Active Data: A Programming Model for Managing Big Data Life Cycle

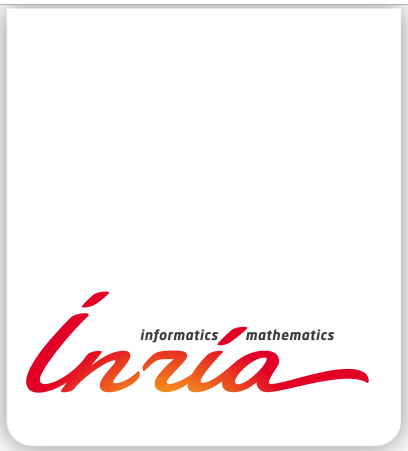
Anthony Simonet, Gilles Fedak, Matei Ripeanu

Active Data: A Programming Model for Managing Big Data Life Cycle

Anthony Simonet, Gilles Fedak, Matei Ripeanu

RESEARCH REPORT
N° 8062
September 2012
Project-Team Avalon

ISSN 0249-6399 ISRN INRIA/RR--8062--FR+ENG



Active Data: A Programming Model for Managing Big Data Life Cycle

Anthony Simonet*, Gilles Fedak*, Matei Ripeanu†

Project-Team Avalon

Research Report n° 8062 — September 2012 — 23 pages

Abstract: The Big Data challenge consists in managing, storing, analyzing and visualizing these ever growing huge datasets to extract sense and knowledge. As the volume of data grows exponentially, the management of these data becomes more complex in proportion. A key point is to handle the complexity of the data life cycle, i.e. the various operations performed on data: transfer, archiving, replication, deletion... To alleviate the complexity of the data life cycle, we propose Active Data, a programming model to automate and improve the expressiveness of data management applications. We first introduce the concept of *data life cycle* and define a formal model based on Petri Net. We present the concept of the Active Data programming model, which allows code execution at each stage of the data life cycle. With Active Data, routines provided by programmers are executed when a set of events (creation, replication, transfer, deletion) happen to any data. We implement and evaluate the model with three use cases: a storage cache to Amazon S3, a cooperative sensor network, and an incremental implementation of the MapReduce programming model. Altogether, these scenarios illustrate the adequateness of the model to program applications which manage distributed and dynamic data. We also show that applications that do not leverage on data life cycle can benefit from Active Data to improve their performances.

Key-words: programming model, distributed storage system, file system

* INRIA, University of Lyon
† University of British Columbia, Canada

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Active Data : un modèle de programmation pour la gestion des cycles de vie de Big Data

Résumé : Le défi Big Data consiste à gérer, stocker, analyser et visualiser des jeux de données toujours plus grands pour en extraire sens et connaissance. Alors que ces volumes de données croissent de manière exponentielle, leur gestion s'en complique d'autant. Un point clef est d'aborder la complexité du cycle de vie des données, c'est à dire les diverses opérations dans lesquelles elles sont impliquées : transfert, archivage, réplication, suppression... Pour diminuer la complexité des cycles de vie des données, nous proposons Active Data, un modèle de programmation pour automatiser et améliorer l'expressivité des applications de gestion de données. Premièrement, nous présentons le concept de *cycle de vie de données* et définissons un modèle formel basé sur les Réseaux de Pétri. Nous présentons ensuite le modèle de programmation Active Data qui permet l'exécution de code à chaque étape du cycle de vie d'une donnée. Avec Active Data, des routines fournies par le programmeur sont exécutées lorsqu'un ensemble d'événements (création, réplication, transfert, suppression) se produit sur n'importe quelle donnée. Nous implémentons et évaluons le modèle avec 3 cas d'utilisation : un cache entre une application et Amazon S3, un réseau de senseurs coopératifs et une implémentation incrémentale du modèle de programmation MapReduce. Ces scénarios illustrent l'adéquation du modèle avec la programmation d'applications qui gèrent des données distribuées et dynamiques. Nous montrons également que des applications qui ne tirent pas partie du cycle de vie des données peuvent bénéficier d'Active Data pour améliorer leurs performances.

Mots-clés : modèle de programmation, système de stockage distribué, systeme de fichiers

1 Introduction

Increasingly, the industrial innovative breakthroughs and the next scientific discoveries will depend on the capacity to extract knowledge and sense from the enourmous amount of *Big Data* information [16]. Examples vary from processing data provided by scientific instruments such as the CERN's LHC, the LSST Telescope in Chile, or the OOI large-scale underwater sensors network; grabbing, indexing and nearly instantaneously mining and searching the Web; building and traversing the billion-edge social network graphs; anticipating market and customer trends through multiple channels of information. Collecting information from various sources, recognizing patterns and returning human scale results from this "data deluge" is the new challenge the community is facing [14].

As the volume of data grows exponentially, the management of these data becomes more complex in proportion. A key challenge is to handle the complexity of *data life cycle management (DLM)*, i.e. the various operations performed on data: transfer, archiving, replication, processing, deletion... One can observe two constraints influencing data life cycle. The first one is due to users and applications which explicitely express operations on data. For example, a common pattern for the production of scientific data is the sequence which consists in the following steps: acquisition by a scientific instruments, pre-processing to reduce the size of the data and storage for further analyze. All the steps of this sequence can be expressed either as operations on data (e.g., data creation and movement) or computations on data (e.g., pre-processing). At the moment, such sequence of operations are programmed independently which makes the coordination between different systems such as the instrument, the buffered pre-processing staging, and the tiered storage difficult to achieve. The second constraint comes from the infrastructure itself. For instance, data-intense applications often imply that the data have to be transferred to a set of machines or computer infrastructures capable of processing them. At several steps during the computation, intermediate results can be backed-up and final results can be archived or moved between computing sites. To optimize the data