like features, allowing to store semi-structured or unstructured data on top of HDFS [45].

2.3.4 Rule-Based Systems

Increased data set sizes implies more administrative tasks to control and more access policies to enforce, as managing and the exploiting large data sets often involve multiple programs that perform operations independently. These programs must fulfill a set of *policies* defined by the organization; to satisfy these policies, programs typically perform consistency checks on their own, ensuring that they leave data in a valid state when they return. However, placing these checks at the application level forces administrators to maintain many different pieces of code, adds a burden when replacing a program with another and may leave data inconsistent due to application failures. Rule based storage systems address these issues by moving consistency checks and policy enforcement operations from the programs to the storage system.

A rule-based system allows administrators to define *micro services*—consistency checks and other housekeeping tasks—and the conditions in which to run them. The rule engine is requested—either periodically or occasionally—to invoke micro services. Micro services are actually invoked only for data objects that match the defined conditions.

In this field, *iRODS* [68] occupies a prominent place thanks to a complete set of features with a distributed datastore with naming virtualization, a catalog for user-defined metadata and a domain-specific rule language for system and user-defined rules.

2.4 Transferring Large Data sets

Cyber-infrastructures involve moving large volumes of data between sites; some transfers are explicitly performed by scientific workflows: raw data are moved from the instruments to the storage infrastructure and then to the computing infrastructure and so forth; some transfers are performed silently by data management systems for load balancing and replication management, for example.

Classical protocols such as HTTP, FTP, SCP and rsync are still used by data intensive applications but mostly for small files, as they tend to be unreliable and scale poorly when the transferred files exceed a few gigabytes in size [69]. Thus, transferring large files requires new strategies to deal with errors and retries, use multiple channels in parallel and cope with unfriendly network conditions. The *BitTorrent* [70] Peer-to-Peer file transfer protocol addresses this large file problem by splitting them into smaller pieces called "chunks" that are easier to handle. Splitting files allows to verify the integrity of transferred chunks using checksums; this way, only chunks that differ between the origin and the destination are retried. This same approach (splitting and checksumming) is also used by rsync.

Considering that different files often need different transfer protocols due to their size, the transfer type (one-to-one, one-to-many etc.) or security concerns, *BitDew* [71] offers a single interface to several protocols (HTTP, FTP, SCP, BitTorrent, email, Saga [72]). BitDew allows to select a different transfer protocol for each file. *Stork* [73] focuses on data placement and also offers a single interface for several protocols; it is also able to choose which protocol suits best a particular situation. *GatorShare* [74] offers features similar to BitDew and offers a filesystem interface that eases application development.

To lift the burden of using these systems, that non-computer scientists would still consider as low-level, several SaaS operated services have appeared. *Globus Online* [75] is the SaaS counterpart of the *Globus toolkit* [76] and implements the *GridFTP* [77] protocol, offering a simple user interface for moving files between distant sites. GridFTP upgrades the FTP protocol for parallel high throughput and secure data transfers between remote sites. It eliminates part of network and disk contention by allowing files to be transferred from several sources simultaneously.

2.5 Programming Models for Distributed and Data Intensive Applications

Big data generates complex experimental workflows where data management occupies a large portion of the code. To make big data applications accessible to the mass, many programming models with a common goal have emerged: enabling users, scientists and companies to focus on their analytical code, abstracting everything; the literature now offers plenty of options for very specific application domains, so that programmers are free from thread programming, fault tolerance, remote procedure calls, data movements etc.

2.5.1 Programing Models with Implicit Parallelism

Writing efficient parallel and distributed code is notoriously challenging for computer scientists and non-computer scientists alike due to the inherent difficulty of concurrency, synchronization and efficient resource allocation [78]. Programing models and languages with implicit parallelism aim at automatically building parallel programs from sequential building blocks and as been discussed for a long time now [79]. A more recent challenge aims at adapting this approach to distributed applications, allowing programmers that are not experts in distributed computing to program distributed applications by focusing on what matters to them—their analytical code—and free them from low-level implementation concerns. As a result, implicit parallelism has started to appear

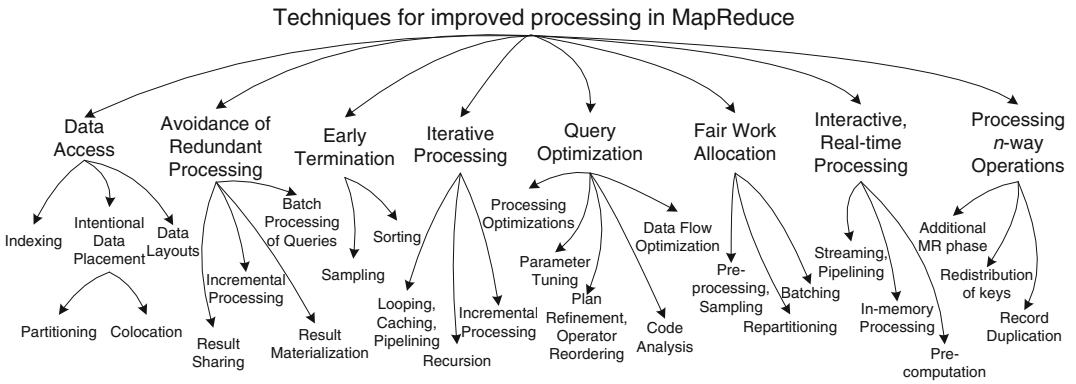


Figure 2.1: Taxonomy of MapReduce improvements for efficient query processing (source: Doulkeridis et al. [83]). Investigated improvement for covers everything from the storage layer to scheduling and some aspects of the programming model itself.

in programing models for distributed environments.

Implicit parallelism in the context of distributed computing is a powerful model in which users provide serial code or implement high-level interfaces, and then rely on a distributed runtime system to automatically distribute and run this code on a set of machines. Inter-process communication, threads and low-level synchronization, processes and data movements are also managed by the runtime environment. Thus, in addition to easing the development of data-intensive applications, this model often leads to a better usage of resources and safer code, the optimisation effort relying at the framework level.

2.5.1.1 MapReduce and Mapreduce-based programming models

MapReduce [80] introduced by Google in 2008 is probably the most famous and most widely used programming model featuring implicit parallelism. MapReduce allows users to program data processing applications by implementing only two primitives: "map" and "reduce". A MapReduce framework then runs the implemented code on a distributed infrastructure, tracks progress, retries failed tasks and deals with data movements and load balancing. There exists many frameworks implementing the MapReduce programming model; Google's MapReduce runs on top of a *GFS* [44] cluster, collocating storage and computation for efficiency; this strategy is also applied by *Hadoop* [81] which with *HDFS* [45] makes the most widely used open-source MapReduce implementation.

A lot of effort has been made by the distributed data processing community into optimizing MapReduce for a wide range of use-cases [82]. [83] proposes an extensive survey of this optimizations and Figure 2.1 gives an idea of how much effort has been made into fitting common problems into the MapReduce programming model.

Thus, numerous programing models have been derived from MapReduce or attempt to broaden the class of applications the framework can support. *MapReduce Online* [84] makes MapReduce suitable for stream processing and online aggregation; *Twister* [85] and *Haloop* [86] support iterative MapReduce jobs; *Pig Latin* [87] is a language and a runtime system enabling high-level queries and analysis to be performed on top of Hadoop, while *YSmart* [88] and *Hive* [89] offer respectively an SQL and an SQL-like interface to MapReduce.

2.5.1.2 Other Implicitly Parallel Programming Models

Other programming models with implicit parallelism have been proposed for applications that cannot be squeezed to fit in the MapReduce paradigm [90] or that cannot reach satisfying performances [91]. *All-Pairs* [92] is one of them; this programming model offers a simple abstraction with implicit parallelism to the classical all-pairs problem. *Spark* [93] is a programming model and runtime environment for distributed memory environments that loads data sets in memory and executes tasks built from a few specific primitives (map, join, group by, filter, etc.) on them. Because Spark does not store and load data sets between tasks, it avoids costly disk access and network latencies in applications that execute multiple tasks on the same data.

Phœnix [94] is a parallel programming model with a message passing semantics that can accommodate highly dynamic environments, where nodes can join and leave the computation at any time (also called "node churn"). In Phœnix each physical node "assumes" (or is responsible for) a set of virtual nodes. A message is sent to the address of a virtual node and the runtime system takes care of delivering it to the physical node that assumes it, despite the fact that the physical node that assumes a virtual node can change at any time. The number of virtual nodes often match the number of partitions in the input data, so that each virtual node is given a piece of the input data. In that sense, Phœnix offers a very elegant way of sending messages to data instead of nodes, erasing concerns of the underlying infrastructure and allowing programmers to focus on data only.

2.5.2 Programming Models for Graph Processing

Many big data sets, in the field of social networks in particular, are structured as graphs and programming models need to account for this structure for maximum efficiency. *Pregel* [95] proposes a programming model for processing arbitrary large graphs, implicitly parallelizing the tasks on vertices. Like in most other programming model for large scale processing, fault tolerance is transparent to the programmer. To program an application with Pregel, one must implement a function that will be called repeatedly on the graph's vertices. The function on each vertex is able to change the state attached to a vertex and to all its arcs. While the function only has access to a partial state, Pregel allows messages to be sent to other vertices. The user-defined function is called in parallel on all the nodes of a cluster with a barrier between iterations and the computation stops when there is no more work to do on all the vertices. The runtime system decides how to partition the graph and no explicit parallelism or synchronization is required from the programmer. *GraphLab* [96–98] is another implicitly parallel programming model for graphs that uses shared memory instead of message passing. Here "vertex functions" are also applied to edges and unlike in Pregel they have access to neighbors of the current vertex.

2.5.3 Programming Models for Stream processing

Some applications have to deal with a constant flow of incoming data [99,100] for which traditional models are not well suited. A stream is an infinite list of bytes indexed by time; a typical stream processing system contains contains three kinds of modules: *sources* inject data asynchronously into the system, *sinks* remove data from the system; in-between the sources and the sinks, a series of *filters* or *operatiors*, often arranged in pipeline, apply different treatments to data. A typical application would setup the

following stages: first, the input data is filtered to keep only the relevant pieces; second, the filtered data is transformed into a format suitable for processing; third, data is processed iteratively or incrementally, which is where the actual computation happens; forth, the result is stored.

Many Data Stream Management Systems (*DSMS*) follow this general design [101–105] for applications in many domains including sensor networks, the Internet of things, online advertising, log analysis and social networks. The modular design adopted by most DSMSs also has the advantage of allowing parallelism at multiple levels; each module can be easily run on a dedicated machine, and still be multi-threaded.

2.5.4 Programming Models for Incremental processing

Incremental processing is a set of techniques that allow a system, when a subset of input data is updated, to run only the parts of a computation that depend on the input data that changed. Thus, incremental processing is suited for applications that need to process the same data set several times, when small pieces of the data set is modified between computations; this scenario is common in data mining applications or web crawlers that mostly index the same web corpus over and over, for example.

In this spirit, [106] proposes a *continuous bulk processing* system that offers abstractions and primitives for repeated computations on the same input data set. [107] proposes a programming model with special instructions for *recording* operations to repeat on the first run, and re-execute them incrementally on the subsequent runs. Variables that are accessed in successive executions must also be declared with a special type to let the system compute the correct data dependencies.

Google's *Percolator* [108] is a system for distributed incremental processing based on an observer/notifier paradigm. A set of Percolator workers scan a BigTable [66] for changed columns and executes trigger-like procedures called "observers" to update some previously computed results. *Presto* [109] is a distributed implementation of the R language [110] packed with incremental features that has a semantic close to Percolator. Presto uses HBase [67] as a storage backend and allows programs to attach *callbacks* to arrays of data (or partitions of an array); when the data rows are updated, the callback is executed, similar to database triggers. These callbacks are the way to express incremental programs as they naturally exhibit dependencies between data and programs.

Solutions for incremental processing sometimes involve modifying an existing system to make it incremental; this is the case for MapReduce, for which several incremental implementations have been proposed over the last few years [111, 112].

2.5.5 Programing Models for Iterative processing

Iterative processing describes computer programs that run repeatedly, producing a better approximate at each iteration. An iterative procedure terminates after a given number of iterations (usually large enough to ensure convergence) or when the approximate converges to an estimate up to an acceptable threshold.

Distributing iterative algorithms introduces difficulties due to asynchronicity that can make otherwise convergent algorithms diverge [113]. As for incremental processing, programming models and MapReduce in particular have been adapted to reduce the latency induced by loading and storing the same data set from and to HDFS repeatedly [85, 86, 114, 115] while still benefiting from high level abstractions, high throughput

and effortless fault tolerance. [115] in particular gives an interesting study of MapReduce parameters that strongly impact iterative jobs.

2.6 Workflow and Dataflow systems

Distributed workflow and dataflow systems allow to build distributed applications from sequential building blocks. They are often represented as Directed Acyclic Graphs (DAG) where vertices represent sequential tasks and edges represent the flow of control—for worflows—or the flow of data—for dataflows.

In the general case, distributed workflow systems take such a DAG as input, compute task or data dependancies, and attempt to run tasks in parallel when it is possible; otherwise, they are serialized. This representation is called *abstract workflow*. From this abstract representation, workflow and dataflow systems generate a *concrete* workflow that maps tasks to machines. Following the trend for implicit parallelism we discussed before, the translation from abstract to concrete workflows tend to make distributed execution on heterogeneous and even hybrid infrastructures as transparent to the user as possible. In addition, the DAG representation allows to perform various scheduling optimizations beforehand.

This is the case of *Swift* [116]; Swift is a scripting language where variables encapsulate files and data sets and where functions encapsulate commands. Function arguments are variables and the corresponding data sets or files are implicitly transferred to the machine where the command is to be executed; Swift computes data dependancies based on variable assignments in the language and runs function calls in a distributed fashion to a variety of infrastructures. *GEL* [117] offers features similar to Swift to the difference that parallelism must be expressed explicitly.

HTCondor combined with *DAGMan* [118] takes a concrete DAG as input, compute task dependencies and schedules the tasks to computer clusters, clouds and desktop grids. To support desktop grids, the HTCondor meta-scheduler checkpoints tasks, detects failures and restarts tasks on different machines to complete the application.

Pegasus [119] allows to generate an abstract workflow from several high-level APIs. It applies transformations and optimizations to the graph (for data locality, improving transfers, adding housekeeping tasks etc.) and uses information about the execution environment to generate a concrete workflow DAG. This concrete workflow representation can be fed to HTCondor or other workflow engines for execution.

Swift and HTCondor deal transparently with data transfers between sites, different scheduling and security policies, allowing users to focus on their application code with no concern of the actual execution environment. Scientists from many communities that are not specialists of distributed systems now rely on workflow and dataflow systems to leverage considerable computing power for their applications. Efficient execution systems are paramount to efficiently utilize their resources.

Dryad [120] is a dataflow system that requires users to program their distributed application as a DAG where vertices are encapsulated programs and where edges are communication channels. Dryad offers interesting features for very small to large clusters; while the DAG model obligates users to think about parallelism, the runtime system takes care of mapping vertices to resources, actually running processes and setting up the communication channels. These channels are hidden by a generic API that transparently implement files, FIFO or network communications based on locality. While programming algorithms is more complicated, the runtime system aims at being as simple to use as MapReduce [80], with transparent fault tolerance.

2.7 Data Provenance

Data provenance—that comes also with the name "lineage" or "pedigree"—constitutes the complete history of derivations and treatments throughout the life of data; as an essential tool to preserve the quality of scientific data assets over time, it has gained significant interest in the e-science community [121, 122]. To answer the simple question "How was produced this piece of data?", provenance information must comprise fine grain documentation about processes (when and where they ran, what arguments they were given, what was their environment), tools (their version and the version of the libraries used, their compiler, compilation options) and input data (what instrument, tools and process generated them). This necessitates specific provenance architectures able to capture, store and query a lot of information provided by scientific workflows (either workflow systems or human manipulations sometimes called "human workflow") [123].

2.7.1 Open Provenance Model

The Open Provenance Model (OPM) is the first attempt at producing a generic model for provenance. The interest for the model drove rapid evolution during the last 7 years [124–126]. The model defines ways to represent provenance and query provenance information. The model was designed to be generic enough to not only benefit the computer science community and represent the provenance of anything.

The OPM defines data items as immutable "artifacts" and operations on artifacts as "processes" [127]. In this definition, processes consume artifacts to produce new artifacts in a certain environment defined by "agents". Defining all the entities of these three groups and causality relations between them is the difficult part of provenance reconstruction.

The OPM is used as an interface for interoperability, allowing provenance information to be shared and queried between systems [128]. Several methodologies [129] and examples [130, 131] of how to implement the Open Provenance Model in existing systems are now available in the literature, making it currently the best candidate for storing and exchanging provenance in the future.

2.7.2 Provenance Systems

A first ad-hoc solution is to store provenance information as metadata in the data files. While this has the advantage of working for systems that are not *provenance enabled*—like a traditional filesystem—it can hardly be efficient because provenance information often grows larger than the data it refers to [121]. This approach has been applied to relational databases where extra provenance fields are added to database schemas [132]. This option quickly reached important limitations that motivated the need for specialized databases that would only store and query provenance information only.

This idea evolved into dedicating relational databases to provenance only, and more specialized provenance databases. While they are numerous, most, of them are part of larger systems that include specific provenance representations, acquisition techniques, specialized query languages and visualization tools.

myGrid [133], for example, is a set of web services, tools and abstractions for bioinformatics experiments on the grid. It features services for executing workflows using

Taverna [134], distributed queries and a data repository called mIR. mIR is a service central to myGrid in that it not only store data sets but also stores provenance information recorded automatically by the other services, including Taverna. Taverna records information about individual tasks and their input and output data, and stores it in a dedicated *Apache Derby* [135] relational database. Taverna can export provenance into a subset of the OPM and into the *Janus* [136] format which is part of the myGrid project and allows third party components to add annotation to provenance entities.

PASOA [137, 138] is a framework for provenance that defines "actors" (users and human workflows or a workflow system), "processes" that are executed by users and "interactions". In this model actors are responsible for recording information about processes (*Process Documentation*) in the form of assertions. Assertions can have three types; "actor state assertions" document the state of actors at the time an interaction took place; "interaction assertions" document what data was exchanged during the interaction; "relationships" documents interactions between actors and how they produced data. Low-level query interfaces are provided to construct querying applications able to answer assertions questioned by users.

Karma [139] is system for collecting and managing provenance information from various workflow systems. It stores provenance information in a relational database and can export provenance in a way that complies with the Open Provenance Model. However, its 2-level model is more detailed than the OPM. The "registry" level is an abstract representation of processes, services and data consumed and produced by them. The "execution" level represents instantiations of a registry-level model, i.e. a concrete workflow execution. Once the registry model defines how services interact, consume and produce data, Karma is able to capture provenance information with no a-priori knowledge of the execution model. In addition, Karma is not tied to a particular workflow system.

Chimera [140] proposes a *Virtual Data System* (VDS) that describes three types of entities: a *transformation* is an arbitrary executable; a *derivations* represents the execution of a transformation; a *data object* is the input or the output of a derivation. The VDS is implemented by a *Virtual Data Catalog* (VDC) that is controlled and queried by a *Virtual Data Language* (VDL). The VDL allows to describe before execution how a data object should be produced and after execution how it can been produced again. A query to the Virtual Data Catalog then serves data objects directly if present, or simply regenerates them otherwise.

2.7.3 Automated Provenance Collection

Provenance recording implies that all actors (the user, the workflow system and applications) participate in the collection of provenance information, agree on a common format and on some parameters, like granularity. Automatic provenance collection is the approach that removes users from the process and attempt to do the collection transparently, eliminating as much *human errors* (through user manipulations and developer errors and poor decisions) as possible [141].

The approach undertaken by the *Provenance Aware Storage System* (*PASS*) is to record provenance directly at the operating system level; in [142] Muniswamy-Reddy et al. propose a novel storage system that plugs on the Linux kernel to automatically and transparently collect provenance as applications and users issue commands. PASS is thus able to capture the binary path, version, command line arguments, environment variables and more of the process that produced a file. The proximity between the

provenance storage and the data storage has the additional benefit of limiting the risks of introducing inconsistencies that can occur when a provenance database is not notified of changes to one or several copies of a file.

2.7.4 Collaborative Data Science

Collaborative Data Science is a wider movement that aims at *open sourcing* and sharing scientific data sets to increase collaboration and public access to information. There are two faces to this trend; first, scientists in many fields that are willing to share the data sets and detailed processes that led to the results they publish in research articles; second, governments of more and more countries like the US [143] and France [144] follow the Open Data initiative by making freely available finance, business, climate, health and more data sets for the sake of transparency.

Collaborative Data Science and Open Data thus naturally encompass the problem of provenance, as the first action users of a publicly available data set take is to check whether it fits their purpose and meet the quality standards they need. Several platforms have emerged, often available as Software as a Service like *Sense* [145] that offers a complete integrated workbench for storing large data sets, prototyping data analysis workflows interactively and reproduce them simply. *DataHub* [146, 147] offers advanced storage features like versioning of large data sets with a large number of files and querying that accounts for data sets structure and versions.

In the scientific world, a lot of attention is invested into increasing the value of data sets through reuse and collaboration. Currently no system offers the possibility of linking scientific publications to data sets for experimental reproducibility.

2.8 Data Life Cycle Management

There have been few works around the concept of Data Life Cycle to improve the management of e-infrastructures. In [148], the authors introduce a model for Scientific Data Lifecycle Management (SDLM), which is a generic description of the different stages and processing steps for scientific data sets when handled by e-science infrastructures. In [149], authors introduce the concept of *Grid Data Life Cycle* (GDLC), and propose a system (also called Active Data) to track how the data are computed across multiple Grid systems so that data can be transparently recreated when needed during execution. FRIEDA [150] leverages the data life cycle concept to provide descriptions of application-specific storage requirements and use this information for storage planning and data deployment on Cloud infrastructures.

2.9 Discussion

This review of the literature offers a broad view of past and current works contributing to the exploitation of big data sets. Both computing infrastructures and software have been adapted and are still evolving to address growing volumes of data. Consequently, one can see a future where storing, moving, filtering and storing data sets only will become more challenging. We dub these operations *Data Life Cycle Management* (DLCM), predicting that it will be a strong limiting factor for data-intensive science in the short term. This section explains the relation and impact of infrastructures and software on Data Life Cycle.

2.9.1 Difficulties Coming from Applications

Data storage and management systems elaborate strategies to provide end users with abstractions that fade infrastructure details away. Typically, users see a flat logical view of data sets when they are in fact highly distributed and replicated in remote data centers. Different software have different logic for providing these abstractions, and orchestrating transfers between them can require fragile ad-hoc scripts. On top of this, users do not necessarily have a choice over which software to use; they may have to incorporate software used by a collaborating research group; they might also depend on legacy code that cannot be changed at a reasonable cost. As the number of systems increases in an application, it gets harder for humans to comprehend the whole system and supervise data life cycle management. This is more true for data intensive applications because of the additional difficulty of adapting data sets to the specific requirements of individual systems that were not designed to cooperate. Performing simple manual tasks in such environments can become incredibly difficult for users because the actual layout, location, distribution and even the actual bytes of their data are most of the time miles away from the view they get, and from what they can see.

These applications can currently be modeled in several ways, and making this model is often the starting point of a workflow or dataflow project: applications are described as graphs where vertices represent tasks and edges represent control or data passing from a task to the next. This representation that focuses on tasks and in which data is simply bound to edges on a graph is too coarse and does not contain enough information for a variety of use-cases. This is the case for incremental applications and applications featuring non trivial data movements between tasks, for example when replication is used. Users wanting to optimize the coordination between tasks in such applications need a finer level of details about data operations that is not offered by traditional models.

2.9.2 Difficulties Coming from the Infrastructures

Adding to the difficulty coming from applications, hybrid infrastructures create their own share of difficulty in data life cycle management. When managing large data sets on hybrid infrastructures, one has to deal with multiple things that depend on node, storage and network properties; the heterogeneity of these resources implies that performance and fault models can all differ from one node to the other. So again, implementing simple data life cycle operations like transfers or copies require a lot of code that is specific to particular situations. Data distribution and replication, storage and transfer of intermediate results, unexpected errors and events are amongst the things that are naturally part of the data life cycle, but require more attention on hybrid infrastructures. For example, accessing two copies of the same file on two infrastructures can be very different regarding software and interfaces, security policies and latency, despite all nodes appearing identical.

2.9.3 Error Detection and Recovery

When many nodes, disks and processes are involved in a computation, the number of events happening at the same time can be huge; in the mass of things that go right, the small things that go wrong can be hard to catch. Because workflow and dataflow systems orchestrate task executions, errors are only detected when a task fail. That is, a process can leave data in an incorrect state completely undetected as long as it does

not fail completely, i.e. as long as it returns 0. DLCM operations are subject to these small silent failures: for example, only a fraction of faulty file transfers can be enough to corrupt a data set and invalidate a whole experiment. Unlike workflow and dataflow DAG models that assemble tasks, a finer representation of applications that focus on data at a finer grain could specify the expected state of data at key points. A system implementing such a fine grain specification could detect errors earlier and save time and resources.

Once errors are detected, the difficulty of exploring several infrastructures and system makes recovery challenging or nearly impossible, forcing the user to step-in and fix the problem manually. As a general rule, there is also a lot of "human workflow" in place for managing the data life cycle: the user launches certain processes manually at key points because they are difficult to automate. These actions, e.g. launching the workflow system when input data become available, cannot be automated by a workflow system. These manual data life cycle operations should be avoided as much as possible; not only are they suboptimal in the way they tend to introduce a delay between the moment when an action *could* be performed and the moment it *is* performed, they are also prone to errors.

2.9.4 Optimal Usage of Hybrid Infrastructures

Data intensive applications cannot achieve top performance and resource usage on hybrid infrastructures without an important flow of information between both. Currently, data life cycle management decisions, like how to distribute data and workloads amongst the infrastructures, are made by users; in the same spirit, conflicts and errors are solved manually by users. However, the reason for this is not because humans are better at it, but because these tasks are nearly impossible to automate. To program the data life cycle operations humans usually do, a computer system should have the same high-level and exhaustive view of the whole application. Such a model does not exist at this time; it would need to represent fine grain data operations, with links to systems and infrastructures clearly shown. Only then could programs make DLCM decisions and take actions that used to be humans'.

2.10 Objectives for this Thesis

From this study of the literature, I now define the data sets and the infrastructures that will be considered in this thesis. Then, I present the objectives for this thesis.

2.10.1 Data Sets and Infrastructures Considered in this Thesis

Characterization of Data Sets In this thesis, we consider data sets that originate from science and the industry with a particular focus on large and collaborative scientific data sets. From the science community, we consider data sets acquired by scientific instruments, simulation, sensor networks or the result scientific experiments in all domains, e.g. physics, biology, computer science or the humanities; from the industry, we consider data sets coming from social networks and ubiquitous devices, continuous streams of customer-produced data, online advertising and content delivery networks.

Considering that data sets must not necessarily be very large to be challenging to manage, we do not define a lower bound on their size. However, we consider data sets that are dynamic. They may grow or shrink in size continuously or occasionally, they can be transformed, derived, partially modified or unexpectedly altered. They are either too big to fit on a single storage node or need to be distributed for availability (serving queries in a decent time) or computation (dividing analysis between several machines). Finally, we consider data sets that are shared: even long after they have been exploited, others might still need a data set and the associated metadata. Several research groups may also be working with copies of the same data at the same time and want to share results with the world.

Characterization of Distributed Infrastructures For managing the type of data sets we have defined, we consider infrastructures ranging from a network of low-powered and low-CPU sensor devices to large high-performance platforms. Our infrastructures may be a a cluster, a grid, a cloud, a desktop grid, or an hybrid composition of them.

As such, the solutions we propose must be able to work with any of these infrastructures and with all at the same time. We describe here some of the properties of this hybrid infrastructure:

Heterogeneity in hardware (nodes, network, storage), software (batch scheduler, operating systems, available storage systems), performance (CPU, disk and network latency), fault models (stable grid resources, volatile cloud and volunteer resources) and administration policies (security, access to resources);

Geographic Distribution of computing sites over arbitrarily long distances, obligating data management systems to cope with different latencies, making synchronicity challenging if not impossible.

To accommodate this model of infrastructure, we level down most of their properties, considering an infrastructure with a high latency network and nodes that can leave and join at any time. Nodes are expected to have low computational power, like sensor nodes or mobile devices. Nodes can be behind a firewall or a NAT, complicating peer-to-peer communications.

2.10.2 Objectives for this Thesis

We express the goal of providing a novel data life cycle management system that will overcome the difficulties presented above, and make distributed data life cycle management easier and more reliable. To this end, we define several sub-objectives for this thesis:

Formalization We need a model to formalize the life cycle of distributed data in data-intensive applications. The model must allow to represent fine grain interactions between processes and data sets, answering the question “what is the effect of a process on data?”.

Automation and programing Our DLCM must automate as much things as possible, from launching processes to recovering from errors. It must offer a simple and powerful way to express DLCM operations without infrastructure concerns.

Coordination Our DLCM system must offer users and applications a way to react to data events to optimize their coordination, even when events occur outside their own scope.

Support of legacy software Because it is not always possible to change existing code, our DLCM system must integrate legacy software just like new dedicated software, without requiring code modifications.

Collaboration Our DLCM system must help researchers keeping track of the origin and the destination of their data when they are shared with collaborators. Sharing data sets must be automated to guarantee the quality of data and their immediate availability.

2.11 Conclusions

In this chapter we have presented the state of the art in infrastructures and software that support data intensive distributed applications. We have identified *Data Life Cycle Management* as a key issue for optimal support of large distributed data sets on hybrid infrastructures.

This thesis proposes a novel system for data life cycle management that has two angles; the first angle is formal, and aims at providing a model for, representing and specifying DLCM; the second angle is practical, and develops a programing model for automating data management tasks, increasing collaboration between heterogeneous systems. The resulting system, called *Active Data*, is thoroughly described in the next chapter.

3 Active Data

Active Data is the main contribution of this thesis; it offers a rigorous model for specifying and programming distributed data management on heterogeneous infrastructures. In this chapter, we first analyze essential aspects of distributed data-intensive applications and determine what features and elements need to be modeled in order to capture essential aspects of distributed data. Based on this analysis, we propose and thoroughly describe a meta model for representing the life cycle of distributed data. We present a prototype implementation for the meta model and the Active Data programming model and runtime environment.

3.1 Representing the Life Cycle of Distributed Data

3.1.1 Generalities on Distributed Data Life Cycles

In this subsection we analyze common patterns observed in data management systems. From our observation, we extract elements that a meta model for data life cycles must support.

3.1.1.1 Data and Life Cycle: Definitions

Before defining and characterizing data life cycles, we need to define what "data" means to us. In this work, we call *data item* an atomic piece of information stored in a computer system. Here "atomic" is relative to said system: on a disk this usually corresponds to a block, when on a file system, it is more likely to be a file. According to this definition, a data item can be, including but not limited to, a file, a storage block, a tuple in a database or an in memory record. A data item does not contain metadata, which we consider to be stored aside. As such, a data item may be empty —in which case we say it has no *payload*— and still have metadata describing it.

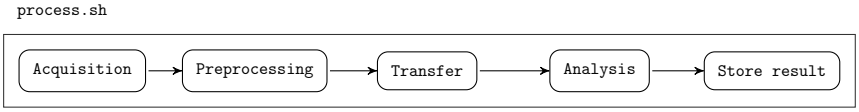


Figure 3.1: Sequential data life cycle represented as a chain of tasks coordinated by a shell script.

Between the creation of a data item and its deletion, many operations that change its payload are applied. For example, consider the process on Figure 3.1 where several successive treatments are applied to a data item; we call *tasks* these operations represented by boxes; implicitly, tasks create *data stages* (see Definition 1).

Definition 1 *A data stage is a step in which a data item pass between its creation and its deletion. A data item is pushed from one stage to the next by a tasks, i.e. an operation that alters it.*

However, each task can push a data item more than one stage further; this is illustrated by Figure 3.2 that features three different stages ("Extract metadata", "Filter" and "Annotate") for the "Preprocessing" task. This means that looking at software and service interfaces only may often not be enough to capture important data stages at a satisfying fine grain.

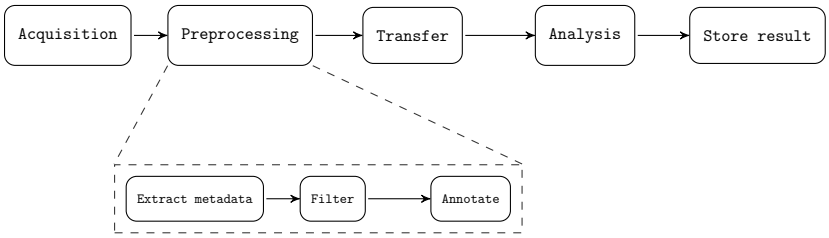


Figure 3.2: The relation between task and state is not *one-to-one*: a single task usually encapsulate several meaningful operations in regard to the data life cycle.

Definition 2 gives a first definition of data life cycle that relates to tasks, as data items are pushed from one stage to the next by tasks and stages. Examples of tasks include acquisition, replication, transfer, deletion, archiving etc. It also defines the life cycle of a data item as its *history*: what the life cycle contains is everything that happened to the data item so far.

Definition 2 *The life cycle of a data item is the succession of operational stages that a data item has been through since its creation.*

We call *Data Management Systems* computer systems that participate in the life cycle of data. Data management systems store, transfer, index and query data, they perform necessary maintenance operations prior to processing and execute processing tasks. A large distributed application involves several data management systems, each performing one or several tasks. For example, a system stores and index data, and another system performs the analysis.

Additionally, data management systems often organize data items into larger sets called "data sets" that allow users and programs to apply batch processing to many data items with a single request, hence making operations simpler for users.

Definition 3 *A data set is a collection of data items in a system that share some property, like their origin or function. Data sets provide a logical view of data items that can be physically distant.*

A data set is said to be distributed *whenever or its storage or processing requires it to be split in smaller units that are handled by different machines.*

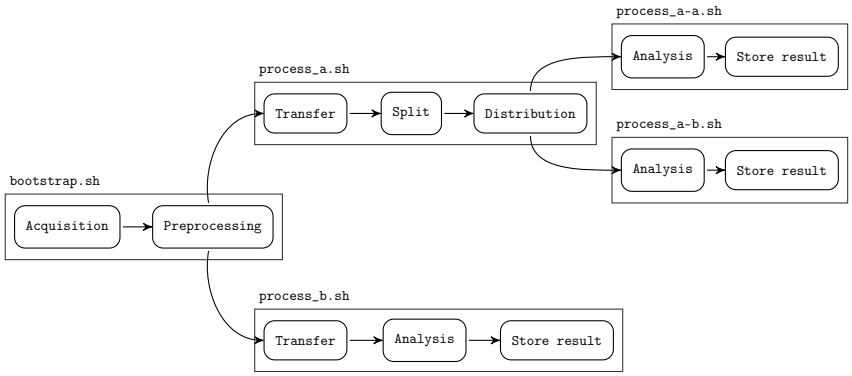


Figure 3.3: Parallel life cycle represented as a tree of tasks. Each local process is coordinated by a specific shell script (represented by a frame) responsible for local task execution and spawning remote processes.

Data management tasks are typically executed sequentially, by different programs, under the supervision of a runtime system; this system can range from a simple ad-hoc script to a more complex workflow system. Figure 3.1 gave a first example of a simple workflow composed of tasks ("acquisition", "preprocessing" etc.) represented by boxes, that can be orchestrated by a single script. In the case of distributed data, some of the tasks happen in parallel, and thus the succession of tasks *(workflow)* is not a chain anymore. This last point is illustrated by Figure 3.3. The figure has two branches ("Preprocessing" and "Distribution"); the two scripts after each branch are executed in parallel.

3.1.1.2 Data state

We just stated that users use scripts or workflow-like systems to orchestrate the various tasks their experiments comprise. In terms of life cycle, we stated that a task may imply several data stages. So each task pushes data items one or several stages further and, at any time, the current stage is what we call the *state* of the data item.

Following on this definition, we distinguish two kinds of states in data life cycle; a first set of states comes from the tasks: for example, on Figure 3.1, if task "Transfer" has returned, but not "Analysis", the state of the data item is implicitly "Being analyzed". A second set of states comes from *inside* the tasks: the current state of a data item being treated by a task depends on what internal stages it has passed yet.

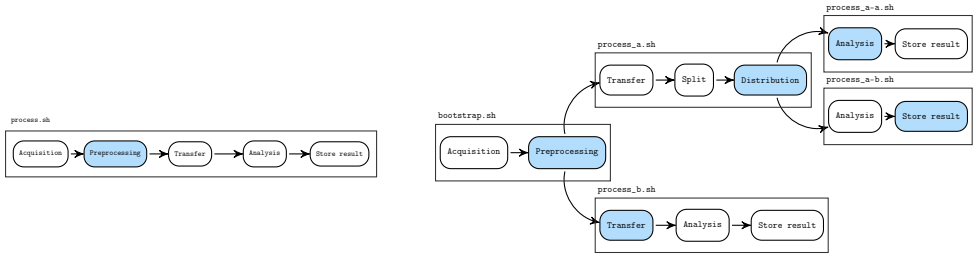
Furthermore, we distinguish two kinds of states:

Explicit states: the state is explicitly coded (e.g. stored along metadata) for every data item in the system, whether exposed to users or not. The set of possible data states can be extracted directly from the code or documentation;

Implicit states: the state is not explicitly coded, but the sequence of operations applied to data items can be extracted from the code or documentation and translated into states.

3.1.1.3 Distributed Data States

The observations above fit a sequential life cycle as described in Subsection 3.1.1.1; however, the definition of state is not sufficient to describe the state of data when



(a) On a single thread, the state implies that the data item went through all the previous tasks.

(b) Each branch represents a different thread. The state of the distributed data item is the conjunction of all the individual states.

Figure 3.4: Illustration of the state of a data item deduced from the current processing task. The state of a distributed life cycle, compared to a sequential one, requires a slightly more complex definition.

distributed. Since several replicas are present simultaneously in different systems, each replica in its own system has its own state. Figure 3.4 represents visually the state of a data item in two different workflows. On Figure 3.4a, the data item has a single copy; it passed "Acquisition" and its state is marked in blue: it is being preprocessed. On Figure 3.4b, 5 replicas of the same data item exist at the same time. The state of each replica is marked in blue as well. Thus, the question we need to answer is "What is the *complete* state of a distributed data item?". We answer this question by asserting that the state of distributed a distributed data item is also distributed (Definition 4).

Definition 4 *The state of a data item distributed in several systems is the set of all the individual local states of the replicas.*

3.1.1.4 Data Identifiers

Earlier we defined what "data" means in this work. Data management systems, storage systems and analysis tools need a way to identify data items—or *discriminate* them. Systems satisfy this need by assigning a unique identifier to each item they manage. Later, when the system needs to apply some treatment to a specific data item, it use its identifier to reference the targeted item. Data management systems use a variety of conventions for data identifiers; for example iRODS [68] assigns a property named `R_DATA_ID` of type `Long` to each file; HDFS [45] refers to files using URIs; database records are retrieved with a reference to the database, a reference to the table and the identifier of targeted tuple (typically an auto-incremented integer).

Because of this variety, Definition 5 does not restrict a type for data identifiers and refers to *words* instead. Later, this identifier is often abbreviated as "UID".

Definition 5 *The* data identifier *is the word that uniquely identifies a data item in a given system.*

3.1.1.5 Data Replication

In addition to simply performing tasks concurrently, data management systems use replication for various reasons including load balancing, fault tolerance and performance optimization. For example, HDFS stores by default three copies of each file; two

copies are placed in the same rack, and the third in a different rack. In addition to discriminating different data items, systems that use replication must discriminate the several replicas of the same data item.

These systems typically hold two identifiers for the data they manage: one is what we previously defined as *data identifier* and is the same for all replicas; we call the other identifier *replica identifier* and it is different for all replicas of the same data.

It is also observed that the data identifier is commonly exposed to the user; it is generally the reference they need to address their data. However the replica identifier is most often hidden by the system because users do not need it: when the system receives a request to perform an operation on data, it decides what replica is used with no user intervention. The level of replication (the number of replica for a data item) is also often hidden from the user and may vary under various circumstances. Definition 6 defines the replica identifier that we sometimes refer to as "RID".

Definition 6 *The* replica identifier *is the word that discriminates a replica of a single data item from all the other replicas of the same data item.*

Also note that in most systems, data identifier must not be confused with the notion of data name. When the identifier is by definition immutable, the name—that is often used by users to refer to a particular data item or file—can change. For this reason, systems often hide data identifiers and map them to names that are exposed. This notion is bound to "logical address" vs. "physical address". In many systems using replication, the true location of an item accessed by users is hidden behind a logical—or virtual—address. When a logical address is used, the system redirects it transparently to one of many physical addresses without exposing it.

3.1.1.6 State Transition

Now that we defined data states, let us focus on how data management systems change the state of a data item over time.

Consider the sequence of operations applied to a data item on a single thread; since states are deduced from tasks, the change from one state to the next is determined by the thread's control flow—arcs in our current representation. In other words, when a task completes for a data item, the state of the item is changed to reflect the situation. For example, when a task named "Filter" returns, the state of the corresponding data item goes from "Unfiltered" to "Filtered".

What task can be executed after a completed task—and therefor what state can follow an assigned state—can be determined beforehand by the program tasks and possible paths; we use the generic expression "state transition" (or simply "transition") to designate a possible change of state in the course of a life cycle.

So far, on our visual representations, states are deduced from tasks, and changes of state are made by tasks. Hence each box seems to fit both purposes and bring confusion. For this reason, we decide that a life cycle model must make a clear distinction between states and state transitions; as such, we give a first definition of transition.

Definition 7 *A state transition is an operation on a data item that changes its state, either distributed or not.*

Definition 7 defines state transitions and clearly separates states and transitions in a data-centric fashion: a operation that does not update the state of a data item is not

a transition, and needs not be represented on the data life cycle. A task is composed of one or several transitions, one for each operation that updates the state of a data item.

In this subsection we have introduced three main components of distributed data management systems: data items and their assigned unique identifier, data states and data transitions.

The traditional view centered on tasks is not sufficient to express the complexity of data life cycles. A model for data states and transitions must show that at any time, a data item can have several states, and that several transition may be occurring. The next subsection introduce the result of our reflexion, that is a meta-model suitable for distributed data life cycles.

3.1.2 Life Cycle Meta-Model

Informally, the life cycle model of a data item is the set of all the states it can have at any time, and the set of transitions that dictate what changes of state are possible.

We find in Petri Networks [151] a suitable starting point for our meta-model as they clearly present states and transition. They naturally expose [152] distributed states and expose concurrency graphically, enabling to simply describe complex systems. As such, Petri Networks have been used for describing and analyzing parallel and distributed applications for a long time [153, 154].

In this section we present how Petri Networks are used and extended to meet our data centric requirements.

3.1.2.1 Petri Networks

A Petri Net is classically a 5-tuple $PN = (P, T, F, W, M_0)$ where:

- $P = \{p_1, p_2, \ldots, p_m\}$ is a finite set of places represented by circles;
- $T = \{t_1, t_2, \ldots, t_n\}$ is a finite set of transitions represented by rectangles;
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of oriented arcs between places and transitions and between transitions and places;
- Places in a Petri Net may contain tokens represented by $\bullet$;
- $W : F \to \mathbb{N}^+$ is a weight function which indicates how many tokens every transition consumes and how many tokens it produces;
- $M_0 : P \to \mathbb{N}$ is a function that indicates the initial marking of places.

A transition $t \in T$ is *enabled* if and only if for any place p as $\{p \in P \mid (p, t) \in F\}$, the number of tokens in p is greater or equal to $w(p, t)$, the weight of the arc between p and t. If the weight is 1, w is generally omitted in the visual representation.

In addition, our definition allows an extension of Petri Networks: inhibitor arcs. When a transition is connected to a place by an inhibitor arc, the transition is disabled whenever the place contains at least one token.

In our meta-model, a Petri Network represents the life cycle model for data in a single system. Petri Net places represent all the possible states of data items in the system, and Petri Net transitions all the operations that initiate a change of state. Tokens represent replication: a token is a single replica, and the place it is on represents

the current state of the replica. The *marking* or *configuration* of a Petri Network is the way tokens are distributed over the places at a given time. As such, and according to Definition 4, the state of a data item matches the marking on its Petri Network.

3.1.2.2 Definitions

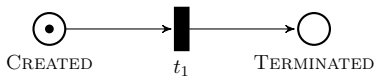


Figure 3.5: Graphical representation of a minimal life cycle model, composed of two places (CREATED and TERMINATED) and a single transition t_1.

We now formally define our life cycle meta-model by extending Petri Networks.

Definition 8 *A data life cycle model is a 6-tuple* $LC = (P, T \cup T', F \cup F', G, W, M_0)$ *which represent respectively a set of places, transitions, arcs, inhibitor arcs, a weight function and an initial marking.*

P, T and F represent the data life cycle as exposed by the data management system. P contains at least the two following places: CREATED $\in P$ is the start place, representing the creation state of the data item; TERMINATED $\in P$ is the end place, representing the state of the data after it has been permanently deleted. In addition, TERMINATED is a sink:

$$\forall t \in T, \nexists (p,t) \in (P \times T) \mid p = \text{TERMINATED} \tag{3.1}$$

When a life cycle starts, the corresponding data item has only one replica on the CREATED place. So the function M_0 is defined as:

$$M_0(p) = \begin{cases} 1 & if\ p = \text{CREATED} \\ 0 & if\ p \in P \setminus \{\text{CREATED}\} \end{cases} \tag{3.2}$$

T', F' and G are not part of the actual data life cycle as exposed by the data management system. Instead they are sets of transitions and arcs added to the model to ensure properties that we discuss later in this section. We decide to neither represent them graphically nor to expose them to users.

As a minimal example, Figure 3.5 presents the life cycle model of a newly created data item. The token present on the place CREATED means that no operation has yet been performed on the corresponding data item. The single possible action is represented by transition t_1 and moves the token to the TERMINATED place.

3.1.2.3 Data Identification and Replication

Packed with elements from Petri Networks, our life cycle model is now able to represent data states, transitions, and replication. However, we need a way to discriminate tokens and assign them the two identifiers discussed in subsections 3.1.1.4 and 3.1.1.5: the data identifier and the replica identifier.

Let δ be a data item; $\delta(id, i, p)$ identifies a replica —or token— of δ, where id is the data identifier, i is the replica identifier and $p \in P$ is a place. This implies that several data replicas may be in different states a any given time, as required.

We guarantee the consistency of the meta-model by maintaining the following properties, for $\delta(id, i, p)$ and $\delta'(id', i', p')$ two replicas of δ:

Property 1 $id = id'$ *iff* $\delta = \delta'$

Property 2 $\forall\ id \nexists\ \delta(id, i, p), \delta'(id, i', p') \mid i = i'$

Equation 3.2 specifies that each instance of a life cycle model has a single token, i.e. a data life cycle starts with a single replica. To create additional data replicas, we use a transition that produces more tokens than it consumes, such as the self-loop presented in Figure 3.6. When t_1 is fired, it consumes a single token from place p_1; because the weight of the outgoing arc from t_1 is 2, two tokens are produced on place p_1. Transition t_1 represents the operation that attributes an identifier to the new token, based on the identifier of the old token, applying the following rule that ensures Property 1 and Property 2:

$$\delta'(id, i + 1, p)$$

In the course of a life cycle, replicas can be deleted without ending the whole life cycle. We represent the deletion of a replicas by removing the corresponding token from the Petri Network with a transition that produces less tokens than it consumes. On Figure 3.6, transition t_2 consumes one token and produces none (it has no outgoing arc). Thus firing it effectively removes a token from p_1.

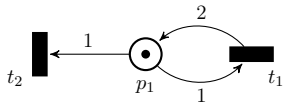


Figure 3.6: Creation and deletion of data replicas. t_1 creates a new replicas and t_2 deletes one.

To maintain an strong link between a model and the reality it represents, we decide that the data identifier *id* attached to each token will always reflect the real-life identifier of the data item they represent.

3.1.2.4 Data Life Cycle Termination

We described how to delete data replicas in the previous subsection. However, we also need to represent the termination of a data life cycle, i.e. when the corresponding data item permanently leaves the system. This situation is difficult to deal with in essence because of lingering references to deleted data that can persist in large distributed systems.

Strongly formalizing termination allows automatic constructions —like life cycle composition, see Subsection 3.1.2.5— and representing the complete deletion of a data item distributed on several systems.

On one system, the end of a life cycle is represented in the life cycle model with a token on the TERMINATED place (see Figure 3.5). After that, no operation can be performed on the data item, and as such no token can move in the corresponding model anymore.

G, a set of inhibitor arcs, prevents all the transitions to be fired with any token. Every transition in T is connected to the TERMINATED place with an inhibitor arc in G:

$$\forall t \in T,\ \exists(\text{TERMINATED}, t) \in G \tag{3.3}$$

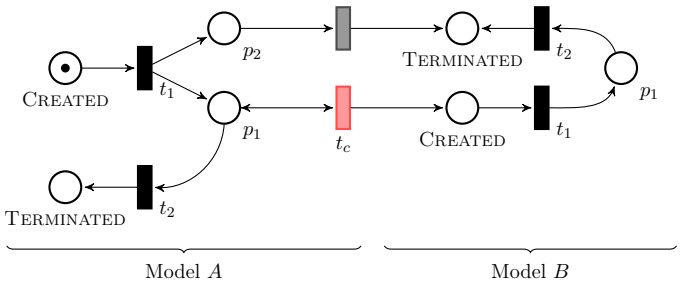


Figure 3.7: Example of life cycle model composition: a token in model A can be consumed by t_c on start place p_1 in order to start a life cycle in B; an other token can be consumed on place p_2 by t_t to stop the life cycle in B. Both life cycle models are valid on their own.

At this point, our meta-model has every element to represent the end-to-end life cycle of any data item in any system. The last missing feature is the ability to combine the resulting models as one, in order to represent data that traverse several systems throughout their life cycle.

3.1.2.5 Life Cycle Composition

Until now, our meta-model only allows to represent the life cycle of data in a single system. However, as discussed at the beginning of this chapter, most distributed applications involve several systems. In order to represent the complete life cycle of data, from end to end, we must be able to represent how data travels from a system to the other, and how data can be present in several systems at the same time.

When constructing a single large model including the places and transitions of all the systems used in a data management workflow would have worked, we decide to provide a way of connecting several life cycle models into one. This provides more reusability as system developers can distribute the life cycle model of their system and let end users assemble them as they need.

Let us consider two different systems that are unrelated, at the exception that they are used by the same user application. Informally, we represent a data item passing from the origin system to the destination system as the origin Petri Network creating a token for the destination Petri Network.

Formal definition We give a formal definition for the composition of two life cycle models that ensures that the composition of a life cycle model remains a life cycle model. Figure 3.7 features a first example of composition.

Definition 9 *We define that the data life cycle model* $A = (P_A, T_A, F_A, G_A, W_A, M_{A0})$*, composed with the data life cycle model* $B = (P_B, T_B, F_B, G_B, W_B, M_{B0})$ *is the data life cycle model* $L_{A,B} = (P, T, F, G, W, M_0)$ *where:*

- $P = P_A \cup P_B$
- $T = T_A \cup T_B \cup T_c(A, B) \cup T_t(A, B)$
- $F = F_A \cup F_B \cup F(A, B)$
- $G = G_A \cup G_B$

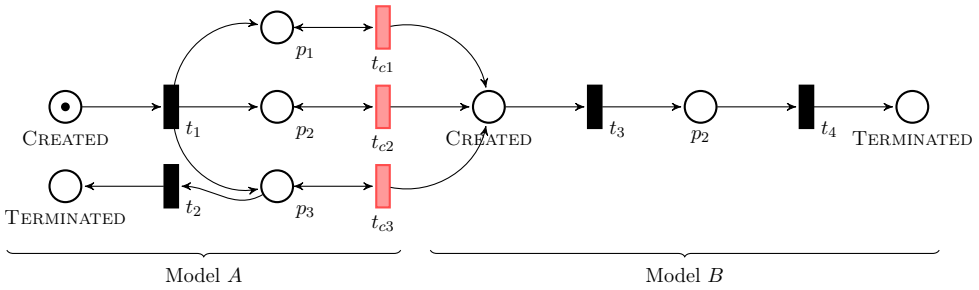


Figure 3.8: Example of life cycle model composition: any of the three composition transitions (in red) can create a new life cycle in B from a life cycle in A. The choice of which composition transition is used to create the new life cycle can be meaningful in applications.

$T_c(A, B)$ is the set of composition transitions *that link places in A to the* CREATED *place of B. $T_t(A, B)$ is the set of* termination transitions *that link places in A to the* TERMINATED *place of B.*

We note $Start(A, B) \in P$ the set of start places *from A to B, the set of places in A that are the input of a composition transition to B. Conversely, $Stop(A, B) \in P$ is the set of* stop places *that are input of a termination transition to B.*

$F(A, B)$ is the set of arcs that connect composition transitions and termination transitions to places of A and B. A composition transition is connected to its source place by a double-ended arc, acting as two arcs pointing in opposite directions.

Definition 9 allows several places in $Start(A, B)$ and $Stop(A, B)$, so system developers can represent several *entry points* from system A to system B, as illustrated by Figure 3.8. In addition, the use of double-ended arcs allows tokens consumed by a composition transition to remain on the source place; this aspect reflects the fact that when a data item is inserted into a new system, is does not necessarily disappear from the source system.

Definition 10 *We say that a life cycle model A is* composed *with another life cycle model B iff* $T_c(A, B) \neq \emptyset \wedge \forall t \in T_c(A, B)\ \exists\ \{(p, t), (t, Created_B)\} \in F(A, B)^2 \mid p \in P_A$.

Definition 10 formalizes vocabulary commonly used later in this dissertation and in other works. It states that the expression "*A is composed with B*" means that there is at least a composition transition between A and B.

Composition and Token Identification According to definition given in subsection 3.1.2.3, the token —that we note $\delta(id, i, p)$— created in the destination model is constructed from a token consumed in the source model. The data identifier assigned to the new token is the same as the existing token and is identical to the real-life data identifier; the replica identifier is incremented by one relative to the existing token; the new token's place is the CREATED place of the destination model. To satisfy the two constraints on data identifiers, the first member of the triplet $\delta(id, replica\ id, p)$ is actually a set of identifiers. Figure 3.9 illustrates how the set of identifiers is maintained in both the already existing and the newly created token when a composition takes place.

This way of identifying tokens also offers a strong link between copies of the same data distributed in non-cooperative systems. A large part of the life cycle of data a

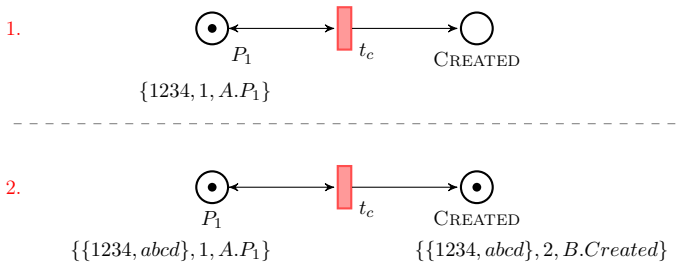


Figure 3.9: Composition process: the figure presents the state of a portion of life cycle model around a composition transition. Before the composition transition is triggered (1.), a token is present on the transition's source place (P_1); after the transition is triggered (2.), a new token is present on the destination place of the composition transition and shares the same data identifier as the old token. The identifier is a set containing the real-life identifiers of the data items they represent.

system manipulates happens outside its scope. Thus, visualizing the whole life cycle is nearly impossible without a reconciliation system like this one.

3.1.2.6 Token Tags and Typed Transitions

The last subsection described how we link replicas of the same data item distributed in different non-cooperative system; life cycle composition now allows tokens from a model to be injected in another model. We can leverage this mechanism to pass more information from a model to the other by attaching information to tokens, in a similar way as Colored Petri Networks.

In this subsection, we describe token tags and typed transitions, and we discuss the advantages of this approach with several angles: loading more information to the life cycle models, representing data collections and passing information from a system to the other.

Token tags Each token in a life cycle model can be attached zero, one or more tags. A tag is similar to a Petri Network color with two exceptions:

- places are not restricted regarding what tags they can contain;
- the set of all possible tags is not finite.

Definition 11 *We note $tags(\delta)$ the set of tags attached to token δ. At creation time of data (δ), $tags(\delta) = \emptyset$.*

Definition 11 states that tokens start their life cycle with no tag at all. Tags are attached and removed from tokens by transitions. As such we extend the definition of tokens, and refine the definition of transitions given in Definition 7.

Definition 12 *A token is a quadruplet $\delta(id, i, p, U)$ where id is the identifier of the data represented by the token, i is the replica identifier, p is the current place and U is the set of tags attached to the token.*

Token tags can be used to represent a number of things including the membership to a data set, a data type, the passing of some tests etc.

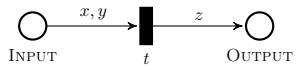


Figure 3.10: Free variables surrounding a typed transition: the cardinality of the input set and of the output set determine respectively the arity of the input arc and of the output arc.

Typed transitions Life cycle transitions, as defined earlier can consume any token present on their input place. This implies that all tokens represent identical data, which can sometimes be wrong. For example, the same data item can be present in a system in two different formats; in this case some operation may require their input data to be in a certain format, and the corresponding transition would consume only certain tokens. With token tags, Life Cycle models can represent the fact that several data types are in use in a single system. We now introduce typed transitions that can be restricted to consume only certain types of tokens. The type of a token is defined by the tags it holds (see Definition 13).

Definition 13 *We call $\theta(A)$ the set of all possible token tags in life cycle model A. Keeping in mind that a token can hold zero, one or more tags, the set of all possible token types for A is then $P(\theta(A)) \cup \emptyset$ where P denotes the power set.*

In addition to restricting the type of their input tokens, we must also specify the type of output tokens. This allows to express, in the model, how typed transitions add and remove tags from tokens.

Transitions are typed with predicates similar to guards in Colored Petri Networks [155].

Definition 14 *A typed transition has its incoming and outgoing arcs annotated with a list of free variables —one for each input token. Each free variable takes value in the set of all possible token types, as defined in Definition 13.*

In addition, a typed transition is attached a list of predicates, each being true or false depending on the free variables's value. We note $Pred(x)$ the predicate for the free variable x. Let t be a typed transition, and let $I_v(t)$ be the set of free variables on the input arcs of t. We say that typed transition t is enabled iff $\forall v \in I_v(t), Pred(v)$ *is true.*

As illustrated on Figure 3.10, in the case of typed transitions, the number of free variables can substitute for the arc arity. On the figure, the cardinality of the set $\{x, y\}$ is 2 and we do not include the arc arity in the Petri Network.

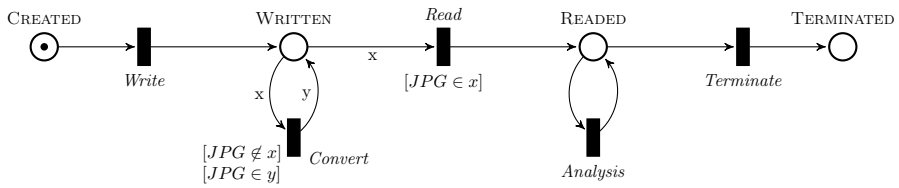


Figure 3.11: Example life cycle model featuring two typed transitions: *Convert* and *Read*. Expressions next to the transitions determine what tags input tokens must have to be consumed, and what tags are attached to output tokens.

Figure 3.11 features an example life cycle model, complete with two typed transitions; the model represents the life cycle of a data item that starts empty —as usual. Transition *Write* adds a data payload to the item; this payload is an image and its format may vary depending of circumstances we ignore. However, the program that performs the write operation represented by the transition signals what type of graphics it produced by placing a tag on the token. Further processing requires a JPG image as input, as indicated by the free variable on the input arc of *Read* and its predicate. In case the token does not have the "JPG" tag, the transition *Convert* is triggered, taking a token without the "JPG" tag as input and producing a "JPG" token as output.

We note here that instead of listing all the elements of the free variables, predicates only state the inclusion of certain tags of interest in the free variables. This allows to manipulate a hierarchy of types —which root is the empty set— and allows a transition to be triggered for a type tree by only specifying its root. Formally, let us consider two sets of token tags A and B (that are also two token types). We say that A is a subtype of B *iff* $B \subset A$.

Tags and composition When using life cycle composition, the token created on the destination model is free of any tag. However, as we discussed earlier, token tags are an easy way of sharing metadata between systems. Composition transitions can be typed to indicate that some or all tags from the input tokens are placed on the output token, effectively passing type information from one system to another.

Applications The expressive power of token tags and typed transitions allows users to pack more information in their life cycle meta-models. Here we argue about four important uses for them.

- Tags allow to insert more state information in tokens keeping the number of places as small as possible in the model. State information may include things like data format, provenance information, confidentiality level, quality checks and so forth;
- Typed transitions allow to represent operations on data as typed functions. As all tokens in a same life cycle model represent copies of the same data item, typed transitions document the format in which software components expect data items to be in order to use them;
- The same tag attached to different data items (either sharing the same life cycle model or not) allows to represent collections of related data. In conjunction, typed transitions can represent operations on collections of data.
- Combined with composition, tags allow information to travel from a system to the other, facilitating data integration.

In the remaining of this chapter, we describe the prototype implementation of the meta model and how we connect it to existing applications. We believe that providing a complete runtime system will enable users to come up with more creative use cases.

3.2 The Active Data Programming Model

The meta model described at the beginning of this chapter enables system designers to reason about the data life cycles of their products; it also helps users of these systems to integrate them better and observe complex data workflows at a human-friendly scale.

From the meta model, we now develop *Active Data*, a programming model that facilitates the development, the deployment and the maintenance of data life cycle management systems. In this section, we describe the Active Data programming model and illustrate concepts with pseudocode close to Java, the language in which the first implementation is available. However, all examples can apply to future implementations in various languages.

3.2.1 Principles

Active Data is a programming model and a runtime environment that can be defined as "transition driven". It allows to program applications by providing, for each life cycle transition, the code to be executed when it is triggered. The code is executed on the client side, in a way that does not alter the system's performance. The model is data centric and the user code receives information about the data life cycle only, and not about the infrastructure, or the actual compute node that performed the transition. For this programming model to work, data management systems and scientific applications must report what they do with data and to data: this is called *publishing* life cycle transitions. Data management systems and users are all *clients* of Active Data.

Here are the 3 main steps to get Active Data to work:

1. Make a computer representation of the life cycle model of the target data;
2. Connect the data management application to Active Data and make it report when it performs an operation that corresponds to a life cycle transition;
3. Provide code to be automatically executed in reaction of life cycle transitions being published.

The first two steps are the responsibility of the system developer and can be shipped with software releases. The user, on their side, only has to provide the code they want to execute in reaction to life cycle transitions.

At the center of Active Data is the *Active Data Service* that stores the state of all live life cycles in a given application, updates them when transitions are reported by programs, and notifies clients.

All actors benefit from this model; from the system developer perspective, providing the life cycle model along their software releases represents a very small effort and a big added value to their users. From the user perspective, the life cycle model is valuable documentation, and the programming effort required to exploit it is minimal.

The remaining of this section covers all aspects of programming with Active Data through a synthetic use case involving three characters working in a neurology laboratory: Louisa, Bob and Wallace. A particular experiment involves measuring brain activity using EEG over long periods of time (several hours at once). Figures 3.12 presents the workflow setup for this experiment: several electrodes are implanted in the patient's brain. Each electrode records electric signal in a small part of the brain and transmits it to the acquisition machine through an *analog-to-digital converter* (ADC). Each ADC can handle up to 80 electrodes, hence the presence of 3 ADCs allowing to use up to 240 electrodes at the same time. ADCs have a limited storage, allowing them to keep recording for a few hours when they are unplugged from the acquisition machine. On the acquisition machine, Louisa's application controls the electrodes and

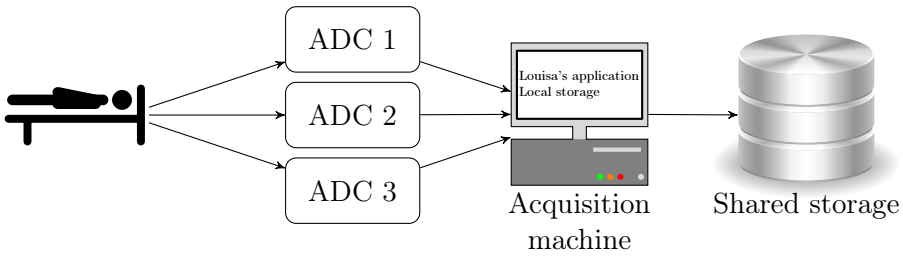


Figure 3.12: Workflow setup for the example use case. The patient is connected to the acquisition machine through one to three analog-to-digital converters. The acquisition machine performs analysis and stores raw data and results on the shared storage.

drives experiments. The limited local storage allows to edit and analyze raw signal for one experiment at a time; raw signal and results are then transferred to a 9TB shared storage.

We summarize the work and responsibilities of Louisa, Bob and Wallace as follows:

Louisa is a research engineer in charge of installing and configuring the EEG recording application, and implementing the various analysis algorithms needed by researchers.

Bob is a technician responsible of IT support in the laboratory; one of his duties is to maintain the shared storage system that researchers use to store raw EEG signal, experimental intermediate data and results.

Wallace is a neurologist. He tries to find evidence of a pattern in brain activity specific to a certain kind of seizures; he records data with the acquisition machine, processes records with Louisa's application and stores experimental data and results on the shared storage.

The workflow is data intensive: each electrode produces a files of about 1 gigabyte per hour of recording. The size of the experimental data makes it difficult for Wallace and other scientists to manage. They would all like to have a more integrated solution so they could keep track of their data from the electrodes to the storage, through the ADCs and the acquisition machine. In particular, they do not want to accidentally remove or lose files, they do not want to forget to which patient a file belongs or what treatment has been applied to a recording. Because their days are already quite busy, they also want to be notified of progress without having to directly check the programs.

Louisa and Bob decide to turn their applications *Active Data-enabled* to allow their users to automate custom task at key steps without having to modify any of the applications for each user.

3.2.2 Life Cycle Object Model

The first step for Louisa and Bob is to represent the data life cycle of their systems separately. They have drawn their complete life cycle model which is presented on Figure 3.13. Because the ADCs and the acquisition software are tightly connected, Louisa decides they are part of a single system. The remote shared storage, maintained by Bob, is a second system. The composition transitions are part of Wallace's particular workflow.

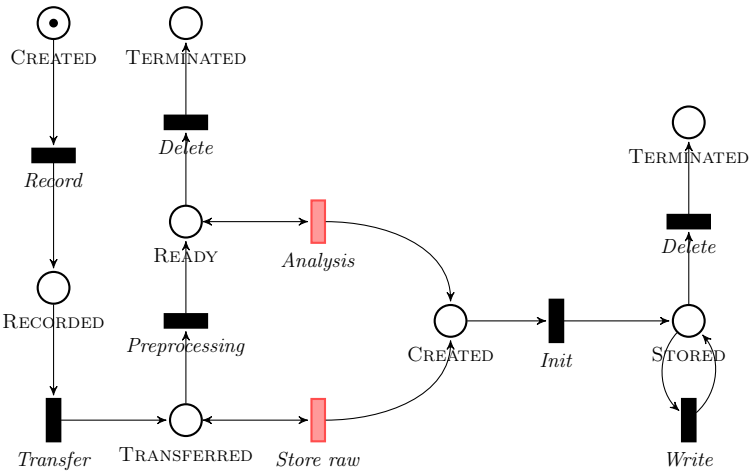


Figure 3.13: Life cycle model for the example use case. The life cycle created by the analog-to-digital converters and programs running on the acquisition machine is drawn on the left. To the right of the two composition transitions is the model of the life cycle created by the storage system.

Louisa focuses on data stages rather than physical infrastructure; this is why the ADCs and the machine do not appear in the life cycle model. In the first system, a token represents a file containing EEG data. The token on the Recorded place represents that the acquired file is still present on the ADC's memory. The token moves to the Transferred place after the file reaches the acquisition machine. After some preprocessing operations return, the token moves to the Ready place, meaning it is now ready for analysis. From this point, the data file can be deleted from the acquisition machine at any time with the *Delete* transition. There are two points of entry between the acquisition machine and the shared storage: the *Store raw* composition transition (when the raw data file is transferred to the shared storage) and the *Analysis* composition transition (when new analysis results are produced and stored directly on the shared storage). *Analysis* can be executed several times to produce more results, as long as the file is not deleted, and the token is on the Ready place.

The Created place of the shared storage represents an empty file, similar to the result of the "touch" Unix utility command. When the transfer from the source to the shared storage is completed, *Init* is executed and the token is moved to Stored. After that, the file can be overwritten several times or deleted.

Now, Louisa and Bob have to represent their life cycle models programmatically. To do so, they use the object-oriented representation of the meta model offered by Active Data.

The root of the model is a class called `LifeCycleModel`; objects of this class link to `Place`, `Transition` objects.

Listing 3.1 presents the pseudocode for the life cycle model drawn on Figure 3.13. The pseudocode is divided in three parts: first comes the definition of the left portion of the life cycle model, then comes the definition of the storage life cycle model, and the last two lines compose them. For conciseness, we present all the code at once, but Louisa and Bob conveniently put each model in a separate file that they distribute to Wallace. Wallace, on his side, gets the two models and compose them in a third file that is specific to him.

```
// Create the first life cycle model, specifying its SID
LifeCycleModel appModel = new LifeCycleModel("application");
// Add places, transitions and arcs
Place appCreated = appModel.getStartPlace();
Place recorded = appModel.addPlace("Recorded");
Place transferred = appModel.addPlace("Transferred");
Place ready = appModel.addPlace("Ready");
...
Transition record = appModel.addTransition("Record");
Transition transfer = appModel.addTransition("Transfer");
...
appModel.addArc(appCreated, record);
appModel.addArc(record, recorded);
...

// Create the second life cycle model
LifeCycleModel storageModel = new LifeCycleModel("storage");
Place storageCreated = storageModel.getStartPlace();
Place stored = storageModel.addPlace("Stored");
...
Transition init = storageModel.addTransition("Init");
Transition write = storageModel.addTransition("Write");
...
storageModel.addArc(storageCreated, init);
storageModel.addArc(init, stored);
storageModel.addArc(stored, write);
storageModel.addArc(write, stored);
...

// Compose the two models with composition transitions
appModel.addCompositionTransition("Store raw", transferred, storageModel);
appModel.addCompositionTransition("Analysis", ready, storageModel);
```

Listing 3.1: Pseudocode for the example life cycle models. Both models are implemented separately and then joined by composition transitions.

Constructing a `LifeCycleModel` requires the system identifier (SID) of the system; this class has methods for adding places and transitions, that also require names. Then the places and transitions contained in the `LifeCycleModel` object can be retrieved using the SID and the name associated with the place or transition:

```
Place p = appModel.getPlace("application.Recorded");
Transition t = appModel.getTransition("storage.Init");
```

This feature is very useful when using composition; in the example, after joining the two systems with composition transitions, only `appModel` is manipulated; it now contains the places and transitions of both models, and they are named with the "SID.name" format.

The call to `addCompositionTransition(String name, Place source, LifeCycleModel destination)` does three things: *i*) it adds a composition transition to the model, named `name`; *ii*) it adds an arc between the `source` place and the new composition transition; and *iii*) it adds an arc between the new composition transition and the created place of the `destination` model.

Louisa and Bob have now fully modeled their life cycle model, and Wallace has a complete representation of his experiment workflow. However, to fully benefit from the power of Active Data, they must now link their application to the life cycle model.

```
// Make the uid and setup the experiment parameters
String uid = "Doe_2400Hz_1";
String outputPath = "/path/to/raw/data/" + uid;
setupExperiment(outPath);

// Publish the new life cycle
ActiveDataClient client = ActiveDataClient.getInstance();
LifeCycle fileLc = client.createAndPublishLifeCycle(appModel, uid);

// Actually start the experiment
startExperiment();
```

Listing 3.2: Pseudocode for publishing a new life cycle. The UID used is a combination of experimental parameters that allow to quickly reference the actual data file.

3.2.3 Publishing a New Life Cycle

Louisa, Bob and all the users of Active Data will manipulate `LifeCycle` objects. A `LifeCycle` represents a data item in the system—a file in our example; it links to the `LifeCycleModel` and contains all the tokens on the right places.

Now, Louisa's application and Bob's storage system must create `LifeCycle` objects to link their files to Active Data. After that, the service saves the state of the life cycles and is able to receive transition publications regarding the newly created file. Informing the service that a new data item has been created is called *publishing a new life cycle*.

The arguments required to publish a new life cycle are the life cycle model and the data item unique identifier. In her application, Louisa makes a call to the Active Data client interface when an experiment starts, before any data is recorded (Listing 3.2). In the code, we see that Louisa chose to use experimental parameters as unique identifier[1] for the life cycle. Louisa is clever: if she had chosen to use the file path as identifier, she would have had to deal with different absolute paths between the ADCs and the acquisition machines and file movements. Instead she uses an identifier that:

- Allows to construct the target file absolute path both on the ADCs and on the acquisition machine;
- Can be constructed on the fly: later in her code, when Louisa faces "experiment number 1 for patient John Doe at 2400Hz", she can reference the life cycle as the one identified by "Doe_2400Hz_1".

Sometimes users want to be informed that a new data item has been created in a particular system. Subscribing to transitions from the model will delay the information because an arbitrarily long time can pass between the creation of a data item and its first transition. For this reason, an additional transition called *create transition* is automatically added to each life cycle model. When a client publishes a new life cycle, the service silently publishes its create transition. Clients can subscribe to this transition like to any other. Clients get the create transition from a life cycle model with a method call on a `LifeCycleModel` object:

```
Transition create = appModel.getCreateTransition();
```

Louisa's application now creates a life cycle in Active Data for every new experiment started by Wallace and other users. These life cycles all have only one token on the model's CREATED place.

[1] The identifier here is only hardcoded for readability.

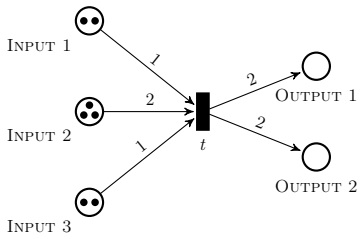


Figure 3.14: Transition requiring the publishing process to specify which input tokens are consumed, and how they are laid out on the output places.

3.2.4 Publishing Transitions

Now, Louisa's application must move the unique token originally present on the Created place to other places to reflect the progress of files in her application.

The application and the storage system must notify Active Data of the operations they perform on data. This is called *publishing* a transition; transitions must be published as soon as possible after an operation is performed. In the simplest case, Louisa's application stored the `LifeCycle` object that was returned when she published the new life cycle (Listing 3.2). Publishing the first transition (after a record is stored on the ADC) is a single call:

```
LifeCycle fileLc = client.createAndPublishLifeCycle(appModel, uid);
...
Transition t = appModel.getTransition("application.Record");
client.publishTransition(t, fileLc);
```

The first line is the same as before and just creates a life cycle; the last two lines get the `Transition` object by its name and publish it by calling the `publishTransition(Transition, LifeCycle)` method of the Active Data client interface; the transition to publish and the life cycle that is being updated are given as arguments.

This function call is asynchronous: it returns immediately, regardless of whether someone has subscribed to the transition or not.

In more complex cases, the transition's input place has more than one token and triggering the transition requires to tell Active Data *which* token has to go through the transition. An even more complex case is when the transition has several input and/or output places; in this case (such as the one depicted on Figure 3.14), the publishing code must specify:

- What tokens are consumed, and from which input places;
- How the output tokens are distributed over the output places.

To specify this behavior, developers can attach a *transition dealer* to transitions in the life cycle model. A dealer is a class that extends the `TransitionDealer` class. It is implemented by the publisher—in our case, Louisa and Bob—and must implement the `doDeal(LifeCycle, Transition)` method. When implementing the dealer, the publisher has access to three primitives; calling these primitives while the publication is being processed indicates how tokens are consumed and produced:

`consume(Token)` indicate that the specified token is consumed by the transition;

`produce(Token, Place)` indicate that the specified token is produced by the transition on the specified place;

`produceNewToken(Token, Place)` indicates that a new token is produced by the transition on the specified place.

The difference between `produce` and `produceNewToken` is that the former indicates that the "produced" token is in reality a token moved from an input place to an output place; the latter indicates a new token had to be produced because the total number of output tokens is greater than the total number of input tokens for this transition.

Listing 3.3 gives an example of transition dealer for the transition on Figure 3.14. The core of the code is the set of calls to the `consume` and `produce` methods that orchestrate the movement of tokens. Before that, the sample shows how to examine the places and get the tokens they contain, in order to pick and chose the tokens. The method `LifeCycle.getTokens(Place)` returns the tokens contained on a given place for a given life cycle. The result is a hash table that maps a replica identifier (see Subsection 3.1.1.5) to a `Token` object. The end of the example shows how the dealer is attached to the transition; the transition is then published normally.

The dealer —which gets a chance to examine every place and token and decide which to pick— is called internally by Active Data when a process is attempting to publish the transition it is attached to. If no dealer has been explicitly attached to the transition (which is the general case) a default transition dealer is present. For complex cases such as the one we are discussing, the default transition dealer only guarantees that the right number of tokens is consumed from the input places, and that the right number of tokens is produced on the output places. If the number of output tokens is greater than the number of input tokens, the default dealer produces the necessary number of new tokens by calling `produceNewToken(Token, Place)`.

Note that a transition dealer may be attached to multiple transitions, but that a transition can only have one dealer attached. Once the dealer is attached, it replaces the default behavior and is called each time the transition is published.

At this point, Louisa's application is *Active Data-enabled*. Bob does the same with his storage system. In order for Wallace's files in the storage system to be linked through Active Data to the files in Louisa's application, some additional code is required to publish the composition transitions.

3.2.5 Publishing a Composition Transition

To inform Active Data of the link between Louisa's application and Bob's storage system, Louisa's application must publish a composition transition every time it sends a file to Bob's storage system.

Publishing a composition transition is similar to publishing a regular transition, except it requires an additional argument: the unique identifier of the data item in the new system. Listing 3.4 demonstrates how to publish the *Analysis* transition.

Here, the composition transition consumes the only token present on the Ready place. This token's unique identifier is "Doe_2400Hz_1"; the analysis task reads its input file on the acquisition machine and stores the result in Bob's storage system. In return, the storage system assigns a numerical identifier of its own to the result file, and returns it. This numerical identifier (for example, 1234) allows Wallace to later get the physical file in the storage system. When Louisa's application publishes the

```
TransitionDealer dealer = new TransitionDealer() {
   public void doDeal(LifeCycle lc, Transition transition) {
      // Get the input and output places from the life cycle model
      LifeCycleModel model = transition.getModel();
      Place input1  = model.getPlace("example.input1");
      Place input2  = model.getPlace("example.input2");
      Place input3  = model.getPlace("example.input3");
      Place output1 = model.getPlace("example.output1");
      Place output2 = model.getPlace("example.output2");

      // Get the input tokens, indexed by their replica id
      Map<Integer, Token> t1 = lc.getTokens(input1);
      Map<Integer, Token> t2 = lc.getTokens(input2);
      Map<Integer, Token> t3 = lc.getTokens(input3);

      // Pick the tokens to consume
      Token token1 = t1.get(1);
      Token token2 = t2.get(6);
      Token token3 = t2.get(2);
      Token token4 = t3.get(4);

      // Consume the input tokens
      consume(t1);
      consume(t2);
      consume(t3);
      consume(t4);

      // Move the tokens to the output places
      produce(token1, output2)
      produce(token2, output2)
      produce(token3, output1)
      produce(token4, output1)
   }
}

Transition transition = model.getTransition("example.t");
transition.setDealer(dealer);
client.publishTransition(lc, transition);
```

Listing 3.3: Example of transition dealer implementing a custom behavior for the transition featured on Figure 3.14. The dealer is attached to a transition, and the transition is published.

composition transition with this identifier, Active Data creates a new token on the CREATED place of the storage model. The identifier of the new token is "4321".

Because composition transitions only consume one token and produce a single new token on one place, no transition dealer is ever needed. The only question is which token is consumed by the transition, and it is answered directly in the call.

Now Louisa's and Bob's systems are fully *Active Data-enabled*. For any operation from the life cycle model performed on data by either system, the life cycle in Active Data is updated to reflect the current state of the data.

3.2.6 Transition Handlers

Wallace is quite happy now. The two applications he uses every day can now report to him programmatically. With little programming savviness he can spend less time monitoring his experiments and more time writing his article. Indeed, reacting to data transition publications requires to write a *transition handler*, that is then *subscribed* to a transition using the Active Data client interface. Active Data later decides when to

```
// Publish the new life cycle and transitions
String uid = "Doe_2400Hz_1";
LifeCycle fileLc = client.createAndPublishLifeCycle(appModel, uid);
...
client.publishTransition(preprocessing, fileLc);

// Launch an analysis that writes its results in the remote storage
Long storageId = analyze("/path/to/data/" + fileLc.getUid());

// Publish the composition transition with the storage identifier
Token token = fileLc.getTokens(ready).get(0);
Transition trans = appModel.getTransition("application.Analysis");

client.publishCompositionTransition(trans, token, storageId);
```

Listing 3.4: Publication of a composition transition. The identifier for the new token is provided by the destination system, then passed to Active Data.

execute the handler.

Wallace decides to start with an easy handler: he wants to receive an email every time an acquired file is ready for processing, i.e. its token is on the READY place, right after the transition *Preprocessing* is executed.

```
TransitionHandler handler = new TransitionHandler() {
   public void handler(Transition transition, bool isLocal, Token[] inTokens, ↘
       →Token[] outTokens) {
      Token fileToken = outTokens[0];
      String uid = fileToken.getUid();
      String body = "The experiment " + uid + " produced a file that is ready ↘
          →to be processed in Louisa's application.";

      sendEmail("wallace@domain.edu", "EEG record ready to be processed", body);
   }
}
```

Listing 3.5: Example of a transition handler that sends an email every time the transition *Preprocessing* is published.

The handler is presented in Listing 3.5. A handler is an object that implements the `TransitionHandler` interface. Wallace writes his code in the `handler` method. The method receives four arguments that provide the context necessary to write flexible code:

`transition` is the object representing the transition that was published and caused this handler to be executed;

`isLocal` is `true` only if the client running the handler is the same client that published the transition (this is valid because in Active Data every client can be both subscriber and publisher);

`inTokens` the set of tokens that were consumed by the transition;

`outTokens` the set of tokens that were produced by the transition.

In the handler, Wallace examines the token produced by the transition, extracts its unique identifier and uses it in the email body. Since the couple (SID, UID) links to the real data file, Wallace could even attach the preprocessed file to the email. Additionally, tokens also link to the complete `LifeCycle` object which allows to observe

where the other tokens are located, and examine the complete state of the data.

The `Transition` argument allows to tie the same handler code to several transitions. In addition, the same `TransitionHandler` object is used by Active Data every time in needs to be executed; in other words, transition handlers are stateful. For example, imagine Wallace does not want to get an email every time a file is preprocessed; he would like the handler to send en email every time 5 files are ready instead. Listing 3.6 shows his new transition handler.

```
TransitionHandler handler = new TransitionHandler() {
   private int count = 1;
   private uids = "";

   public void handler(Transition transition, bool isLocal, Token[] inTokens, ↘
       →Token[] outTokens) {
      // Store a string that contains all the UIDs
      uids += outTokens[0].getUid() + "\n";

      // Continue only 1 time out of 5
      if((++count % 5) == 0)
         return;

      // Construct a message using all the UIDs
      String body = "The experiment produced 5 files that are ready to be ↘
          →processed:\n" + uids;
      sendEmail("wallace@domain.edu", "EEG records ready to be processed", ↘
          →body);
      uids = "";
   }
}
```

Listing 3.6: Example of a stateful transition handler that sends an email every 5 times the transition *Preprocessing* is published.

This new handler is slightly different; every time it is executed, it looks at the token produced on the Place and gets its UID. The UID is concatenated to a string that will be joined to the mail body. In addition to this string, the handler stores a counter that counts the number of times it has been executed since the last email was sent. When the counter is equal to 0 (one time out of five), the email is sent to Wallace.

Wallace now has his handler ready for Active Data. The next step is to subscribe it to the *Preprocessing* transition.

3.2.7 Transition Subscription

Wallace now wants to attach his transition handler to a transition, so it can be run when a transition is published; this action is called *subscribing* to a transition.

Because many data items share the same life cycle model, the number of transitions being published can grow rapidly. Wallace, however, wants his handler to be executed only when it is relevant to him, and ignore everything else. Active Data allows programmers to cut down the number of transitions their code will react to by providing two kinds subscriptions: *i*) subscribing to a specific transition for any data item and *ii*) subscribing to a specific data item for any life cycle transitions.

Paying close attention to Wallace's handlers in Listings 3.5 and 3.6, we notice that he doesn't test the value of the `Transition` argument; he does not need to, because

he subscribed the handler to the *Preprocessing* transition only. Listing 3.7 defines a handler and now subscribes it this particular transition.

Now, imagine that Wallace has an extremely important experiment. This time, he wants to be notified of every transition, but only for a particular life cycle. Listing 3.8 shows how he uses the second type of subscription to subscribe not to a transition, but to a life cycle. The handler now includes the name of the published transition in the email.

As another way of avoiding getting overloaded by transition notifications, Active Data provides a method to return a copy of a life cycle (including all places and the tokens they contain). This feature, discussed in Subsection 3.2.10, allows Wallace to occasionally get the state of a life cycle without reacting to all the intermediate transitions. Even more ways to filter out irrelevant transition events are detailed later in this section, in particular in Subsection 3.2.9.

Wallace now gets notifications that allow him to spend less time looking at progress bars, and to spend more time writing his article. He will soon find out he can do more with Active Data's advanced features.

3.2.8 Tagging

As explained in Subsection 3.1.2.6, tags are attached to and removed from tokens by transitions. In this subsection we describe how token tags are coded and the possibilities Active Data offers for tagging.

3.2.8.1 Token Type

Despite containing results for varying experimental conditions, Wallace's files are represented in Active Data with identical tokens—except their identifiers. Token tags can be used to attach types that reflect data types and many more things to tokens.

Active Data provides a specific method to examine tokens and determine the presence of tags:

```
Token.hasTag(String tag)
```

The type of a token—defined as the set of all its tags—can be retrieved using a method that returns the set of tags attached to a token:

```
Token.getTags();
```

3.2.8.2 Taggers

Active Data offers Wallace two ways to tag tokens; the main difference between both is where tagging occurs:

```
TransitionHandler handler = new TransitionHandler() {
   ...
};

// Subscribed the handler to a single transition
Transition preprocessing = appModel.getTransition("application.Preprocessing");
client.subscribeTo(preprocessing, handler);
```

Listing 3.7: Subscribing to a single transition: the handler will be executed for any data item going through this transition only.

```
TransitionHandler handler = new TransitionHandler() {
   public void handler(Transition transition, bool isLocal, Token[] inTokens, ↘
       →Token[] outTokens) {
      String uid = outTokens[0].getUid();
      String transitionName = transition.getName();
      String body = "Life cycle " +
                    uid +
                    " has passed transition " +
                    transitionName;
      sendEmail("wallace@domain.org", "Progress", body);
   }
};

// Create a life cycle, then subscribe to it
LifeCycle lc = client.createAndPublishLifeCycle(appModel, uid);
client.subscribeTo(lc, handler);
```

Listing 3.8: Subscribing to a single life cycle: the handler will be executed for any transition traversed by this life cycle only.

- Attaching tags to tokens in a transition dealer, i.e. on the publishing process side;
- Attaching a *tagger* to the transition in the model, i.e. on the Active Data Service side (the tagger being part of the life cycle model).

The first option simply relies on two methods to call inside a dealer to add and remove tags from a token:

```
Token.addTag(String tag);
Token.removeTag(String tag);
```

The second option makes use of objects called `Tagger`. A tagger is attached to a transition in the `LifeCycleModel`. When present, it is automatically called when the Active Data Service receives a publish request from a client process. Thus, unlike the option involving a dealer, using a tagger does not require any client intervention and is not affected by client-side errors.

```
Tagger tagger = new Tagger() {
   public void tag(Transition transition, Tokens[] inTokens, Token[] ↘
       →outTokens, Token[] newTokens) {
      for(Token t: newTokens)
         t.addTag("raw data");
   }
}

Transition storeRaw = appModel.getTransition("application.Store raw");
storeRaw.setTagger(tagger);
```

Listing 3.9: Tagger example: tag each new token in the storage system coming from the *Store raw* composition transition.

A tagger is an object that implements the `Tagger` class and its `tag(Transition, Token[], Token[], Token[])` method. This method receives four arguments: the transition being published, and the tokens going through the transition, ordered in three sets: input, output and new tokens; these three sets result directly from the dealer.

As a first example, Wallace would like tokens in the storage system to be tagged differently if they represent raw data. A simple way to do it is to add a tagger to the *Store raw* transition. This way, every time a client publishes this transition, Active Data attaches the "raw data" tag to the output token. This solution is presented in Listing 3.9.

```
TransitionHandler handler = new TransitionHandler() {
   public void handler(Transition transition, bool isLocal, Token[] inTokens, ↘
       →Token[] outTokens) {
      ...
      sendEmail("wallace@domain.edu", "EEG raw record written", body);
   }
}

HandlerG guard = new HandlerGuard() {
   public boolean accept(Transition transition, Token[] tokens) {
      return tokens[0].hasTag("raw data");
   }
}

Transition = appModel.getTransition("storage.Write");
client.subscribTo(transition, handler, guard);
```

Listing 3.10: Using a handler guard to run a transition handler only for some types of tokens. The handler code, very similar the one in Listing 3.5, is not shown.

Taggers have the ability to examine the actual file represented by a token and place tags according to information external to the model. Thus, taggers are a good way to inject external information in the live life cycle. As the tags will stick to tokens during composition, the information they carry will be available to new systems when tokens reach them.

3.2.9 Handler Guards

In additions to mechanisms already discussed to cut down the number of transition notifications a client receives, we introduce *handler guards*. Handler guards are not part of the life cycle model and are not the responsibility of the system's developer. Instead, they are specific to users and are attached to subscriptions.

When subscribing to a transition, Wallace can provide an object called `HandlerGuard`. This object implements a single boolean method `accept(Transition, Token[])`. The Active Data Service stores handler guards along handler subscriptions. When Wallace calls a variant of `publishTransition`, the service passes the corresponding couple $(transition, inputtokens)$ to the `accept` method; if the method returns `true`, Wallace's handler will be executed; otherwise, the event is dropped. Since guards are associated to subscriptions, Wallace's guard has no effect on other clients.

We illustrate this principle with Listing 3.10. Here Wallace wants to benefit from the "raw data" tag placed by his tagger in Listing 3.9; he wants to receive an email each time a raw data file is overwritten, which corresponds to the transition *Write*. He starts by writing a very simple handler that always sends an email; then he writes a guard that returns `true` only if the transition's input token has the tag "raw data"; finally he subscribes his handler to the *Write* transition, along with the guard. Because the Service notifies the client only if the guard returns `true`, the handler needs not test the presence of the tag and is executed fewer times.

Handler guards are very effective on reducing the number of handler executions because they are evaluated by the service. They also allow clients to write simpler code; the alternative approach would be to start a transition handler with a test; this would lighten the charge on the service but also make client applications trickier to

```
String storageSID = "storage";
String storageUID = "36253";

ActiveDataClient client = ActiveDataClient.getInstance();
LifeCycle lifeCycle = client.getLifeCycle(storageSID, storageUID);
Place ready = appModel.getTransition("application.Ready");

Set<Token> appTokens = lifeCycle.getTokens(ready).values();
Token token = appTokens.get(0);
String appUID = token.getUID();

copyDataFromAcquisitionMachine(appUID);
```

Listing 3.11: Querying the complete life cycle of a data item from a partial knowledge. The UID of a token links to the real life data.

implement and lead to poor design.

3.2.10 Life Cycle Querying

Offering a complete view of the state of data distributed over multiple heterogenous systems and infrastructure is a strong motivation for this work, and guided the design of our meta-model. To this end, we briefly introduced the `LifeCycle` class that allows to examine all places, transitions and tokens for any data item. An important question remains: how to make this information accessible to any programmer, no matter where their code is running in a complex distributed application?

To answer this question, we consider the only piece of information that is available to any code, in any system: the couple (SID, UID). In other words, we consider that any piece of code handling a data item can answer two questions: *i*) in what system is this code running? and *ii*) what is the local identifier of the data item being manipulated?

Active Data includes primitives for querying the whole life cycle of any data item. In order to make these primitives available anywhere in any system, they require the two arguments discussed above: a SID and a UID. Let us consider the storage system of our example use case (Figure 3.13); the storage system is used for many applications other than Louisa's. This system is extremely minimal and does not store the origin of data. Unfortunately, a problem occurred: Wallace noticed a specific file is corrupted, and it has to be taken from the source again.

The storage system cannot possibly know that the file is coming from the acquisition machine, and that a copy may still be there. Using Active Data, Wallace's view expands beyond the storage system and he gets information pointing to the source (and the identifiers) in Louisa's application. This principle is illustrated in Listing 3.11; from the local knowledge, in the storage system, of the system identifier ("storage") and of the unique identifier of the corrupted file ("36253"), a query for the complete life cycle is made to the Active Data Service. Then the code examines the returned `LifeCycle` and the READY place in particular (we could have examined every place to determine where tokens are, but we decide to keep this example short); the first token found is also examined to get its UID; the unique identifier, as said previously, links directly to the real-life file.

The query feature is an important building block for data integration: Active Data integrates under a single namespace the identifiers of the same data from different systems that were not designed to collaborate and provides them to users and any

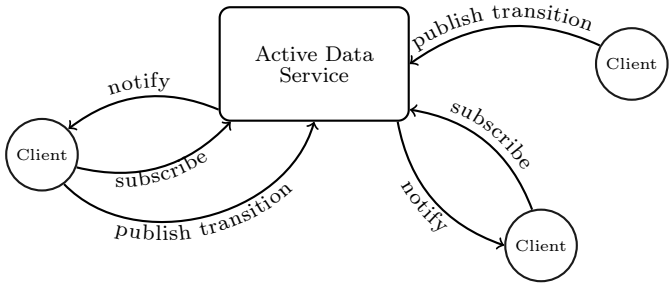


Figure 3.15: Architecture of Active Data: clients (data management systems and users) communicate with a centralized service in a Publish/Subscribe fashion.

process.

3.3 System Design

In this section we discuss the prototype architecture of Active Data and design choices.

3.3.1 Architecture

Active Data is composed of two parts: *i*) the Active Data Service, which manages data life cycles, receives transition publications, notifies clients of publications, and guarantees that the execution is correct with respect to the life cycle model, and *ii*) the programming interface (API), which allows data management systems to publish transitions and programmers to develop applications by subscribing transition handlers.

For the Active Data Service, managing life cycles has two functions: *i*) maintaining the current state of each life cycle and *ii*) allowing clients to query life cycles.

Clients of the Active Data Service are of two kinds:

- data management systems and user applications are client of the Active Data Service when they publish transitions; sometimes the same client that publishes transitions can subscribe to handlers;
- end users are clients of the Active Data Service when they query the state of life cycles and subscribe handlers to get notifications or run local tasks.

At this time the service is centralized, which allows it to easily maintain the consistency of life cycles.

3.3.2 Active Data Service

The Active Data service is responsible for maintaining the current state of the life cycle of any data item in the system. It must satisfy queries from clients who want to obtain the current state of a life cycle and transition publications. This subsection describes how the Active Data Service communicates with client and how the most important operations are executed.

3.3.2.1 Client-Service Communication

Figure 3.15 shows the architecture of Active Data. Multiple clients all communicate with the Active Data Service; clients never communicate with each other and the service

never initiates communication with clients. This *pull* design allows Active Data to work when clients are deployed behind NATs and firewalls that can be found on some infrastructures, like desktop grids.

Because the Active Data programing model is mainly based on two operations that are publishing transitions and subscribing handlers, it seems natural to use the Publish/Subscribe paradigm [156] for client-server communication. To accommodate the two types of clients of Active Data, every client can be both publisher and subscriber at the same time.

The Active Data Service maintains data structures that link each transition to a set of subscriptions, and a messaging queue for each client. Then, when a client publishes a transition, the service issues an *event* for each subscribed client and places it in the client's queue. Events are placed in the queue in the order they arrived on the service, meaning the service maintains a total order on events[2] Subscribers regularly make requests to the service to get all the events in their queue, after which the service clears it. Finally clients can locally run their handlers according to the information contained in events: which transition was published, and on which life cycle.

3.3.2.2 Performance Considerations

The frequency at which clients pull events from the service in an important parameter; by default, it is set to 60 seconds and can be set to a different value by each client. We call this parameter *pull frequency*. A short pull frequency makes a system more reactive (the time between a transition publication and handler execution is smaller) but creates a heavier load on the service. Conversely, a long pull frequency allows the system to handler many more clients but applications relying on handler executions will take more time before being notified of transitions. A pull frequency of a second or under is unnecessary for most applications that notify users, share data sets or perform maintenance tasks on data. On infrastructures like desktop grids where compute nodes are very loosely coupled and can be offline for a long time, a pull frequency of several hours—or even days—can be satisfying.

At this time, the Active Data service is mono-threaded and can process only one client request at a time. However, its event-driven design and the small time required to publish a transition and pulling events allow it to handle a system with thousands of life cycles and clients.

3.3.2.3 Publication Processing

When a client issues a transition publication to the service, the latter performs several tasks to ensure that the request is valid and that the concerned life cycle is left in a correct state. Here are the steps the Active Data Service takes to handle a transition publication:

1. *Checks*: the service checks that the transition being published is actually part of the life cycle model of the data item concerned and that it is enabled;

2. *Perform the transition*: the service moves the tokens from the input places to the output places according to what the client's dealer decided. New tokens are also produced according to the dealer, but the service assigns them definitive replica identifiers that are guaranteed to be unique;

[2]Ordering the events according to when the operation corresponding to a transition took place would require Active Data to implement a global clock, which has not been considered yet.

3. *Execution of the tagger*: if a tagger is attached to the transition, the service executes it, passing the tokens as arguments;

4. *Creation of events*: for each subscribed client, the service creates an event and places it in the client's queue.

3.3.2.4 Taggers Execution

The execution of user-supplied code through taggers during step 3 of the above paragraph brings some difficulties. Because they are executed on the service's main thread during the publication of a transition, it is critical that the code of taggers execute very shortly; a blocking tagger would be a very serious issue as it would slow down the whole system, making the publication of some transitions abnormally long.

It is also very important that taggers remain relatively small to keep the service's memory footprint from growing too large, which would also impair the whole system.

3.3.3 Active Data Client and Execution Model

3.3.3.1 Events Pulling

Clients communicate with the service via an interface called *Active Data Client*. The first time it is called, the interface creates a session with the service and identifies itself with a unique string called *client identifier*. A client can keep the same identifier over several successive sessions. When the client interface receives local publish and subscribe requests, it adds the client identifier and forwards them to the service. When subscribing to transitions, the handlers is not sent to the service. Instead the client interface sends the transition and the client identifier, which is enough for the service to store the subscription.

The Active Data Client interface manages a separate thread called `PullEventsThread` which only purpose is to regularly pulls events from the service —still providing the client identifier—and run transition handlers. All transition publications, subscriptions and other queries are performed on the caller's thread. When the PullEventsThread perform a `pullEvents()` call to the Active Data Service, it gets a list of `Event` objects in return; if no transition to which the client had subscribed has been published since the last call to `pullEvents()`, an empty list is returned. An Event object contains the information necessary to locally run the subscribed transition handlers: the name of the transition that was published, the tokens that were consumed and produced; as a commodity Event object also contain a boolean that indicates whether the client pulling the event was the publisher of the transition.

3.3.3.2 Transition Handlers Execution

When `pullEvents()` returns a non-empty list, the PullEventsThread loops through it and for each event it gets the `TransitionHandler` object corresponding to the transition and executes is `handler()` method. The handlers are run serially by this thread, in a blocking way. Because the events in the queue are totally ordered, the handlers are run in the order the transitions were received by the service. If several handlers were subscribed for the same transition or life cycle, the order in which they are executed is unspecified. Handlers must return shortly to avoid blocking the local queue and perform any lengthy operation in a separate thread. In the case when a handler would take a long

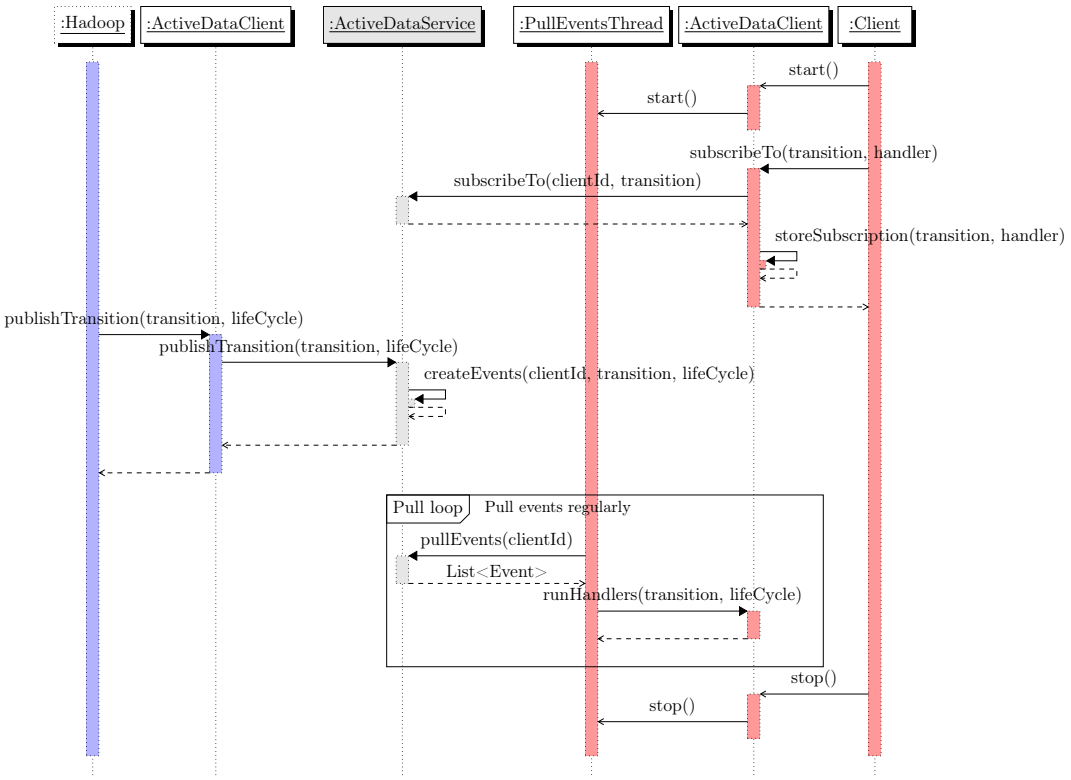


Figure 3.16: Sequence diagram for handler subscription and execution. A publisher ("Hadoop", in red) publishes transitions through its own local Active Data Client. A subscriber ("Client", in blue) also makes calls to its local Active Data Client instance to subscribe to the transition. Once the client is *started* (after the call to `start()`), a separate thread managed by the Active Data Client checks for events with no further intervention from the client.

time to run, it would not block the entire system, but only the local PullEventThread. As such, even a blocked transition handler would not prevent the client to publish further transitions.

This sequence of operation is illustrated by Figure 3.16. This diagram shows that each side—subscriber and publisher—has its own Active Data Client and never communicates with the service directly. The figure also shows that when subscribing to a transition, the transition handler stays on the client and never reaches the service. The colors on the figure indicate where the code is running; threads sharing the same color run on the same machine. It is clear from the diagram that the client code makes a single call for their handler, and that the burden of treating events and managing the execution of handlers is on the Active Data Client.

3.3.4 Verification

Detecting incorrect behavior of processes in large distributed systems is very difficult in nature. In a task centric approach, a task is considered successful as long as it returns successfully (with a 0 return value on Unix systems or the equivalent on other platforms). However, the success—or the absence of explicit failure—of a task does not mean the data is left in the expected state. Focusing on the progress of data life cycle,

incorrect steps can be detected very early, as soon as an operation attempts to modify the state of a data item in a way that is not valid with respect to the life cycle model.

Coupling the life cycle meta-model with the service-side runtime environment allows model checking at runtime. Every time a client attempts to publish a transition through the API, the Active Data Service checks the request against the life cycle it has in memory to determine if it is valid in two ways:

- The transition published must be valid (i.e. be part of the concerned life cycle) and enabled;
- the tokens specified by the transition dealer must be on the place the dealer claims they are.

This verification allows to maintain a consistent life cycle on the service, forbidding several clients to consume several times the same tokens with one or different transitions. Additionally, the service raises an exception each time a client tries to publish a transition in a way that violate the model, making client processes fail early. In addition to providing the information earlier to users, the information is more accurate, since the granularity of data life cycles is finer than the one of tasks (as discussed in Subsection 3.1.1.1 and with Figure 3.2 in particular).

3.3.5 Consistency

When a client makes a query to the Active Data Service for the current state of a life cycle, the returned object might already be out-of-date; this happens when another client publishes a transition between the moment when the service returns the life cycle and the client gets a chance to examine it. This implies that the view that clients have of life cycles can always be inconsistent with the actual state of the life cycle, and that the only consistent copy of a life cycle is the one that exists on the service. It is neither good practice nor efficient to subscribe to all transitions to keep a local up-to-date copy of a life cycle because this creates an unnecessary load on the service and the client and does not prevent inconsistencies.

3.3.6 Techniques to Publish Transitions

Different strategies exist for making a system *Active Data-enabled*, i.e. making it publish transitions to an Active Data Service. We identify, detail and discuss the advantages and disadvantages of three of them: instrumenting the system, reading its logs and using an existing notification system.

3.3.6.1 Instrumentation

The most obvious and straightforward way of making a data management system Active Data enabled is to instrument its code. In the code of the system, each time an important operation is performed, we add a call to the Active Data client API to publish the corresponding transition.

There are many advantages to this technique: because the transition is published as soon as the operation is done, the service is informed right away, with no delay. It is also the technique that allows the finest grain for life cycle models, because every operation can be represented on the model. However instrumenting the code requires to have access to the code, which is not always the case. Additionally, this technique

is intrusive: the API calls to Active Data can get in the way of the system code, and make it harder to maintain. This last point is alleviated by the relatively small size of the Active Data library and the conciseness of its API calls.

This technique is the one recommended for developers that want to make their data management system Active Data-enabled in the most accurate way (without missing any important operation) and allow their users to connect it to Active Data effortlessly.

3.3.6.2 Log Processing

Another approach is to publish transitions based on application logs. Many applications provide one or several log files of varying verbosity as they run. It is often possible to read about the creation of data items or operations from the logs. For this to work, the logs must present at least a string that uniquely maps to an operation—so we know which transition to publish—and the identifier of the data item.

The framework involved is always the same: the application is running normally and writes messages to a log file. In parallel, a "scraper" process reads the log file, extract information about data operations, translates them to life cycle transitions and publishes them. The application is usually not aware of Active Data and requires no modification.

The main advantage of this technique is that it is completely not intrusive, and does not require the application source code. It is also a viable alternative to instrumenting the source code even when we have access to it: inserting well formed log messages in the code is less intrusive, does disturb the maintenance of the application and allows to achieve about the same result.

The first downside of this approach is that it requires the implementation of a scraper program and to run it beside the application. The second important downside comes from the decoupling of the application and its scraper: in many systems, logs are insufficient to get the desired level of detail, and the resulting life cycle model will be coarse grained; in addition, because the format of logs is rarely documented and fixed, application updates may break the scraper.

This technique is the right tradeoff for users of a non Active Data-enabled system when they do not have access to the source code, or do not have the resources to study the code and instrument it.

3.3.6.3 Notification Systems

The third approach uses notification services existing in applications and translates native notifications into life cycle transitions. As with log processing, this approach requires an additional program to run beside the data management system. This program gets notified of events directly by the system, finds the life cycle transition that corresponds to each event, and publishes it to the Active Data Service. Such notification systems are available in many system, like triggers' in a database management system, inotify [157] for Linux file systems or even email notifications.

This approach shares advantages and disadvantages of the two previous approaches. It is completely non intrusive, and usually well documented. Notifications are usually delivered quickly, which allows the same level of reactivity than instrumentation without having to modify the code of the application. A second program must be developed on the side to receive notifications from the system and treat them, but it is more efficient than with log processing thanks to its event-driven design. Like with log processing, the level of detail provided by notifications might not be enough

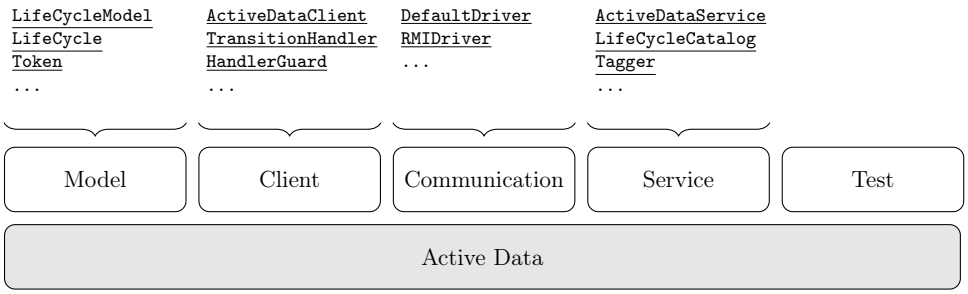


Figure 3.17: Project structure: the Active Data library is organized in four code modules and a module for unit testing; the "Service" module contains the code of the Active Data Service and the "Client" module contains the client interface used to communicate with the service. The two other modules are used by both clients and the service. Above each module is presented a preview of the most important classes it contains.

to determine precisely what transition is to be published, and the granularity of notifications may not cover all the needs of users.

To conclude, the choice of one approach compared to the others depends on the target system and on the needs of users. A user may be satisfied simply reading the standard output of a program and publishing a couple transitions during a very long run. However, the help of the system developer always achieves the best results.

3.4 Implementation

This section gives about the implementation of Active Data.

3.4.1 Library Description

Active Data is released as a Java JAR library; the code base represents about 4,500 lines of code (excluding comments). The client library was designed to be minimal, allowing it to run on big supercomputers as well as the most restrictive platforms, such as sensors or acquisition machines. Its small compiled size of about 140Kb, its small memory footprint and low CPU requirement make it easy to deploy on nodes with few memory and computing power.

Active Data is licensed under the GNU General Public License; it development tree is private, but its code is released with each major version and can be downloaded at the Inria Forge at `http://active-data.gforge.inria.fr/`.

Active Data is entirely object oriented. The project structure is described on Figure 3.17. The code base is divided in five modules: "Client", "Service", "Model", "Communication" and "Tests". The "Client" module contains the interface to communicate with the service, which code is in the "Service" module. "Model" contains the meta-model implementation and is used by clients and the service: `LifeCycleModel`, `LifeCycle`, `Place`, `Transition`, `Token`, etc. The "Communication" module contains communication protocols for clients and the service. It is discussed in the next subsection. The fifth module contains over 20 unit test cases for every other modules. The unit tests are automatically run by the compilation workflow, insuring it stops when any test fails.

The code is very flexible and allows essential components to be easily replaced by implementing generic interfaces. As such, offering a new communication protocol between

```
java -cp active_data_lib.jar
      org.inria.activedata.examples.cmdline.PublishTransition
      <AD host>
      [-p Ad port]
      -m <model class>
      -t <transition name>
      -sid <data item SID>
      -uid <data item UID>
      [-newId <new UID>]
```

Listing 3.12: Usage of the command line tool for publishing a transition.

the clients and the service simply requires to implement the `ActiveDataClientDriver` interface; entirely changing the strategy for storing the life cycles only requires to implement the `LifeCycleCatalog` interface.

3.4.2 Communication Protocol

The "Communication" module contains protocol implementations for clients and the service to communicate in a way that meets the user requirements. At this time, two protocols are available: a default protocol making direct method calls—suitable for testing, when the client and the service run inside the same Java Virtual Machine—, and Java RMI.

The modularity of Active Data make it very easy to add new communication protocols by implementing the `ActiveDataClientDriver` interface for the client, and a server counterpart for running on the service.

Clients must implement a maximum of 14 methods such as `publishTransition()`, `newLifeCycle()`, `subscribeTo()`, etc. Some methods are optional, e.g. `connect()` and `disconnect()`.

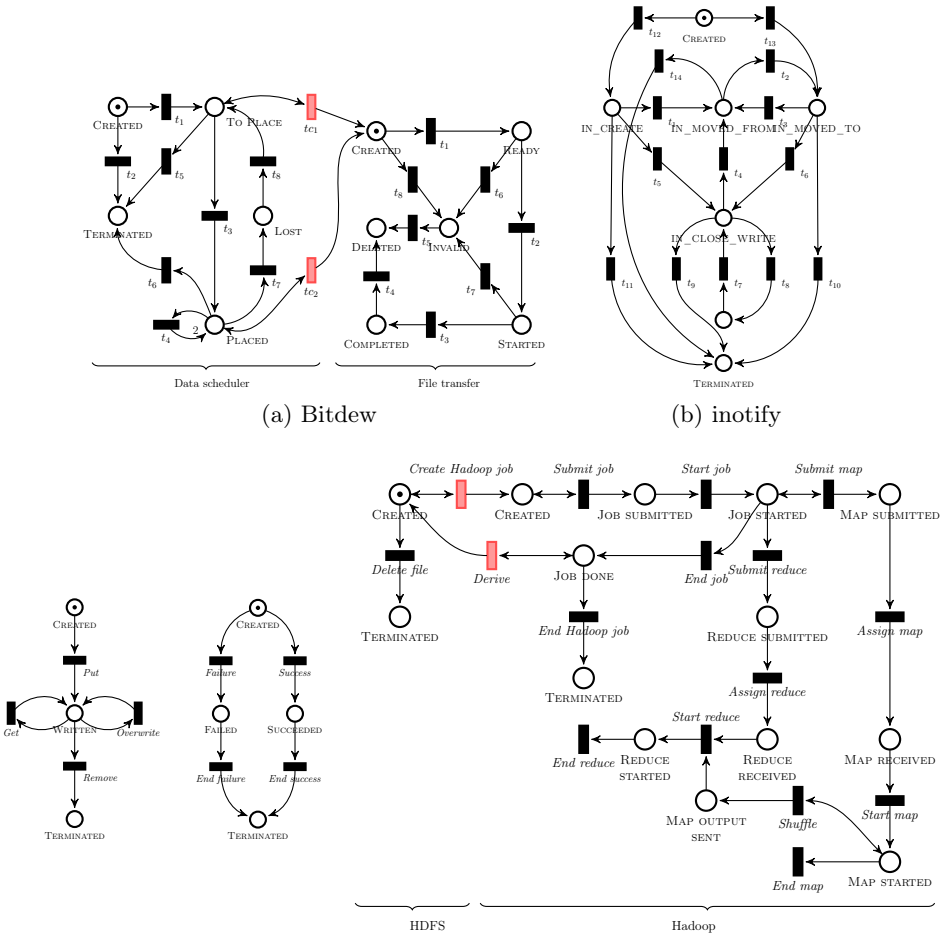
On the service side, a class must listen to client requests and transmit them to the local `ActiveDataService` singleton. It is possible to have several protocols transmitting requests to the Active Data Service because it is thread-safe. For example a RMI server and a HTTP server can send requests to the same Active Data Service.

3.4.3 Command Line Tool

In addition to the Java API, Active Data offers a command line tool for interoperability. The command line tool allow programs to publish transitions with a simple process call, without having to write Java code.

Listing 3.12 shows the usage of the `PublishTransition` command; the first two arguments are the host name or ip address and port of the Active Data Service; the following two arguments are the class containing the `LifeCycleModel` of the transition being published, and the name of the transition; the last two arguments are the SID and UID of the life cycle to update. A last optional argument allows to publish a composition transition, in which case the UID of the new token must be provided.

Listing 3.13 provides a full working example that publishes the transition *application.Transfer* of the life cycle model on Figure 3.13.



(a) Bitdew (b) inotify

(c) iRODS (d) Globus Online (e) HDFS & Hadoop

Figure 3.18: Data life cycle models for five data management systems.

```
java -cp active_data_lib.jar
      org.inria.activedata.examples.cmdline.PublishTransition
      192.168.2.45
      -m active_data_lib.jar org.louisaapp.application.Model
      -t application.Transfer
      -sid application
      -uid 354632
```

Listing 3.13: Example of the command line tool for publishing a transition.

3.4.4 Systems Integration

We report now on the integration of Active Data with five widely used data management systems. BitDew [71] is a middleware developed by Inria for easy data management on various distributed infrastructures; it offers a programmable environment, a data scheduler and a reliable file transfer service. The inotify Linux kernel subsystem [157] allows to *watch* a directory and receive events about the files it contains, such as creation, modification, write, movement and deletion. iRODS [68] is a rule-oriented data management system developed at the DICE center at the University of North Carolina. iRODS provides a virtual data collections of distributed data, a metadata catalog and replication. Globus Online [76] is a file transfer service developed at the Argonne National Laboratory, which offers researchers a fast, simple and reliable way to transfer large volumes of data. Hadoop [81] is a project maintained by the Apache Software Foundation that includes a widely used implementation of the MapReduce [80] programming model and a distributed file system called HDFS [45].

When integrating Active Data into a legacy system, it is necessary to understand how the system exposes its life cycle, in order to detect and convey life cycle transitions. The main approach consists in relying on notifications provided by the system, and publishing these notifications in terms of Active Data transitions. For instance, inotify provides a notification service, which wakes up programs when files are altered on the file system. inotify informs about the new state of the file, thus it is easy to deduct the corresponding place, and publish the corresponding transition. When such a notification service is not available, a second approach is to look at the internals of the system. For instance, iRODS relies on a PostgreSQL database. We implemented a trigger executed by the database each time a file is created or modified in iRODS. The trigger acts as an Active Data proxy and forwards only relevant life cycle transitions to the Active Data Service. A third approach is to deduce data-related events from a system's logs. In the case of Hadoop, for example, the submission, start and completion of jobs and tasks is reported on the job tracker and task trackers logs. A lightweight scraper is executed on every machine of the cluster, watches the logs and publishes the corresponding transitions. Overall, we think there exist many sources of information to obtain complete or partial representations of data life cycles in a non-intrusive fashion: logs, email notifications, databases and so forth.

For each system, we have to represent its data life cycle model. Systems that expose only partially their life cycle make this task intricate: this is the case for Globus Online and iRODS. As Globus Online source code is not available, we have an incomplete knowledge of the file transfer life cycle. However, Globus Online emails the user upon successful file transfer completion or failure. From this partial information, we reconstruct the life cycle model presented in Figure 3.18d and enable Active Data users to monitor Globus Online transfers. With iRODS, we do not need the whole iRODS data

life cycle and chose to represent only the portion that is relevant to our application (see the use-case presented in Subsection 4.2.4). This model is represented in Figure 3.18c.

Conversely, two data life cycles are complete: inotify, BitDew. Figure 3.18b presents the inotify life cycle model constructed from its documentation.

Reading the source code of BitDew, we observe that data items are managed by instances of the `Data` class, and this class has the `status` variable which holds the data item state. Therefore, we simply deduce from the enumeration of the possible value of `status` the set of corresponding places in the Petri Net (see Figure 3.18a). By further analyzing the source code, we construct the model and summarize how high level DLCM features are modelized using Active Data model:

Scheduling and replication Part of the complexity of the data life cycle in BitDew comes from the Data Scheduler that places data on nodes. Whenever a data item is placed on a node, a new replica is created. We represent replicas with a loop on the PLACED state that creates an additional token every time a token passes through it.

Fault tolerance Because one of BitDew's target architectures is Desktop Grids, it must deal with frequent faults, i.e. nodes going offline. When a data item is placed on a node, and the node disappears from the system, it is marked with a LOST state and will be placed on an other node. This is represented by the loop PLACED, LOST, TOPLACE.

Composition of File Transfer and Data Scheduler In BitDew, the Data Scheduler and File Transfer Service are closely related, and so are their life cycles. A file transfer cannot exist without an associated data item, and a deleted data item cannot be transferred. To connect the two Petri Nets we need to define the start and stop places as explained in section 3.1.2.5. In BitDew, a new file transfer can be started for a `Data` object in any state, except TERMINATED, LOST and LOOP. To represent this, we define all the places but the three mentioned above as start places and connect them to the transfer life cycle model.

The life cycle model for Hadoop represents the life cycle of a file used as input for a Hadoop MapReduce job. Because the input and output files of a Hadoop job are stored in HDFS, the life cycle model of Hadoop is connected to the life cycle model of HDFS. The life cycle model for Hadoop and HDFS presented on Figure 3.18e is composed of two separate models; the one on the left represents the life cycle of a file stored in HDFS with a coarse granularity (only one transition is included) because not enough meaningful information can be extracted from the logs yet; the one on the right represents the life cycle of a file during a Hadoop job. The Hadoop model features transitions for job submission and termination, and map and reduce tasks submission, distribution, launch and termination (*Submit map*, *Assign map*, *Start map*, *End map* etc.). It also represents data transfers during the shuffle phase (transition *Shuffle*). A second composition transition called *Derive* represents the production of one or several output files in HDFS.

3.5 Conclusions

In this chapter, we have analyzed the requirements for a meta-model allowing to represent the life cycle of data in any distributed application. We presented a novel data-centric meta-model with Petri Networks as a starting point and adding many specific features and restrictions, such as tokens identification, life cycle composition and termination. From this meta-model, we introduced the Active Data programming model

and runtime system. Active Data allows to represent a data life cycle model in a program and use it for programming high-level applications to notify users, ease sharing, perform optimisations and various tasks. Chapters 5 and 4 present examples of such applications as well as performance evaluations.

4 Evaluation

This chapter studies the prototype implementation of Active Data using micro-benchmarks and synthetic use-case scenarios. Performance metrics will evaluate key features, characterize the quality of the code, identify potential bottlenecks and shed some light on what Active Data is currently suited for, and where there is room for improvement.

4.1 Micro-Benchmarks

This section presents a set of micro-benchmarks that focus potential bottlenecks. The centralized design of Active Data in particular may be a limiting factor for scaling the system to big data workloads. For this reason, this first evaluation focuses on throughput (the number of common operations the system can handle each second) and response time (how fast the system responds to a common request). Another major concern of users is whether Active Data will slow their systems down when it reports transition publications in addition to normal operations. Last, we study how Active Data performs with a large data sets and a common benchmark, Hadoop TeraSort.

The benchmarks are based on the version 0.1.2 of the prototype, and client-service communications are implemented with Java RMI.

4.1.1 Experimental setup

Experiments in this section are run on the Griffon cluster of the Grid'5000 experimental testbed [158]. Grid'5000 is a large-scale testbed available to french researchers in all areas of computer science but with a focus on distributed and parallel computing, including cloud, HPC and big data. Griffon is part of the Nancy site; it is composed of 92 2-CPU nodes, each of which is an Intel Xeon L5420 with 4 cores running at 2.5Ghz. Each node is equipped with 16GB of RAM and a 320GB Sata II hard drive. Nodes are interconnected with Gigabit Ethernet and are running Linux 3.2.

All the benchmarks in this sections use the data life cycle model presented on Figure 4.1. The two transitions t_2 and t_3 arranged as a loop allow to publish as many transitions as required by benchmark scenarios.

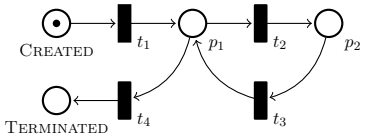


Figure 4.1: Data life cycle model used for the evaluations. It conveniently contains a loop allowing an unlimited number of transition publications.

4.1.2 Transition Publication Throughput

As a first metric, we want to measure how many queries the Active Data Service can serve under maximum stress. The query used in this benchmark is the publication of transitions because it is the most frequently used, and as such it is the most likely to create contention on the Service. Thus, throughput is measured as the number of transition publications Active Data is able to handle per second. In order to stress the system we run a single Active Data Service and a varying number of clients, each running on a dedicated node. Each client creates and publishes a life cycle for the model in Figure 4.1; then, they each publish 10,000 transitions for the life cycle, in a loop, without pausing. Each second, the Service counts how many publish query it processed.

Figure 4.2 plots the average number of transitions processed per second against the number of clients. There is a steady increase of the number of queries treated per second between 10 and 100 clients. The Active Data Service seems to reach its maximum capacity of 32,000 queries per second with 450 concurrent clients. Between 100 and 550 clients, the increase of the number of clients does not seem to have a negative impact on the throughput of the Active Data Service. This result can be explained by the fact that publishing a transition requires to lock several data structures to safely update a `LifeCycle` object on the Service, and then requires to create an event per subscribed client. For these reasons, publishing a transition is more expensive than pulling events and querying a life cycle.

This metric is satisfying considering the centralized nature of the prototype implementation of Active Data. While its design could certainly be greatly improved by a distributed implementation of the publish/subscribe layer, whether Active Data per-

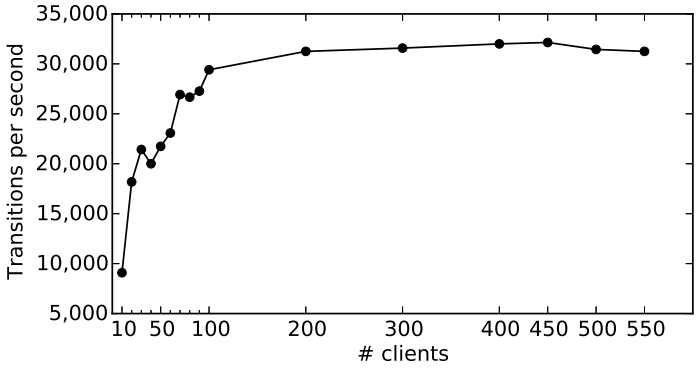


Figure 4.2: Average number of transitions handled by the Active Data Service per second with a varying number of clients. Each client publishes 10,000 transitions without pausing between iterations.

		med	90th centile	std dev
Response time	Local	0.77 ms	0.81 ms	18.68 ms
	Eth.	1.25 ms	1.45 ms	12.97 ms

		w/o AD	with AD
Overhead	Eth.	38.04 s	40.6 s (4.6%)

Table 4.1: Response time in milliseconds for life cycle creation and publication, transition publication and overhead measured using BitDew file transfers with and without Active Data.

formances are sufficient depends on the requirements of a particular application. A transition is published after some tasks was performed on a data item. In most systems, clients publish transitions at a pace much slower than in this benchmark, allowing several hundred thousands of clients to publish transitions every few seconds or minutes.

4.1.3 Response Time and Overhead

We evaluate the average response time, i.e the time it takes for the service to satisfy a client request. We pack two requests that are often performed together: create and publish a new life cycle and publish a transition on that life cycle. In the same spirit, we measure the overhead of publishing transitions on the client, i.e. the additional time incurred by the Active Data runtime environment. As we need an ordinary application to measure its overhead, we use the BitDew file transfer operation; this operation has the interesting feature of regularly publishing an "in progress" transition. The experiment consists in creating and uploading 1,000 1KB files in a single burst to stress the system. When Active Data is enabled, more than 6,000 transitions are published during the files' transfer. The execution time is recorded with and without Active Data.

Table 4.1 shows the median, 90th centile and standard deviation for the response time in milliseconds, when the service and the client run on the same machine (local) and on two different machines (Ether). The overhead is given in seconds and as a percentage compared with the vanilla BitDew. We observe that the response time is in the order of the millisecond, with a remarkable stability. The overhead, even when the system is highly stressed, remains less than 5%.

All together, these experiments demonstrate that the prototype implementation performs well enough to provide reactivity and scalability, and is fully able to handle the case studies presented in the next section as well as complex real-life experiments such as the one presented in the next chapter.

4.1.4 Hadoop Benchmarks

To complete the performance evaluation presented in the last subsection, we evaluate Active Data's performances against a real life workload. To this end we use the life cycle model of files in HDFS and Hadoop and run an intensive sorting benchmark on a 1TB data set. Active Data must be able to handle the many transition publications induced by the map and reduce tasks.

We execute the Terasort benchmark on the Suno cluster located at the Sophia site of Grid'5000. Suno has 45 nodes, each having two 4-core Intel Xeon E5520 CPUs running at 2.26GHz, 32GB of memory and two 300GB hard drives. The nodes are

interconnected with gigabyte ethernet. We run the Active Data Service alone on one node and we setup Hadoop to run a mapper or reducer on every core of the remaining nodes. Thus on this setup we run 280 mappers and we decide to run 70 reducers.

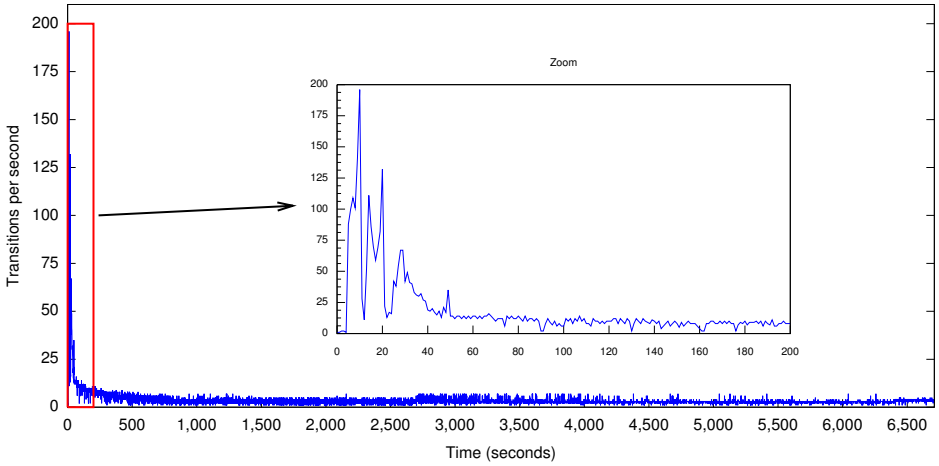


Figure 4.3: Number of transitions published each second during the Hadoop Terasort execution.

The Terasort benchmark runs in two phases: a random data set is generated by a first job called Teragen that we do not monitor, then the actual Terasort job is executed to sort it. The Active Data Service records how many transition publications it performs every second, like in the last subsection. Figure 4.3 plots the resulting data; after a peak during the first few seconds of the job, the number of transitions per second is relatively low as map and reduce tasks progress. The first peak is due to the many *Submit map*, *Submit reduce*, *Assign map* and *Assign reduce* transitions that are published in a burst when the job tracker starts the job and partitions the workload. During this first peak, a maximum of 196 transitions per second is recorded. After this, transitions are published at a much slower pace of 3 per second on average with a standard deviation of only 5, as map and reduce tasks complete.

This benchmark confirms our intuition that the 32,000 transitions per seconds limit gives enough margin to handle real-life applications like sorting 1TB of data with Hadoop.

4.2 Scenarios

We conduct four case studies to evaluate the ability of the Active Data programming model to deal with complex data management scenarios. These case studies present problems that we express in terms of transitions in the life cycle. Further, we define four criteria for evaluating the fitness of Active Data for common application requirements.

- Active Data allows to *write distributed applications based on data life cycle transitions*. In the first example, we show how to implement a storage cache between an application and the remote Cloud storage Amazon S3. We present the cache policy and describe its implementation based on transitions published during file transfers. This experiment shows that a simple cache, programmed in few dozen

lines of codes can effectively both improve performances and decrease Cloud usage costs.

- Active Data can *model data life cycle in existing systems and allows programmers to manage data sets distributed across systems or infrastructures.* To illustrate this, we present how to create a life cycle model for the Linux Kernel extension `inotify` which notifies applications of local filesystem modifications. Active Data allows applications to receive inotify notifications through its unique API and to integrate them in the whole application's data life cycle model. We present a case study fairly usual in big data science, where a set of sensors coordinates to implement data acquisition throttling, pre-processing to reduce the data size and archiving to a remote storage site. This scenario implies coordination between a set of local storage; a scenario difficult to achieve using ad-hoc scripting solutions.

- Active Data *allows to react to dynamic data change, such as a data set that dynamically grows, shrinks or gets partly modified.* We show that Active Data can optimize systems that do not fully take into account the life cycle of data. In the third use case, we present an incremental MapReduce that leverages the Active Data model to handle dynamic data. We modify an existing MapReduce implementation [159] so that it incrementally updates the result of a MapReduce job when a subset of the input data is modified.

- Active Data can *expose to the programmer a single data life cycle, even if data items are managed by several heterogeneous systems.* The last scenario presents the construction of a unified life cycle model based on the composition of two different systems: iRods and the Globus data transfer service. Thanks to this model and its ability to reconciliate data identifiers, we present an application which automatically keeps track of data items provenance when they move from one system to the other.

We evaluate the uses-cases against four criteria that are representative of the systems and application class we support:

- The unique high-level view of data life cycles offered by Active Data must allow users to implement cross-system optimizations;

- The global namespace maintained by Active Data must allow to integrate in the same scope data objects from several non-cooperative systems;

- The programming model must allow to program distributed data life cycle management tasks easily, benefiting from implicit parallelism;

- The event driven model must allow to program applications able to react to changing data.

4.2.1 Storage Cache

This scenario demonstrates the ability and the easiness to program distributed applications with Active Data. To this end, we study the case of implementing a storage cache between a computing infrastructure and a storage backend in terms of data life cycle transitions. Storage caches are widely used by scientific applications to minimize cost, network bandwidth, latency and energy consumption.

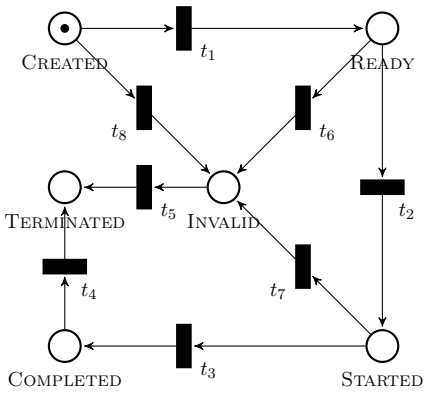


Figure 4.4: Life cycle model of BitDew's file transfer module. Subscribing transition handlers on some transitions allows us to dynamically manipulate file transfers in BitDew and implement a write-through cache.

This scenario involves BitDew, which was introduced in Section 3.4.4. BitDew offers a programmable environment and a set of services including the "data repository", the "data scheduler" and the "data transfer" services. The data scheduler performs data placement on a set of nodes according to high-level constraints; the data transfer service offers a unique asynchronous transfer API for various transfer protocols such as HTTP, FTP, GridFTP and Bittorrent.

In this scenario, we consider a cache between a computer infrastructure and the Amazon Simple Storage Service [160] (S3). S3 users pay according to the storage space they use, the number of put and get requests they perform and the amount of data they transfer from and to the S3 storage. Caching S3 avoids unnecessary data transfers to and from S3 which both improves the performances of applications accessing S3 data and decreases the S3 usage cost. To function properly, the cache application has to determine when data are present or not in the cache and perform the necessary file transfers accordingly. In terms of data life cycle this translates in reacting to file transfer events, i.e. when a file transfer starts or ends. Our implementation relies on the life cycle model of BitDew data transfer service. The life cycle model, connected to Active Data by instrumenting the code of the BitDew file transfer model is presented on Figure 4.4.

The now Active Data-enabled BitDew takes the place of the ordinary BitDew in the application, acting as main memory. Clients are connected to the cache application that runs on a local server node and that uses a fixed portion of its local storage to cache remote files. The cache is also a client of the Amazon S3 platform. Because we assume that the cache can possibly fail, we implement a write-through cache policy; this policy allows to have a durable copy of each file written to the cache, while allowing to save bandwidth on read operations. We express the cache application with only two transitions of the BitDew transfer life cycle model:

- t_1 (a transfer begins) is subscribed by the cache. The handler examines the transfer object that corresponds to the token; if it detects that the transfer is a *get* from the cache, it checks if the file corresponding to the data item is in the cache. If it is (*cache hit*), the handler serves it from the cache; if the file is not in the cache (*cache miss*), the handler downloads it from Amazon S3 and then serves the file from the cache.

	w cache	w/o cache	Difference
In	2350 MB	2350 MB	0 MB
Out	0.15 MB	1976.17 MB	1976.02 MB
#Put	13	13	0
#Get	0	20	20
Cost in USD	0.3	0.53	0.23

Table 4.2: Cache experiment evaluation results: using the simple write-through cache saved 1.9 GB of transfer and cut down the cost of Amazon S3 by 43%.

- t_9 (a transfer ends) is subscribed by the cache as well. If the handler detects that the token corresponds to a put in the cache, it transfers the same data item to Amazon S3 (hence the "write-through" policy); local file can be deleted from the cache according to various eviction policies.

We evaluate the cache with a scenario which mimics a master/worker computation that is fairly common to scientific applications. The scenario involves 10 clients, a 5GB cache server and Amazon S3. The master first transfers three files to the Amazon S3: a 200MB executable to be run by all client nodes and 2 input data-sets of 50MB and 100MB. Once the files are available in Amazon S3 for the clients, each client downloads the program. Half of the clients download the smallest data set while the others download the largest. Table 4.2 shows that during this short experiment, using the storage cache avoided performing 1.9 GB of unnecessary data transfers from Amazon S3, cutting the cost of the service by 43%.

This scenario illustrates the ability to rapidly prototype data management applications with Active Data: the source code is less than 100 lines of code and can be developed in a day. The resulting cache can benefit any application that uses BitDew with no code modification. It is distributed and yet requires no synchronization, no thread spawning and no forking.

4.2.2 Collaborative Sensor Network

This case study illustrates: *i*) the adaptability to legacy data management systems, *ii*) the ability to develop distributed applications that support independent data life cycles distributed over several local systems and *iii*) how easy it is to implement coordination between distributed nodes with Active Data.

It is a common practice for applications acquiring data—for example from a scientific instrument or a sensor network—to apply some pre-processing before pushing it to a computing platform, and archived. Pre-processing can be used to filter, compress data or remove invalid data. Such a sequence of operations can easily be—and is often—scripted using ad-hoc languages or programs. Data throttling is also a common practice to reduce the amount of data injected in a system at a given time. Decentralized data throttling enables to reduce the load on the system by dropping data before they are injected. However, it requires coordination between multiple sensors, which can be tricky in systems composed of many nodes distributed on multiple infrastructures. This scenario demonstrate that decentralized data throttling can be achieved simply when expressed with Active Data.

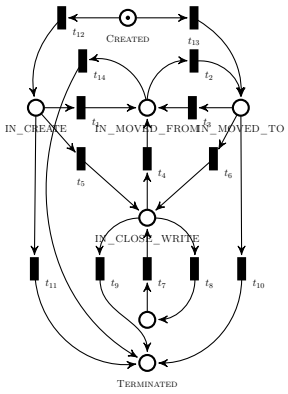


Figure 4.5: Data life cycle model for inotify. The life cycle model represents common operations that can be performed on files and reported by inotify.

We consider a system where large high-resolution images are acquired from a network of cameras, each connected to its own pre-processing node. Images are regularly written on these nodes' filesystems in the TIFF format. The images are large, so each node must independently perform some pre-processing to compress them in the JPEG format. The resulting JPEG files are then transferred to a distributed storage system where they will be available for further processing. In addition to this, we want the nodes to perform decentralized data throttling: they must drop TIFF images received from their camera if the global number of images pre-processed p during a defined time window w in seconds reaches a threshold n.

As soon as a camera writes an image file on a node's filesystem, it is considered as a newly created data item and its life cycle begins. To capture life cycle transitions on files, we use the inotify Linux kernel subsystem. Inotify allows to *watch* a directory and receive events about the files it contains. Events regard file creations, modifications, writes, movements and deletions. As inotify events represent filesystem events, and filesystems contain data (files) that are subject to transitions, we can represent the life cycle model of files from the perspective of inotify. Figure 4.5 presents the inotify data life cycle model, constructed from the official documentation. The combination of Active Data and inotify creates a *distributed inotify*: all nodes can now coordinate based on transitions happening on other nodes' filesystems. Nodes locally run a small daemon program that reads inotify events from their Linux kernel and publishes the corresponding life cycle transition to Active Data.

With sensor nodes able to react to remote filesystem transitions, we can express our problem in terms of life cycle transitions. Each node independently runs a program that subscribes handlers to two inotify transitions:

- t_{12}: the handler checks if the transition is local or remote: if it is remote and if the associated file is a JPEG image, then a TIFF image has been pre-processed on a remote sensor and the handler increments its local counter p.
- t_5: if the transition is local, the handler checks the associated file type: if it is TIFF, the handler pre-process the file only if $p < n$; otherwise, the file is removed.

Thus the way the algorithm work is by having on each node a handler maintaining p, the number of images converted in the current time period. Nodes have only a local copy of the variable p; by reacting to image conversions happening on other nodes, they try to keep their value p up to date. This is obviously a best-effort situation where

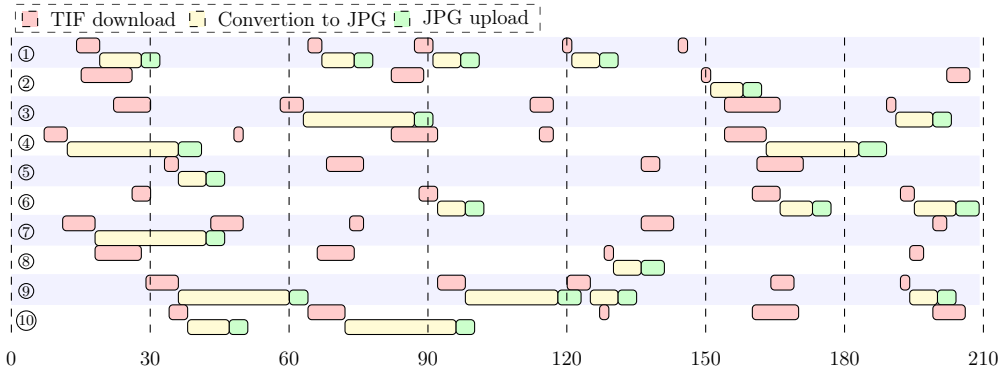


Figure 4.6: Collaborative network of 10 sensors: the x axis plots the time in seconds, for a window size $w = 30$ seconds.

the "pull frequency" discussed in Subsection 3.3.2 has an effect on the accuracy of p: reducing significantly the pull frequency will allow each local variable p to be closer to the actual number of images converted in the current time period. In this particular application, we decide that a small error is acceptable and will not prevent the data throttling to fulfill its mission. In addition, each node sets p to 0 every w seconds in order to start a new time period.

We implement and evaluate a simple scenario with 10 machines, each randomly downloading 5 TIFF images (between 121MB and 502MB) in a watched directory. We implement and configure the transition handlers for $n = 3$ and $w = 30$ *seconds*. The pull frequency is set to 1,000 ms: the Active Data client queries the service for events every second.

Figure 4.6 presents the Gantt chart of the scenario which lasts 279 seconds; each numbered pair of lines represent the activity of one sensor. Red bars plot data acquisition times, yellow bars plot data pre-processing and green bars plot the upload time of pre-processed files. On each sensor, data acquisition and pre-processing and upload are effectively performed in parallel. We see that the system behaves as expected: for example in the time window [60,90], 8 new JPEG images are downloaded on the nodes, but only 3 are pre-processed; the other images are dropped.

This scenario illustrates a powerful feature: Active Data can easily turn into a distributed system, any local system that is able to expose its local data life cycle. It also demonstrates the ease with which distributed data life cycle management tasks can be expressed with Active Data.

4.2.3 Incremental MapReduce

In this case study, we investigate how an existing system can be optimized by leveraging on Active Data's ability to cope with dynamic data.

One of the strongest limitations of the MapReduce programming model is its inefficiency to handle mutating data; when a MapReduce job is run several times and only a subset of its input data set has changed between two job executions, all map and reduce tasks must be run again. Making MapReduce incremental, i.e. re-run map and reduce tasks only for the data input chunks that have changed, necessitates to modify the complex data flow of MapReduce. However, if a classical MapReduce framework becomes *aware* of the life cycle of the data involved, it may be able to dynamically

adapt its computation to data modification.

We consider the MapReduce implementation built on top of the Active Date-enabled BitDew storage system [159]. In this implementation, the BitDew instance has 3 types of clients: a master node that places input data chunks in the BitDew storage and launches a MapReduce execution and; mappers and reducers that execute map and reduce tasks respectively. However, input data can be updated directly in the storage by external applications. To make this MapReduce implementation incremental, we simply add a "dirty" tag to the tokens representing certain data chunks. When a chunk's token is tagged as dirty, the mapper that previously mapped the chunk executes again the map task on the new chunk content and sends the updated intermediate results to the reducers. Otherwise, the mapper returns the intermediate data previously memoized. Reducers proceed as usual to compute again the final result. To update the chunk dirty state, we need the master and the mappers to react to transitions in the life cycle of the chunks. More precisely, nodes subscribe to two transitions published by BitDew transfers with the life cycle model from Figure 4.4:

- t_3 is subscribed by the master node. After this transition is published, the handler on the master node checks whether the transfer is local and whether it modifies an input chunk. Such case happens when the master puts all the data chunks in BitDew's storage system before launching the job. If both conditions are true, the transition handler tags the corresponding token as dirty;
- t_1 is subscribed by all the mappers. When this transition is published, handlers on the mappers check whether the transfer is distant and is overwriting one of their chunks. In this case, the transition handler on the mapper marks the token as dirty.

In addition, the mapper implementation is modified to run or not to run the map method according to the presence of the dirty tag on the chunk's token.

To evaluate the performance of the incremental MapReduce, we compare the time to process the full data set with the time to update the result after modifying a part of the data set. The experiment is configured as follows; the benchmark is the word count application running with 10 mappers and 5 reducers; the data set is 3.2 GB split in 200 chunks. Table 4.3 presents the time to update the result with respect to the original computation time when a varying fraction of the data set is modified. As expected, the less the data set is modified, the less time it takes to update the result: it takes 27% of the original computation time to update the result when 20% of the data chunks are modified. However, there is an overhead due to the fact that the shuffle and the reduce phase are fully executed in our implementation. In addition, the modified chunks are not evenly distributed amongst the nodes, which provokes a load imbalance. Further optimizations would possibly decrease the overhead but would require significant modification of the MapReduce runtime. However, thanks to Active Data, we demonstrate that we can reach significant speedup with a patch that impacts less than 2% of the BitDew MapReduce runtime source code.

4.2.4 Data Provenance

We have discussed in Chapter 2 how reconstructing provenance is difficult in the general case; it is in fact even more challenging in applications that involve multiple non-collaborative systems. Some systems may be *provenance-aware*, others may not. In such

Fraction modified	20%	40%	60%	80%
Update time	27%	49%	71%	94%

Table 4.3: Incremental MapReduce: time to update the result compared with the fraction of the data set modified.

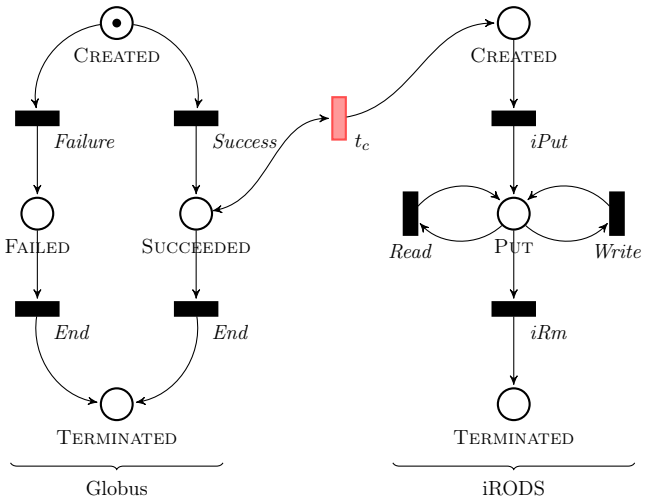


Figure 4.7: Life cycle models of transfers in Globus and files in iRODS. The two models are connected with a composition transition; the composition transition may be published only if the Globus transfer is successful.

cases, only an entity with a exhaustive view of the whole application can reconstruct the provenance of generated data. Active Data's unique high-level and end-to-end view of data life cycles offers the exhaustive view we need. Thanks to Active Data's ability to react to all life cycle transitions, users are able to reconstruct provenance from the sequence of operations applied to data sets across loosely coupled systems distributed on any computing infrastructure.

Here we consider a scenario where Active Data receives events regarding data that evolve in two completely independent systems. To mimic cooperation of data management systems within data centric infrastructures, our scenario features one service to handle file transfers (Globus Online) and one software to store data and provide a metadata catalog (iRODS).

In our scenario, for any data file in iRODS, we want to record file transfer provenance: the transfer endpoints, start and completion times and possible transfer failures. This scenario illustrates two difficulties; the first difficulty to access all copies and all metadata of a single data element distributed in several systems, with several identifiers; the second difficulty is that Globus Online is a SaaS and we access it remotely; we have no access to its source code and we are not aware of the full internal data life cycle it creates.

Active Data is the glue that enables both Globus and iRODS to see the part of data life cycles that is outside their scope. We use iRODS's user-defined metadata to record provenance information along with data files.

Figure 4.7 represents the life cycles of data in Globus and iRODS. When a Globus file transfer starts, Globus creates a transfer task and returns its `Task Id`. The user creates

a data life cycle with this task id and waits for the transfer to complete. When a transfer completes, Globus sends a notification back to the user, containing "SUCCEEDED" if the transfer was successful or "FAILED" otherwise. Depending on this email notification, either the transition t_1 or t_2 is published to reflect the state of the transfer. A token on iRODS's CREATED place is the equivalent of a "touch": an empty file is created and a data structure is allocated in iRODS backend database. The *iPut* transition is named after an iRODS command that stores a file in iRODS data repository. More `iput` operations trigger the *Write* transition and `iget` commands trigger the *Read* transition. *iRm* is the transition that corresponds to the iRODS command for removing a file.

To compose the two life cycles, the place SUCCEEDED from Globus is a start place which creates a token in the iRODS life cycle model. The reception of a "success" email notification causes transition t_1 to be triggered, and a handler to store the file in iRODS. iRODS returns an identifier called `DATA_ID` that is added to the token; it now contains both identifiers.

A second transition handler is attached to iRODS's creation transition: it is executed when any iRODS data is created. This handler requests the life cycle from the Active Data Service to see if it contains a Globus identifier. In such case, it queries the Globus REST API to get file transfer information.

To demonstrate our solution, file transfers are launched from a remote Globus endpoint to a local temporary storage every few seconds. We observe that when a transfer ends, it appears immediately in the iRODS data catalog with the correct Globus `Task Id` and meta-data information (endpoint, completion date, request time). Listing 4.1 shows the metadata set for one of these iRODS files after the transfer is done.

```
$ imeta ls -d test/out_test_4628
AVUs defined for dataObj test/out_test_4628:
attribute: GO_FAULTS
value: 0
----
attribute: GO_COMPLETION_TIME
value: 2013-03-21 19:28:41Z
----
attribute: GO_REQUEST_TIME
value: 2013-03-21 19:28:17Z
----
attribute: GO_TASK_ID
value: 7b9e02c4-925d-11e2-97ce-123139404f2e
----
attribute: GO_SOURCE
value: go#ep1/~/test
----
attribute: GO_DESTINATION
value: asimonet#fraise/~/out_test_4628
```

Listing 4.1: Metadata associated to an iRODS data file transferred with Globus

The global and unique namespace provided by Active Data of data sets over heterogeneous and non-cooperative systems significantly simplifies the challenge of global provenance reconstruction. In addition, we have demonstrated that systems can be extended to do more, thanks to the information from outside their scope provided by Active Data.

4.3 Conclusions

This evaluation brings four main contributions to Active Data's prototype implementation. First, it demonstrates that despite its early development stage, Active Data can support the workloads that are found in common distributed scientific applications. Second, the use-cases show that the programming model is expressive enough to program common data management tasks and optimizations. Third, they demonstrate how Active Data can optimize existing systems by extending their scope, with minimal change. Fourth, the use-cases can be used by users as examples of how to start programming with Active Data.

5 A Framework for Data Surveillance Based on Active Data

The *Advanced Photon Source* (APS) is a research project that includes several large-scale material physics experiments. They generate and handle very large data sets, and involve several systems and software. This Chapter describes the development of a *data surveillance framework* for a particular experiment; the framework offers a unique combination of features: ability to monitor heterogeneous systems and distributed infrastructures, user notifications through plugins (e.g., Twitter), automated data tagging and ability to convey tags across systems, data filtering and rule-based programming. We demonstrate the benefit of this data surveillance system by implementing a prototype based on the Active Data.

5.1 Background

We study a particular e-Science experiment conducted at the Advanced Photon Source [161], a research facility featuring a synchrotron-radiation light source for material physics, biological science, environmental, geophysical and planetary science. It is located at Argonne National Lab, in Illinois. The application we study is part of a materials science experiment that generates up to 1TB of data per day that must be moved, cataloged, and analyzed to satisfy user needs.

5.1.1 The APS Experiment

The materials science experiment that we consider is conducted at an APS beamline that is used to analyze different sample materials using synchrotron x-rays. Scientists apply techniques such as high-energy diffraction microscopy (HEDM) and combined high-energy small and wide-angle x-ray scattering (HE-SAXS/WAXS) to characterize different samples. The end-to-end process of the particular experiment we consider, shown in Figure 5.1, is both compute and data intensive, using thousands of cores to enable near-real-time analysis of data as it is acquired. This rapid analysis permits immediate feedback so that experiment parameters can be adjusted immediately. Experiments at the beamline currently generate 3-5TB of data per week.

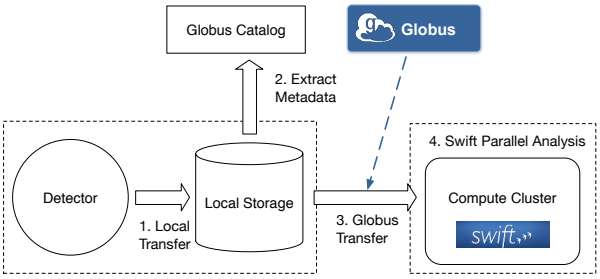


Figure 5.1: The Advanced Photon Source experiment

Data is obtained from a beamline detector with a direct connection to an acquisition machine. This acquisition machine runs proprietary detector software and contains modest storage and compute resources. As data is acquired, it is moved from the acquisition machine to a larger shared cluster with greater storage capacity. While this step is currently performed out of band, in the near future it will be replaced with automated Globus transfers.

After transfer to the shared cluster, data is processed with several data reduction and aggregation scripts (e.g., refinement operations and calculation of stresses and strains for individual grains). As there may be many files related to a particular experiment or sample within that experiment, files are grouped into data sets. The data is then cataloged in the Globus Catalog. Here automated python-based parsers are used to extract metadata, for example related to the user, experiment, and sample. Knowledge of the Nexus HDF format is used to extract structured metadata from the file. The catalog associates various different raw and derived data, scripts, and metadata into self-contained data sets.

Following cataloging, data is moved to large scale compute resources. Here a parallel Swift-based analysis pipeline is run to fit a crystal structure to the observed image. This processes is computationally intensive and involves iterative processing of many rows using a C function on each parameter and a reduction phase to merge the results into a single output. Throughout the analysis provenance information is recorded in the Catalog and associated with the sample data set.

5.1.2 Workflow Tools

While not a complete list of tools used in this process, the following are the core tools that act upon data.

Globus Online provides high performance, secure, third party data transfer and synchronization. Operated as Software-as-a-Service, Globus Online enables researchers to manage large data transfers between "endpoints" via a web interface or REST API. Globus Online handles all the difficult aspects of data transfer allowing a user to start an asynchronous data transfer, while tuning parameters to maximize bandwidth usage, managing security configurations, providing automatic fault recovery, and notifying users of completion and errors. Globus Online also allows researchers to share large data sets from their usual storage repositories.

Globus Catalog is a service that enables the creation and management of user-defined "catalogs" that contain data sets and references to associated data and metadata. Within a catalog, users can create data sets, associate data members (files and directories), and specify user-defined metadata in the form of key-value annotations. Data sets are logical collections of, potentially distributed data. Globus Catalog defines a

schemaless metadata model in which arbitrary metadata can be attached to data sets or data members. Globus Catalog's query interfaces allow users to discover and retrieve data sets based on their annotations.

Swift is a parallel scripting language designed for composing applications into workflows that can be executed in parallel on multicore processors, clusters, grids, clouds and supercomputers. Swift focuses on orchestrating independent and distributed tasks across distributed computing systems. Swift uses a low level C-like scripting model. Swift is implicitly parallel: users do not explicitly fork and join processes; they do not program their workflows to be parallel or to handle synchronization, file transfer, and do not explicitly chose execution locations.

5.2 Objectives

While the APS use case currently satisfies the needs of its users, there is potential for significant inefficiency, unreported failures and even errors due to the complexity of dealing with several terabytes of data and a number of different tools and systems. Here we present four useful features desired by scientists. These features appear simple at small scales and when executed on a single machine, however with large distributed data sets they present significant challenges to users.

5.2.1 Progress Monitoring

Mechanisms are required to monitor the entire workflow from a high level, generate reports on progress, and identify potential errors without examining every data set and log file. Monitoring is not limited to estimating completion time, but also: *i*) receiving a single relevant notification when several related events occurred in different systems; *ii*) quickly noticing that an operation failed within the mass of operations that completed normally; *iii*) identifying steps that take longer to run than usual, backtracking the chain of causality, fixing the problem at runtime and optimizing the workflow for future executions; *iv*) accelerating data sharing with the community by pushing notifications to collaborators.

5.2.2 Automation

The APS experiment, like many scientific experiments and workflows, requires explicit human interventions to make data progress between stages and to recover from unexpected events. Such interventions include running scripts on generated data sets on the shared cluster, registering data sets in the Globus catalog, and executing Swift analysis scripts on the compute cluster. Such interventions cannot be easily integrated in a traditional workflow system, because they reside a level of abstraction above the workflow system. In fact, they are the operations that *start* the workflow systems. Because the code to automate them would need to have a high-level view similar to that of the human operator, they are performed by a human operator.

5.2.3 Sharing and Notification

As a result of analysis, numerous files are produced and registered in the Globus catalog, many of which may be valuable to the community. Sharing these files can be made more efficient by allowing other scientists to be notified of events (e.g., new data sets

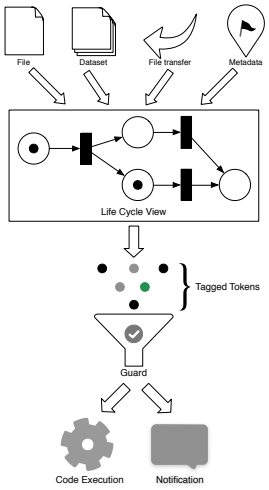


Figure 5.2: Data surveillance framework design

available in the catalog) with powerful filters to extract only the notifications they need, and even to start processes as soon as files are available. We believe the best way for scientists to automatically integrate new data sets in their workflows is to rely on widely used dissemination mechanisms—such as Twitter. Twitter is an efficient asynchronous notification mechanism that is commonly used by scientists. It also can be simply integrated via its APIs that enable straightforward integration with external systems.

5.2.4 Error Discovery and Recovery

Each system participating in the APS experiment has only a partial picture of the entire process, which impairs the ability to recover from unexpected events. Thus, when such events occur, systems often fail ungracefully, leaving the scientists as the only one able to resolve the problem through costly manipulations. For example, if a Swift script fails when processing a file, it simply reports the error in a log file and returns. Later, when inspecting output files, a scientist may see that some results are missing; they will read the log and discover the faulty file; a quick look at the file will tell them it was corrupted and their only measure will be to again transfer the file to the computing cluster and re-start the analysis. This requirement for low level human intervention and manipulation delays experiment completion, places a burden on scientists and wastes valuable computing resources. Moreover, such approaches are also prone to human error. Automating these steps would improve each of these aspects.

5.3 System Design

This section presents the data surveillance framework designed to satisfy the APS users' needs presented above.

5.3.1 Data Surveillance Framework

Figure 5.2 shows the main features of the data surveillance framework; the framework is able to track any data object (such as files and data sets) as well as elements related to data (such as file transfers and metadata). The framework receives events from different systems and integrates the identifiers of data and related elements in a single

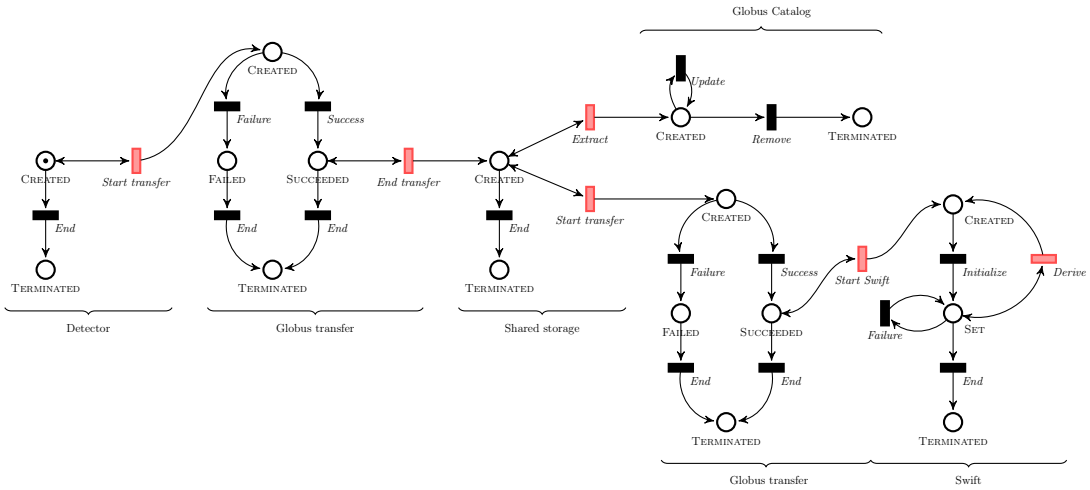


Figure 5.3: The life cycle model for the APS experiment comprises three main systems (Globus transfers, Globus Catalog, and Swift) as well as intermediate storage. Data and information transfers between systems are represented with red composition transitions.

global namespace. Data replicas and their current state are exposed as tokens in a unified namespace used across systems. Tokens provide a model for linking together related data in the different systems. Arbitrary tags can be associated with tokens to add additional information. Tokens can be tagged automatically by the framework, or manually added by users. Tags are used to pass information amongst the different systems, and to implement conditional behaviors based on the current state of data. From the user's perspective, the "state of data" is both the distributed state of all the replicas of a data item, and additional state information attached as tags. The framework allows users to be notified of the progress of their data and to automatically run custom code at many operational stages of the life cycle. Additionally, tags are used to filter tokens and trigger notifications and code execution on a subset of events, according to user choice.

5.3.2 Active Data

The complexity of implementing the surveillance framework comes from a lack of *integration*, impairing tight coordination between loosely coupled systems. Such coordination is challenging because, in order to be efficient, it must be as noninvasive as possible. Additional challenges stem from the lack of feedback derived from the systems, that are mostly regarded as black boxes from the user perspective.

We use Active Data's life cycle model to individually represent and expose the internal life cycle of the three APS systems: Globus Online, Globus Catalog and Swift. Then, we use Active Data's composition ability to represent how data moves between them.

5.3.3 APS Experiment Life Cycle Model

We now present the life cycle model of data in the APS experiment. We first represent each system independently, and then compose the system models together to model the whole experiment.

The complete model, shown in Figure 5.3, is divided into six parts, separated by composition transitions. On the left, we represent the detector with a minimal life cycle, that is a CREATED and a TERMINATED place, with a unique transition between them; the detector's CREATED place is connected to the life cycle of a Globus transfer with a composition transition that represents the moment when Active Data records the identifier in the destination system (Globus Online) and maps it to the identifier in the source system (the detector).

In the Globus Online model, a token represents a transfer task containing a directory tree with several files. A transfer task can succeed or fail. In the case of success only, a new composition transition can create a token in the shared storage.

The shared storage is also a minimal life cycle. From there, two things happen: a Python script registers each directory in the completed transfer as a data set in the Globus Catalog and annotates it with metadata extracted from the files; another Globus transfer copies each data set to a computing platform for analysis. The order in which these two stages happen does not matter in the model; the order in which both transitions are triggered simply indicates which happened first.

On the Globus Catalog model, metadata can be added to a data set one by one, and all metadata can be removed at once. On the Globus transfer model, a transfer can succeed or fail. If the transfer succeeds, the *Start Swift* composition transition maps the Globus task identifier to the identifier (path) of each file contained in the data set. These files are used as input to the Swift script, that will in turn produce an output file. Swift features a *self composition*: each input file is naturally linked to the output file it produced by the *Derive* transition. An error in the Swift script results in the output file not being created and the *Failure* transition being triggered.

5.3.4 Event Capture

Having represented the entire model with Active Data, we now describe how information from an execution of the APS experiment is mapped to the model. This translates to creating a token in the model for every file, transfer and data set manipulated, and then publishing the right life cycle transition to Active Data.

To collect runtime information and translate it to life cycle transitions, we use some of the techniques presented in Subsection 3.3.6.

1) *Detector*: as the first transfer from the detector is started manually, the publication of the *Start transfer* transition is also done manually, with an Active Data command line tool.

2) *Globus transfers*: we developed a simple Java program that uses a Globus REST API query for completed transfers every 30 seconds, examines the transfer status (success or failure), and publishes the corresponding transition.

3) *Shared storage*: on the shared storage, every action is currently performed manually. To publish the transition *Shared storage.Extract*, we modified the Python extraction script to call the Active Data command line tool before exiting. The Globus transfer is started manually, so the publication of the second Globus transfer is also performed manually from the command line.

4) *Globus Catalog*: similar to the REST API for transfers, the Catalog allows information about all data sets for a given user to be retrieved. In this case we developed a Python script that queries the last modification time of each data set, and publishes the *Globus Catalog.Update* transition for data sets that have been modified since their

```
Tagger t = new Tagger() {
   public void tag(Transition transition, Token[] inTokens, Token[] outTokens, ↘
       →Token[] newTokens) {
      String filename = newTokens[0].getUid();

      if(filename.endsWith(".hdf")
         newTokens[0].addTag("hdf");
   }
};
```

Listing 5.1: Example of tagger attached to transition *Shared storage.Start transfer*.

last recorded modification time.

5) *Swift*: while running, Swift writes information about intermediary input and output files to a log file. By parsing the log file while Swift is running with a Python script, we are able to publish the *Swift.Error* transition when a task fails, and to publish the *Swift.Derive* transition when a task produces an output file from an input file.

Following the principles of Active Data, the methods for instrumenting the three systems—Globus transfers, Globus Catalog and Swift—are completely non-intrusive and reusable. The individual life cycle models and the scripts we have developed can be used independently from the rest of this work by scientists wanting to make their tools and workflows "Active Data-enabled."

5.3.5 Tagging

To make tokens more useful to users, we attach relevant information to them using Active Data's tag features. We mostly employ taggers for their advantage of tagging tokens on the Active Data Service, without requiring user intervention. This guarantees that tokens that must be tagged cannot be missed by potential user-side defects.

Table 5.1 describes the tags used in the APS experiment. Listing 5.1 shows an example of a tagger attached to transition *Shared storage.Start transfer*. Here the tagger examines the identifier of the token created by the composition transition (the only element in `newTokens`) and adds a tag to the token according to the file extension (the actual code considers more file types). Despite this tag being added in the shared storage, it will be propagated to subsequent systems (e.g., Swift).

Transition	Tags
Detector.Start transfer	Detector name
Globus transfer.End transfer	Data set name
Shared storage.Start transfer	File type
Globus transfer.Start Swift	"swift-input"
Swift.Initialize	Program name
Swift.Derive	"swift-output"

Table 5.1: APS life cycle transitions and corresponding tags.

5.4 Results

To evaluate our data surveillance framework we demonstrate the ease by which the four desired end user features described in section 5.2 (monitoring, automation, notification,

and error recovery) can be provided using Active Data.

5.4.1 Monitoring Progress

We demonstrate the ability to monitor progress by observing all systems in the experiment at the same time. To this end, a line of text is recorded in a file every time a data set received from the detector is fully processed by Swift, i.e. passes through the entire experiment workflow. In order to generate this line of text correctly, the writer needs to know how many files are contained in each data set, what data set a file belongs to, and how many files from each data set Swift has processed. This capability is achieved by running a single transition handler when files are transferred from the detector (transition *Detector.Start transfer*) and when Swift derives an output file from an input file (transition *Swift.Derive*).

```
ActiveDataClient ad = ActiveDataClient.getInstance();

TransitionHandler handler = new TransitionHandler() {
   private Map<String, int> datasetSize;
   private Map<String, int> datasetTreated;

   public void handler(Transition t, boolean isLocal, Token[] inTokens, ↘
       →Token[] outTokens) {
      String datasetId = "";

      // Count total number of files in the data set
      if(t.equals("Detector.Start transfer")) {
         datasetId = inTokens[0].getUid();
         int size = new File(datasetId).listFiles().length;
         datasetSize.set(datasetId, size);
      }

      // Count number of treated files in the data set so far
      if(t.equals("Swift.Derive")) {
         LifeCycle lc = ad.getLifeCycle(inTokens[0]);
         datasetId = lc.getTokens("Detector.Created")[0].getUid();
         datasetTreated.set(datasetId, datasetTreated.get(datasetId) + 1);
      }

      // If done, update file
      if(datasetSize.get(datasetId) == datasetTreated.get(datasetId)) {
         FileOutputStream out = new FileOutputStream("~/aps.log");
         out.write("Done: " + datasetId);
         out.close();
      }
   }
};

ad.subscribeTo("Detector.Start transfer", handler);
ad.subscribeTo("Swift.Derive", handler);
```

Listing 5.2: Monitoring the progress of the experiment life cycle: a log file is automatically updated each time a data set has been entirely processed by a Swift script.

The handler code is presented in Listing 5.2. The code keeps two counters for each data set, recording how many files it contains, and how many have been processed. If the event that triggered the execution was a data set leaving the detector, then handler stores how many files it contains. If the execution was triggered because Swift finished processing a file, it updates the counter of processed files for the data set, despite the fact that Swift has no knowledge of which data set the file originates from. To learn this

```
ActiveDataClient ad = ActiveDataClient.getInstance();

TransitionHandler handler = new TransitionHandler() {
   public void handler(Transition t, boolean isLocal, Token[] inTokens, ↘
       →Token[] outTokens) {
      Token dataset = outTokens[0];
      String path = "~/aps/incoming/" + dataset.getUid();
      Runtime r = Runtime.getRuntime();
      Process p = r.exec("catalog_loader.py " + path);
      p.waitFor();
   }
};

ad.subscribeTo("Globus Transfer.End Transfer", handler);
```

Listing 5.3: Transition handler that automatically launches metadata extraction when new files arrive on the shared storage.

information, the handler queries the Active Data API to discover the data set identifier. Finally, if the last file in the data set has been processed, the log file is updated.

5.4.2 Automation

We now demonstrate how tasks that were previously performed manually can be automated using our approach. In particular, this feature demonstrates how the Python metadata extraction script can be run automatically when a transfer to the shared storage completes. In addition, a specific Swift analysis script can be executed when the transfer of a HDF file to the compute cluster completes.

A first transition handler is set to run after transition *Globus Transfer.End transfer* is triggered; Listing 5.3 shows the corresponding code. This handler examines the token produced by the transition and get its identifier; then, it turns this identifier into a local path which is used as an argument to the Python extraction script.

The second task is implemented with a different transition handler that is set to run after transition *Globus transfer.Start Swift* is published (Listing 5.4). This code is similar to the previous example, except for the use of a transition guard. Because tokens have been tagged according to their corresponding file type, we can now filter out every token that does not link to a HDF file. The transition guard implements the predicate "HDF" $\in tags(inTokens[0])$, i.e. the first token consumed by the transition must have the tag "HDF".

5.4.3 Sharing and Notification

We now consider extending the notification capabilities of the system by advertising the availability of new data sets on Twitter. After metadata extraction, each data set is available on the Globus Catalog and annotated with metadata. In the model shown in Figure 5.3, this corresponds to transition *Shared storage.Extract*. Listing 5.5 shows the transition handler used to notify other users of the new data set. Because the token identifier in the Globus Catalog life cycle is the URL of the data set, the code can directly derive a unique link to include in the tweet. To provide additional information, every tag attached to the token is written as a "hashtag" in the tweet. The remainder of the code makes a REST call to the Twitter API to actually publish the tweet.

```
ActiveDataClient ad = ActiveDataClient.getInstance();

TransitionHandler handler = new TransitionHandler() {
   public void handler(Transition t, boolean isLocal, Token[] inTokens, ↘
       →Token[] outTokens) {
      // Start aps-hdf.swift
      Token file = outTokens[0];
      String path = "~/aps/hdf/input/" + file.getUid();
      Runtime r = Runtime.getRuntime();
      Process p = r.exec("swift aps-hdf.swift " + path);
      p.waitFor();
   }
};

HandlerGuard guard = new HandlerGuard() {
   public boolean accept(Transition transition, Token[] inTokens, Token[] ↘
       →outTokens) {
      return inTokens[0].hasTag("hdf");
   }
}

ad.subscribeTo("Globus Transfer.End Transfer", handler, guard);
```

Listing 5.4: Automatically launch Swift analysis when input files are available.

```
ActiveDataClient ad = ActiveDataClient.getInstance();

TransitionHandler handler = new TransitionHandler() {
   public void handler(Transition t, boolean isLocal, Token[] inTokens, ↘
       →Token[] outTokens) {
      // Construct the URL and the hashtags
      String url = "https://catalog.globus.org/dataset/" + ↘
          →outTokens[0].getUid();
      String hashTags = "";
      for(String tag: inToken[0].getTags())
         hashTags += " #" + tag;

      // Twitter API call
      String msg = "New data set available " + url + hashTags;
      Twitter twitter = new TwitterFactory().getInstance();
      Status status = twitter.updateStatus(msg);
   }
};

ad.subscribeTo("Globus Catalog.Extract", handler);
```

Listing 5.5: Automatically notify users of the availability of new data sets via Twitter.

```
ActiveDataClient ad = ActiveDataClient.getInstance();

TransitionHandler handler = new TransitionHandler() {
   public void handler(Transition t, boolean isLocal, Token[] inTokens, ↘
       →Token[] outTokens) {
      // Get the data set identifier
      LifeCycle lc = ad.getLifeCycle(inTokens[0]);
      datasetId = lc.getTokens("Shared storage.Created")[0].getUid();

      // Remove the data set annotations from the catalog
      String url = "https://catalog.globus.org/dataset/" + datasetId;
      Runtime r = Runtime.getRuntime();
      Process p = r.exec("catalog_client.py remove " + url);
      p.waitFor();

      // Locally, remove the data sets
      String path = "~/aps/" + datasetId;
      FileUtils.deleteDirectory(new File(path));

      // Publish the "Detector.End"
      Token root = lc.getTokens("Detector.Created")[0];
      ad.publishTransition("Detector.End", lc);

      // Notify the user
      sendEmail("user@server.com", "APS - Corrupted data set " + datasetId);
   }
};

HandlerGuard guard = new HandlerGuard() {
   public boolean accept(Transition t, Token[] inTokens, Token[] outTokens) {
      return inTokens[0].hasTag("failure-corrupted");
   }
}

ad.subscribeTo("Swift.Failure", handler, guard);
```

Listing 5.6: Automatically recover from faulty files.

5.4.4 Error Detection and Recovery

We finally consider the problem of detecting and recovering from experiment-wide errors. Faulty files sometimes acquired by the detector are only identified during analysis, near the end of the process. With its limited scope, the analysis program can only fail, with the effect of also stopping the Swift script. In this situation, the measures taken by users are to drop the data set entirely and to reacquire the data set with the same (or fixed) parameters. This procedure can take a considerable amount of time and represents a significant overhead on the experiment process. We show here how the surveillance framework can detect these errors immediately by observing Swift failures and automatically take the appropriate recovery measures, in addition to notifying the user.

The code presented in Listing 5.6 benefits from the high-level view of the whole experiment to automatically recover from errors in the same way that the user would. The handler is executed on a node of the shared storage cluster for any Swift token with a "failure-corrupted" tag. The handler uses the Active Data client API to retrieve the `LifeCycle` object that corresponds to the file, as previously shown in section 5.4.1. The `LifeCycle` object is used to access remote elements through their identifier. Associated metadata are removed from the Globus Catalog (which will trigger the *Globus Catalog.Remove* transition), the files are removed from the shared storage, the *Detector.End* transition is triggered for the data set and finally the user is notified via email. After

all this, a transition handler running on the detector's acquisition machine can react to the *Detector.End* transition by running the acquisition again.

5.5 Conclusions

Large scientific experiments are increasingly complex, distributed and data-centric. It is common for researchers to employ a range of tools, applications, and services in their scientific processes and rely on a collection of distributed storage and compute resources. These factors collectively introduce new data management challenges and require the development of novel techniques to manage the entire data life cycle. To be widely adopted, these techniques must be as simple and user-friendly as possible. The implicit surveillance approach proposed in this Chapter allows users to monitor real-time data life cycle progress non-invasively. Users of the surveillance framework benefit from Active Data, its life cycle meta-model and execution system with no concern of the complexity resulting from the actual infrastructures. The framework provides important features, including automation, progress monitoring, sharing and notification, and error detection and recovery. We demonstrated the ease through which users and developers alike can leverage such functionality by instrumenting a real-world materials science experiment at a large user facility.

In comparison with other approaches, the proposed approach is both efficient and easy to use. The user need not perform extensive application-specific development; instead they leverage instrumented existing and commonly used scientific tools. Conversely, application developers need not implement user-specific optimisations. Instead, they make a single investment to make their scientific tools "Active Data-enabled". Creating a comparable infrastructure without Active Data would require the use and extension of existing systems (e.g., scientific workflows and provenance systems) as well as the development of significant new functionalities at several levels of the software stack.

6 Conclusions and Perspectives

Efficient data management is a key element for big data success. Existing approaches rely heavily on ad-hoc solutions that are difficult to maintain and reuse. In this thesis, we focused on designing a simple and robust approach that will help scientists manipulate enormous data sets as easily as if they could fit in the comfort of their laptop.

6.1 Conclusions

In this thesis, we tackled the problem of managing large-scale data sets on hybrid distributed infrastructures. Considering that scientific and industrial big data applications often involve multiple non-collaborative systems orchestrated by workflow or workflow-like systems, we found the need for a novel, rigorous data management paradigm. We decided that this paradigm should be data-centric and erase from users' view any element that is not strictly related to data: infrastructure and hardware details, data location, housekeeping tasks etc.

Active Data, the meta model introduced in this thesis, focuses on the most important aspects of distributed data life cycles, i.e. the state of distributed data, and operations that change their state (e.g. creation, replication, transfer, deletion). Its graphical representation naturally exposes distribution on many systems simultaneously as well as replication. Then, we have developed the Active Data runtime system that enables existing data management systems to report on their actions by publishing *data transitions* to the world. Conversely, users can react to these actions by automatically running code when a data transition has been published. The resulting programming model allows not only users but also any program to get the current state of any data item, and use it to make informed decisions. Overall, the combination of the formal meta model and of the runtime system leads the way for more clever data management, and thus more valuable data sets.

The Active Data programming model offers a rich set of features; its asynchronous execution model allows a large number of clients to run code without impairing normal operations; the life cycle catalog enables clients to query the complete state of a data item distributed on multiple systems and infrastructures from a partial knowledge—a

local data identifier; the tagging system enables clients to have information travel across systems in an elegant way, easing coordination between systems.

Active Data was evaluated in three ways that witness that the idea defended in the thesis is both realistic and relevant to its goals. First, a set of micro benchmarks were conducted on the Grid'5000 experimental testbed; these benchmarks measured how Active Data performs when facing important loads and gathered performance metrics. In particular, we showed that Active Data is adapted for use with common workloads. Second, four synthetic use-cases have been studied and demonstrated that Active Data can be used to program implicitly parallel applications, and to optimize existing systems by becoming *data life cycle-aware*. We described the implementation of a distributed storage cache; we created a distributed version of the inotify notification system; we showed how to handle dynamic data by making applications incremental, and how the unified view of data life cycles can help collecting provenance information automatically. Finally, we studied a real-life application, the Advanced Photon Source, during a two-month visit to Ian Foster and Kyle Chard at the University of Chicago and Argonne National Lab. This application demonstrates that Active Data can greatly improve existing applications without changing any of their code. It demonstrated the construction of a *surveillance framework* that can help users manage, publish and share large data sets even when heterogeneous systems and infrastructures are involved. It can be used to closely monitor the progress of scientific experiments without manipulating the systems, to recover from errors without human interventions and to share results as soon as they are produced.

We now advocate for wide adoption of Active Data by data management systems developers to offer users simple, efficient and safe tools to cure, exploit and share their data with the world, and to encourage challenged users to take a chance at big data.

6.2 Future Directions

At key times in the course of this thesis, some research directions have been preferred over others, leaving entire territories unexplored. Here we describe some of the directions that can be followed in the near future and provide pointers to the adventurous reader who would like to contribute to this new exciting project.

Provenance Recording In Chapter 4, we demonstrated how one can use Active Data to capture and make sense of provenance information coming from two non-collaborative systems. While these results are promising at this scale, we feel that Active Data's ability to collect everything that happens to any data item in a system has great potential for exhaustive, end-to-end provenance capture. A number of provenance databases exist, featuring different languages and schemas for representing and storing provenance information [121, 126, 139, 142]. Special transition handlers could be implemented and, when subscribed to all the transitions in a life cycle model, convert data events into provenance information. However, the information that Active Data currently gets from data management systems is only that a specific operation has occurred. Transition handlers are executed in a different context than the one that performed the operation. The difficulty for extensive provenance recording will be to make sure that in this context, Active Data can gather enough information about the environment where an operation was executed, e.g. the version of the operating system, installed programs and libraries, environment variables.

Data Resources Traceability In the continuation of automatic and exhaustive provenance collection, Active Data could lead to better promote research results and asses the quality of data over extended time periods. Within the next few years, we can imagine that scientists will publish their data sets the way they currently publish research, and reuse data sets the way they cite research. In this perspective Active Data offers solutions to many challenges by allowing to track data sets from the acquisition machine to the research paper and to the editor. Active Data can also enable scientists to measure the quality of data sets with new metrics, such as how expensive or eco-friendly a data set is, or valuable it is based on its citations and derivations. To make this happen, after all data transitions have been identified and integrated in a data life cycle model, the first challenge would be to associate a "cost" to each transition e.g. CPU cycles, power consumption, financial cost, emitted CO_2. Establishing these cost models would require extensive static and dynamic analysis of data management systems, analytics programs and storage systems. The cost model is tightly linked to determining how much of the power consumed by a computer corresponds to a specific program or instruction, which is an open problem at the moment and can only be estimated. Further, a second challenge is to implement the infrastructure of such a system. In particular, new ways of uniquely identifying data sets and versions will be needed as scientists will produce and reuse data sets in a common namespace that spans over all continents and scientific domains [162].

Cross-system optimization In this thesis, we have only started to explore the possibilities of cross-system optimizations in the so called "big data stack", i.e. the software and hardware stack used to manipulate big data. However, this stack often comprises dozens of software, most of which cannot react in a smart way when other components in the stack fail. This perfectly normal situation (after all, a software is naturally limited by it scope) can be improved by extending the scope of data management systems. For example, consider how HDFS replicates files; by default, it keeps 3 replicas of each files, with at least a replica in a separate rack. When a replica is missing, HDFS creates a new one to maintain a count of 3 replicas per file. However, when both racks go down at once, HDFS loses all 3 replicas and reports the file as permanently lost. This situation illustrates the challenge in cross-system optimization: if we widen the scope of HDFS, it could silently recover the file from somewhere else. Another example would involve two systems running on top of HDFS like Pig [87] and YSmart [88]. Each would read the same HDFS files and possibly compute identical intermediate files, also stored in HDFS. Because they do not collaborate, they cannot take advantage of this situation and use the already computed intermediate files. The idea of using Active Data for this kind of tasks was explored in Chapter 5 and needs to be extended. To this end, we devise several research axis that combined together will make transparent cross-system optimizations a reality. The first axis is more of a prerequisite and requires making every data management system participating in an application Active Data-enabled. The second axis is to formalize the concept of cross-system communication and optimization through generic interfaces; this means allowing systems to expose problems they are facing (e.g. data loss, inconsistencies, full storage), and solutions they can provide (e.g. data replicated, possibility to regenerate a file). This interface is partly offered by Active Data and can be extended by the addition of a set of standard life cycle transitions that would mean the same thing for any life cycle model (data lost, data replicated, etc.) in addition to the existing "data created" and "data deleted". A third ambitious research axis would focus on making cross-system optimizations effort-

less with massive code reuse, by providing a set of generic transition handlers for anyone to use, in a sort of centralized transition handlers store.

Cloud Service Deployment and Monitoring This thesis has explored using a meta model to represent the internal life cycle of data management systems and interactions between them; this is a *top-down* approach. A *bottom-up* approach could use the meta model to specify interactions between systems *a priori*. Asma Ben Cheikh is a Ph.D candidate at the University of Tunis; she is currently developing system that takes a data life cycle model as a specification for the deployment of applications in the cloud. First, each service available for deployment must be modeled and made Active Data-enabled; then, users describe the services they need and their interaction by composing the life cycle models of different services into one model; last, the deployment system reads the data life cycle model and deploys the necessary services. The deployed infrastructure fully reports to an Active Data instance that users can use to monitor their applications.

Verification Active Data defines a meta model based on Petri Networks to represent the life cycle of data in distributed systems. In this thesis, the meta model has been mostly used for graphical representations and to ensure at runtime that an execution is correct with respect to a user-provided model. There is however much more to do in the domain of analysis of life cycle models. Petri Network specialists could study classical techniques to prove properties on life cycle models, helping users determine whether their construction is correct, through liveness or the absence of data loss. Verification techniques could also open the door to automatic composition, i.e. allowing a program to compose life cycle models together, possibly live, to reflect a situation (observing a running system and composing a life cycle model as it go), or to allow life cycle models to be composed from higher-level specifications by an automated process, as it will be the case for cloud services deployment.

Bibliography

[1] The Large Hadron Collider. `http://home.web.cern.ch/topics/large-hadron-collider`. (Cited page 1.)

[2] The Large Synoptic Survey Telescope. `http://www.lsst.org/lsst/`. (Cited page 1.)

[3] The OOI RSN Cable System. `http://www.interactiveoceans.washington.edu/story/The_OOI_RSN_Cable_System`. (Cited page 1.)

[4] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The google cluster architecture. *Micro, IEEE*, 23(2):22–28, March 2003. (2 citations pages 1 and 9.)

[5] Anthony Simonet, Gilles Fedak, and Matei Ripeanu. Active Data: A Programming Model to Manage Data Life Cycle Across Heterogeneous Systems and Infrastructures. *Future Generation Computer Systems*, page 49, 2015. (Cited page 4.)

[6] Gabriel Antoniu, Julien Bigot, Christophe Blanchet, Luc Bougé, François Briant, Franck Cappello, Alexandru Costan, Frédéric Desprez, Gilles Fedak, Sylvain Gault, Kate Keahey, Bogdan Nicolae, Christian Pérez, Anthony Simonet, Frédéric Suter, Bing Tang, and Raphael Terreux. Scalable data management for mapreduce-based data-intensive applications: a view for cloud and hybrid infrastructures. *International Journal of Cloud Computing*, 2(2):150–170, 2013. (Cited page 4.)

[7] Anthony Simonet, Kyle Chard, Gilles Fedak, and Ian Foster. Using active data to provide smart data surveillance to e-science users. In *Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed and Network-based Processing*, PDP 2015. IEEE, 2015. (Cited page 4.)

[8] Gabriel Antoniu, Julien Bigot, Christophe Blanchet, Luc Bougé, François Briant, Franck Cappello, Alexandru Costan, Frédéric Desprez, Gilles Fedak, Sylvain Gault, Kate Keahey, Bogdan Nicolae, Christian Pérez, Anthony Simonet, Frédéric Suter, Bing Tang, and Raphael Terreux. Towards Scalable Data Management for Map-Reduce-based Data-Intensive Applications on Cloud and Hybrid Infrastructures. In *1st International IBM Cloud Academy Conference - ICA CON 2012*, Research Triangle Park, North Carolina, 2012. (Cited page 4.)

[9] Anthony Simonet, Gilles Fedak, Matei Ripeanu, and Samer Al-Kiswany. Active data: a data-centric approach to data life-cycle management. In *Proceedings of the 8th Parallel Data Storage Workshop*, pages 39–44. ACM, 2013. (Cited page 5.)

[10] Anthony Simonet. Active data: Un modèle pour représenter et programmer le cycle de vie des données distribuées. In *ComPAS'2014*, 2014. (Cited page 5.)

[11] Anthony Simonet, Gilles Fedak, and Matei Ripeanu. Active Data: A Programming Model to Manage Data Life Cycle Across Heterogeneous Systems and Infrastructures. Research Report RR-8062, Inria - Research Centre Grenoble – Rhône-Alpes ; ENS Lyon ; University of British Columbia, 2015. (Cited page 5.)

[12] Min Chen, Shiwen Mao, and Yunhao Liu. Big data: A survey. *Mobile Networks and Applications*, 19(2):171–209, 2014. (Cited page 7.)

[13] Viktor Mayer-Schönberger and Kenneth Cukier. *Big Data: A revolution that will transform how we live, work, and think*. Houghton Mifflin Harcourt, 2013. (Cited page 7.)

[14] Danah Boyd and Kate Crawford. Critical questions for big data: Provocations for a cultural, technological, and scholarly phenomenon. *Information, Communication & Society*, 15(5):662–679, 2012. (Cited page 7.)

[15] Lev Manovich. *Trending: the promises and the challenges of big social data*, chapter 27. The University of Minnesota Press, Minneapolis, 2012. (2 citations pages 7 and 13.)

[16] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787 – 2805, 2010. (Cited page 8.)

[17] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497 – 1516, 2012. (Cited page 8.)

[18] William Gropp, Ewing Lusk, and Thomas Lawrence Sterling. *Beowulf cluster computing with Linux*. MIT Press, Cambridge, MA, USA, 2 edition, 2002. (Cited page 9.)

[19] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: A high performance, server-centric network architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 63–74, New York, NY, USA, 2009. ACM. (Cited page 9.)

[20] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23(3):187 – 200, 2000. (Cited page 9.)

[21] Srikumar Venugopal, Rajkumar Buyya, and Kotagiri Ramamohanarao. A taxonomy of data grids for distributed data sharing, management, and processing. *ACM Computing Surveys*, 38(1), mar. 2006. (Cited page 9.)

[22] Peter Z. Kunszt and Leanne P. Guy. *The Open Grid Services Architecture, and Data Grids*, pages 385–407. John Wiley & Sons, Ltd, 2003. (Cited page 9.)

[23] Kavitha Ranganathan and Ian Foster. Identifying dynamic replication strategies for a high-performance data grid. In CraigA. Lee, editor, *Grid Computing — GRID 2001*, volume 2242 of *Lecture Notes in Computer Science*, pages 75–86. Springer Berlin Heidelberg, 2001. (Cited page 9.)

[24] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001. (Cited page 9.)

[25] Fran Berman, Geoffrey Fox, and Anthony JG Hey. *Grid Computing: Making the Global Infrastructure a Reality*, volume 2. John Wiley and sons, 2003. (Cited page 9.)

[26] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, 2003. (Cited page 9.)

[27] Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data management in an international data grid project. In Rajkumar Buyya and Mark Baker, editors, *Grid Computing — GRID 2000*, volume 1971 of *Lecture Notes in Computer Science*, pages 77–90. Springer Berlin Heidelberg, 2000. (Cited page 9.)

[28] Diana Bosio, James Casey, Akos Frohner, Leanne Guy, Peter Kunszt, Erwin Laure, Sophia Lemaitre, Levi Lucio, Heinz Stockinger, Kurt Stockinger, William Bell, David Cameron, Gavin McCance, Paul Millar, Joni Hahkala, Niklas Karlsson, Ville Nenonen, Mika Silander, Olle Mulmo, Gian-Luca Volpato, Guiseppe Andronico, Federico DiCarlo, Livio Salconi, Andrea Domenici, Ruben Carvajal-Schiaffino, and Floriano Zini. Next-generation EU DataGrid data management services. *ArXiv Physics e-prints*, may 2003. (Cited page 9.)

[29] Worldwide LHC Computing Grid. `http://wlcg.web.cern.ch/`. (Cited page 9.)

[30] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop*, GCE '08, pages 1–10, Nov 2008. (Cited page 10.)

[31] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. A taxonomy and survey of cloud computing systems. In *Proceedings of the 5th International Joint Conference on INC, IMS and IDC*, NCM '09, pages 44–51, Aug 2009. (Cited page 10.)

[32] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud computing: Issues and challenges. In *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications*, AINA 2010, pages 27–33, April 2010. (Cited page 10.)

[33] Yi Wei and M. Brian Blake. Service-oriented computing and cloud computing: Challenges and opportunities. *Internet Computing, IEEE*, 14(6):72–75, Nov 2010. (Cited page 10.)

[34] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, April 2010. (Cited page 10.)

[35] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: An experiment in public-resource computing. *Communications of the ACM*, 45:56–61, November 2002. (Cited page 10.)

[36] SETI@home project stats. `http://www.allprojectstats.com/po.php?projekt=15`. (Cited page 10.)

[37] David P. Anderson. BOINC: a system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Nov 2004. (Cited page 10.)

[38] H. Abbes, C. Cerin, and M. Jemni. Bonjourgrid: Orchestration of multi-instances of grid middlewares on institutional desktop grids. In *Proceedings of the 2009 IEEE International Symposium on Parallel Distributed Processing*, IPDPS 2009, pages 1–8, May 2009. (Cited page 10.)

[39] Heshan Lin, Jeremy Archuleta, Xiaosong Ma, Wu-chun Feng, Zhe Zhang, and Mark Gardner. MOON: MapReduce On Opportunistic eNvironments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 95–106, New York, NY, USA, 2010. ACM. (Cited page 10.)

[40] Simon Delamare, Gilles Fedak, Derrick Kondo, and Oleg Lodygensky. Spequlos: A qos service for bot applications using best effort distributed computing infrastructures. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 173–186, New York, NY, USA, 2012. ACM. (Cited page 10.)

[41] Tran Doan Thanh, Subaji Mohan, Eunmi Choi, SangBum Kim, and Pilsung Kim. A taxonomy and survey on distributed file systems. In *Proceedings of the 4th International Conference on Networked Computing and Advanced Information Management*, volume 1 of *NCM '08*, pages 144–149, Sept 2008. (Cited page 11.)

[42] K. McKusick and S. Quinlan. Gfs: Evolution on fast-forward. *Commun. ACM*, 53(3):42–49, March 2010. (Cited page 11.)

[43] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI'06, pages 307–320. USENIX Association, 2006. (Cited page 12.)

[44] S. Ghemawat, H. Gobioff, and S. Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP'03, pages 29–43, New York, NY, USA, 2003. ACM. (2 citations pages 12 and 16.)

[45] K. Shvachko, K. Hairong, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 26th Symposium on Mass Storage Systems and Technologies*, MSST 2010, pages 1–10. IEEE, May 2010. (5 citations pages 12, 14, 16, 30, and 63.)

[46] P. Schwan. Lustre, Building a File System for 1,000-node Clusters. In *Proceedings of the Linux Symposium*, 2003. (Cited page 12.)

[47] D. Thain, C. Moretti, and J. Hemmes. Chirp: a practical global filesystem for cluster and grid computing. *Journal of Grid Computing*, 7(1):51–72, 2009. (Cited page 12.)

[48] Filesystem in Userspace. `http://fuse.sourceforge.net/`. (Cited page 12.)

[49] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie. Blobseer: Next-generation data management for large scale infrastructures. *Journal of Parallel and Distributed Computing*, 71(2):169 – 184, 2011. Data Intensive Computing. (Cited page 12.)

[50] Samer Al-Kiswany, Abdullah Gharaibeh, and Matei Ripeanu. The case for a versatile storage system. *SIGOPS Operating Systems Review*, 44(1):10–14, March 2010. (Cited page 12.)

[51] Frank Schmuck and Roger 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, Berkeley, CA, USA, 2002. USENIX Association. (Cited page 13.)

[52] L.B. Costa, H. Yang, E. Vairavanathan, A. Barros, K. Maheshwari, G. Fedak, D. Katz, M. Wilde, M. Ripeanu, and S. Al-Kiswany. The case for workflow-aware storage:an opportunity study. *Journal of Grid Computing*, 13(1):95–113, 2015. (Cited page 13.)

[53] Lauro Beltrão Costa, Samer Al-Kiswany, Hao Yang, and Matei Ripeanu. Supporting storage configuration for i/o intensive workflows. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 191–200, New York, NY, USA, 2014. ACM. (Cited page 13.)

[54] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in dram. *SIGOPS Operating Systems Review*, 43(4):92–105, January 2010. (Cited page 13.)

[55] Brad Fitzpatrick. Distributed caching with Memcached. *Linux Journal*, 2004(124):5, 2004. (Cited page 13.)

[56] David DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992. (Cited page 13.)

[57] Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, December 2000. (Cited page 13.)

[58] Jim Gray and Andreas Reuters. *Transaction processing: concepts and techniques*. Morgan Kaufmann Publishers, San Mateo, California, 1993. (Cited page 13.)

[59] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of the 20th International Conference on Distributed Computing Systems*, ICDCS 2000, pages 464–474. IEEE, 2000. (Cited page 13.)

[60] B.G. Tudorica and C. Bucur. A comparison between several nosql databases with comments and notes. In *Proceedings of the 10th Roedunet International Conference*, RoEduNet, pages 1–5. IEEE, June 2011. (Cited page 14.)

[61] Kristina Chodorow. *MongoDB: the definitive guide*. O'Reilly Media, Inc., 2013. (Cited page 14.)

[62] J. Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: the definitive guide*. O'Reilly Media, Inc., 2010. (Cited page 14.)

[63] The Redis open-source key-value cache and store. `http://redis.io/`. (Cited page 14.)

[64] Project Voldemort. `http://www.project-voldemort.com/voldemort/`. (Cited page 14.)

[65] Amazon DynamoDB. `http://aws.amazon.com/fr/dynamodb/`. (Cited page 14.)

[66] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2):4:1–4:26, 2008. (2 citations pages 14 and 18.)

[67] M.N. Vora. Hadoop-hbase for large-scale data. In *Proceedings of the 2011 International Conference on Computer Science and Network Technology*, volume 1 of *ICCSNT*, pages 601–605, Dec 2011. (2 citations pages 14 and 18.)

[68] Arcot Rajasekar, Reagan Moore, Chien-yi Hou, A. Lee Lee, Christopher, Richard Marciano, Antoine de Torcy, Michael Wan, Wayne Schroeder, Sheau-Yen Chen, Lucas Gilbert, Paul Tooby, and Bing Zhu. iRODS primer: integrated rule-oriented data system. *Synthesis Lectures on Information Concepts, Retrieval, and Services*, 2(1), 2010. (3 citations pages 14, 30, and 63.)

[69] Baohua Wei, Gilles Fedak, and Frank Cappello. Collaborative data distribution with bittorrent for computational desktop grids. In *Parallel and Distributed Computing, 2005. ISPDC 2005. The 4th International Symposium on*, pages 250–257, July 2005. (Cited page 15.)

[70] The BitTorrent Protocol Specification. `http://www.bittorrent.org/beps/bep_0003.html`. (Cited page 15.)

[71] Gilles Fedak, Haiwu He, and Franck Cappello. Bitdew: a programmable environment for large-scale data management and distribution. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC'08, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press. (2 citations pages 15 and 63.)

[72] Tom Goodale, Shantenu Jha, Hartmut Kaiser, Thilo Kielmann, Pascal Kleijer, Gregor Von Laszewski, Craig Lee, Andre Merzky, Hrabri Rajic, and John Shalf. Saga: A simple api for grid applications. high-level application programming on the grid. *Computational Methods in Science and Technology*, 12(1):7–20, 2006. (Cited page 15.)

[73] Ttevfik Kosar and Miron Livny. Stork: Making data placement a first class citizen in the grid. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 342–349, 2004. (Cited page 15.)

[74] Jiangyan Xu and Renato Figueiredo. Gatorshare: A file system framework for high-throughput data management. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 776–786, New York, NY, USA, 2010. ACM. (Cited page 15.)

[75] Globus online. `https://www.globus.org/`. (Cited page 15.)

[76] Ian Foster. Globus Online: Accelerating and democratizing science through cloud-based services. *IEEE Internet Computing*, 15(3):70–73, 2011. (2 citations pages 15 and 63.)

[77] I. Mandrichenko, W. Allcock, and T. Perelmutov. Gridftp v2 protocol description. `http://www.ogf.org/documents/GFD.47.pdf`. (Cited page 15.)

[78] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009. (Cited page 15.)

[79] Henry Kasim, Verdi March, Rita Zhang, and Simon See. Survey on parallel programming model. In Jian Cao, Minglu Li, Min-You Wu, and Jinjun Chen, editors, *Network and Parallel Computing*, volume 5245 of *Lecture Notes in Computer Science*, pages 266–275. Springer Berlin Heidelberg, 2008. (Cited page 15.)

[80] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. (3 citations pages 16, 19, and 63.)

[81] Apache Hadoop. `http://hadoop.apache.org/`. (2 citations pages 16 and 63.)

[82] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. Parallel data processing with mapreduce: A survey. *SIGMOD Record*, 40(4):11–20, January 2012. (Cited page 16.)

[83] Christos Doulkeridis and Kjetil Nørvåg. A survey of large-scale analytical query processing in mapreduce. *The VLDB Journal*, 23(3):355–380, 2014. (Cited page 16.)

[84] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked Systems Design and Implementation*, volume 10 of *NSDI*, page 20, 2010. (Cited page 16.)

[85] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC'10, pages 810–818, New York, NY, USA, 2010. ACM. (2 citations pages 16 and 18.)

[86] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010. (2 citations pages 16 and 18.)

[87] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD'08, pages 1099–1110, New York, NY, USA, 2008. ACM. (2 citations pages 16 and 95.)

[88] Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, Yongqiang He, and Xiaodong Zhang. Ysmart: Yet another sql-to-mapreduce translator. In *Proceedings of the 31st International Conference on Distributed Computing Systems*, ICDCS 2011, pages 25–36. IEEE, 2011. (2 citations pages 16 and 95.)

[89] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009. (Cited page 16.)

[90] Jimmy Lin. Mapreduce is good enough? if all you have is a hammer, throw away everything that's not a nail! *Big Data*, 1(1):28–37, 2013. (Cited page 17.)

[91] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM. (Cited page 17.)

[92] Christopher Moretti, Jared Bulosan, Douglas Thain, and Patrick J. Flynn. All-pairs: An abstraction for data-intensive cloud computing. In *Proceedings of the International Symposium on Parallel and Distributed Processing*, IPDPS, pages 1–11. IEEE, April 2008. (Cited page 17.)

[93] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot Topics in Cloud Computing*, pages 10–10, 2010. (Cited page 17.)

[94] Kenjiro Taura, Kenji Kaneda, Toshio Endo, and Akinori Yonezawa. Phœnix: a parallel programming model for accommodating dynamically joining/leaving resources. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 216–229, New York, NY, USA, 2003. ACM. (Cited page 17.)

[95] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. (Cited page 17.)

[96] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, volume abs/1408.2041 of *UAI2010*, 2014. (Cited page 17.)

[97] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, April 2012. (Cited page 17.)

[98] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association. (Cited page 17.)

[99] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997. (Cited page 17.)

[100] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM. (Cited page 17.)

[101] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, August 2003. (Cited page 18.)

[102] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM. (Cited page 18.)

[103] Samuel Madden and Michael J. Franklin. Fjording the stream: an architecture for queries over streaming sensor data. In *Proceedings of the 18th International Conference on Data Engineering*, pages 555–566, 2002. (Cited page 18.)

[104] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system, 2002. (Cited page 18.)

[105] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Çetintemel, Micheal Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE 2005, pages 779–790, April 2005. (Cited page 18.)

[106] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C. Webb, and Ken Yocum. Stateful bulk processing for incremental analytics. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 51–62, New York, NY, USA, 2010. ACM. (Cited page 18.)

[107] Sebastian Burckhardt, Daan Leijen, Caitlin Sadowski, Jaeheon Yi, and Thomas Ball. Two for the price of one: A model for parallel and incremental computation. *SIGPLAN Not.*, 46(10):427–444, October 2011. (Cited page 18.)

[108] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–15, Berkeley, CA, USA, 2010. USENIX Association. (Cited page 18.)

[109] Shivaram Venkataraman, Indrajit Roy, Alvin AuYoung, and Robert S. Schreiber. Using r for iterative and incremental processing. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 11–11, Berkeley, CA, USA, 2012. USENIX Association. (Cited page 18.)

[110] Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996. (Cited page 18.)

[111] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquini. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 7:1–7:14, New York, NY, USA, 2011. ACM. (Cited page 18.)

[112] Pramod Bhatotia, Alexander Wieder, İstemi Ekin Akkuş, Rodrigo Rodrigues, and Umut A. Acar. Large-scale incremental data processing with change propagation. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'11, pages 18–18, Berkeley, CA, USA, 2011. USENIX Association. (Cited page 18.)

[113] Dimitri P. "Bertsekas and John N." Tsitsiklis. "some aspects of parallel and distributed iterative algorithms—a survey". *"Automatica"*, "27"("1"):"3 – 21", "1991". (Cited page 18.)

[114] Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox. Mapreduce for data intensive scientific analyses. In *Proceedings of the 4th International Conference on eScience*, eScience '08, pages 277–284. IEEE, Dec 2008. (Cited page 18.)

[115] Boduo Li, Edward Mazur, Yanlei Diao, Andrew McGregor, and Prashant Shenoy. A platform for scalable one-pass analytics using mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 985–996, New York, NY, USA, 2011. ACM. (2 citations pages 18 and 19.)

[116] Michael Wilde, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011. (Cited page 19.)

[117] Chua Ching Lian, Francis Tang, Praveen Issac, and Arun Krishnan. Gel: Grid execution language. *Journal of Parallel and Distributed Computing*, 65(7):857 – 869, 2005. (Cited page 19.)

[118] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005. (Cited page 19.)

[119] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J. Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 2014. (Cited page 19.)

[120] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM. (Cited page 19.)

[121] Yogesh L Simmhan, Beth Plale, and Dennis Gannon. A survey of data provenance techniques. Technical Report 47405, Computer Science Department, Indiana University, Bloomington IN, 2005. (2 citations pages 20 and 94.)

[122] Kiran-Kumar Muniswamy-Reddy, Peter Macko, and Margo Seltzer. Provenance for the cloud. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 15–14, Berkeley, CA, USA, 2010. USENIX Association. (Cited page 20.)

[123] Luc Moreau, Paul Groth, Simon Miles, Javier Vazquez-Salceda, John Ibbotson, Sheng Jiang, Steve Munroe, Omer Rana, Andreas Schreiber, Victor Tan, et al. The provenance of electronic data. *Communications of the ACM*, 51(4):52–58, 2008. (Cited page 20.)

[124] Luc Moreau, Juliana Freire, Joe Futrelle, Robert McGrath, Jim Myers, and Patrick Paulson. The open provenance model. Technical report, University of Southampton, December 2007. (Cited page 20.)

[125] Luc Moreau, Beth Plale, Simon Miles, Carole Goble, Paolo Missier, Roger Barga, Yogesh Simmhan, Joe Futrelle, Robert McGrath, Jim Myers, Patrick Paulson, Shawn Bowers, Bertram Ludaescher, Natalia Kwasnikowska, Jan Van den Bussche, Tommy Ellkvist, Juliana Freire, and Paul Groth. The open provenance model (v1.01). Technical report, University of Southampton, July 2008. (Cited page 20.)

[126] Luc Moreau, Ben Clifford, Juliana Freire, Yolanda Gil, Paul Groth, Joe Futrelle, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Beth Plale, Yogesh Simmhan, Eric Stephan, and Jan Van den Bussche. The open provenance model—core specification (v1. 1). *Future Generation Computer Systems*, 2009. (2 citations pages 20 and 94.)

[127] Luc Moreau, Juliana Freire, Joe Futrelle, Robert E McGrath, Jim Myers, and Patrick Paulson. The open provenance model: An overview. In *Provenance and Annotation of Data and Processes*, pages 323–326. Springer, 2008. (Cited page 20.)

[128] Yogesh Simmhan, Paul Groth, and Luc Moreau. Special section: The third provenance challenge on using the open provenance model for interoperability. *Future Generation Computer Systems*, 27(6):737 – 742, 2011. (Cited page 20.)

[129] Paul Groth and Luc Moreau. Representing distributed systems using the open provenance model. *Future Generation Computer Systems*, 27(6):757 – 765, 2011. (Cited page 20.)

[130] Natalia Kwasnikowska and Jan Van den Bussche. Mapping the nrc dataflow model to the open provenance model. In Juliana Freire, David Koop, and Luc Moreau, editors, *Provenance and Annotation of Data and Processes*, volume 5272 of *Lecture Notes in Computer Science*, pages 3–16. Springer Berlin Heidelberg, 2008. (Cited page 20.)

[131] Yogesh Simmhan and Roger Barga. Analysis of approaches for supporting the open provenance model: a case study of the trident workflow workbench. *Future Generation Computer Systems*, 27(6):790 – 796, 2011. (Cited page 20.)

[132] Peter Buneman, Adriane Chapman, and James Cheney. Provenance management in curated databases. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 539–550, New York, NY, USA, 2006. ACM. (Cited page 20.)

[133] Robert D. Stevens, Alan J. Robinson, and Carole A. Goble. mygrid: personalised bioinformatics on the information grid. *Bioinformatics*, 19(suppl 1):i302–i304, 2003. (Cited page 20.)

[134] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, Jiten Bhagat, Khalid Belhajjame, Finn Bacall, Alex Hardisty, Abraham Nieva de la Hidalga, Maria P. Balcazar Vargas, Shoaib Sufi, and Carole Goble. The Taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic Acids Research*, 41(W1):W557–W561, 2013. (Cited page 21.)

[135] Apache Derby. `http://db.apache.org/derby/`. (Cited page 21.)

[136] Paolo Missier, SatyaS. Sahoo, Jun Zhao, Carole Goble, and Amit Sheth. Janus: From workflows to semantic provenance and linked open data. In DeborahL. McGuinness, JamesR. Michaelis, and Luc Moreau, editors, *Provenance and Annotation of Data and Processes*, volume 6378 of *Lecture Notes in Computer Science*, pages 129–141. Springer Berlin Heidelberg, 2010. (Cited page 21.)

[137] Simon Miles, Paul Groth, Miguel Branco, and Luc Moreau. The requirements of using provenance in e-science experiments. *Journal of Grid Computing*, 5(1):1–25, 2007. (Cited page 21.)

[138] Paul Groth, Michael Luck, and Luc Moreau. A protocol for recording provenance in service-oriented grids. In Teruo Higashino, editor, *Principles of Distributed Systems*, volume 3544 of *Lecture Notes in Computer Science*, pages 124–139. Springer Berlin Heidelberg, 2005. (Cited page 21.)

[139] Bin Cao, Beth Plale, Girish Subramanian, Ed Robertson, and Yogesh Simmhan. Provenance information model of karma version 3. In *2009 IEEE Congress on Services*, Services 2009, pages 348–351. IEEE, July 2009. (2 citations pages 21 and 94.)

[140] Ian Foster, Jens Vöckler, Michael Wilde, and Yong Zhao. Chimera: a virtual data system for representing, querying, and automating data derivation. In *Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, pages 37–46, 2002. (Cited page 21.)

[141] Uri Braun, Simson Garfinkel, David A. Holland, Kiran-Kumar Muniswamy-Reddy, and Margo I. Seltzer. Issues in automatic provenance collection. In *Provenance and Annotation of Data*, pages 171–183. Springer, 2006. (Cited page 21.)

[142] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference*, USENIX, pages 43–56, 2006. (2 citations pages 21 and 94.)

[143] The home of the U.S. Government's open data. `http://www.data.gov`. (Cited page 22.)

[144] Open platform for french public data. `https://www.data.gouv.fr/en/`. (Cited page 22.)

[145] Sense - Cloud data platform. `https://sense.io/`. (Cited page 22.)

[146] Datahub - The easy way to get, use and share data. `http://datahub.io/`. (Cited page 22.)

[147] Anant P. Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J. Elmore, Samuel Madden, and Aditya G. Parameswaran. Datahub: Collaborative data science & dataset version management at scale. *CoRR*, abs/1409.0798, 2014. (Cited page 22.)

[148] Yuri Demchenko, Paola Grosso, Cees de Laat, and Peter Membrey. Addressing big data issues in scientific data infrastructure. In *Proceedings of the 2013 International Conference on Collaboration Technologies and Systems*, CTS 2013, pages 48–55. IEEE, 2013. (Cited page 22.)

[149] Tim Ho and David Abramson. Active data: Supporting the grid data life cycle. In *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid*, CCGrid 2007, pages 39–46, 2007. (Cited page 22.)

[150] Lavanya Ramakrishnan, Devarshi Ghoshal, Valerie Hendrix, Eugen Feller, Pradeep Mantha, and Christine Morin. Storage and Data Life Cycle Management in Cloud Environments with FRIEDA. In Xiaolin Li and Judy Qiu, editors, *Cloud Computing for Data Intensive Applications*. Springer, 2015. (Cited page 22.)

[151] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, September 1977. (Cited page 32.)

[152] Tadao Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, pages 541–580, April 1989. (Cited page 32.)

[153] W.M.P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 08(01):21–66, 1998. (Cited page 32.)

[154] C.V. Ramamoorthy and Gary S. Ho. Performance evaluation of asynchronous concurrent systems using petri nets. *Software Engineering, IEEE Transactions on*, SE-6(5):440–449, Sept 1980. (Cited page 32.)

[155] K. Jensen. Coloured petri nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties*, volume 254 of *Lecture Notes in Computer Science*, pages 248–299. Springer Berlin Heidelberg, 1987. (Cited page 38.)

[156] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35:114–131, June 2003. (Cited page 55.)

[157] Robert Love. Kernel korner: intro to inotify. *Linux Journal*, 2005(139):8–, 2005. (2 citations pages 59 and 63.)

[158] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stéphane Lanteri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Iréa Touche. Grid5000: A large scale highly reconfigurable experimental grid testbed. *International Journal on High Performance Computing and Applications*, 2006. (Cited page 67.)

[159] Bing Tang, Mircea Moca, Stèphane Chevalier, Haiwu He, and Gilles Fedak. Towards MapReduce for Desktop Grid Computing. In *Fifth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC'10)*, pages 193–200, Fukuoka, Japan, November 2010. IEEE. (2 citations pages 71 and 76.)

[160] Amazon simple storage service. `http://aws.amazon.com/s3/`, 2010. (Cited page 72.)

[161] Advanced photon source. `http://www.aps.anl.gov/`, 2010. (Cited page 81.)

[162] Matan Gavish and David Donoho. A universal identifier for computational results. *Procedia Computer Science*, 4(0):637 – 647, 2011. Proceedings of the International Conference on Computational Science, {ICCS} 2011. (Cited page 95.)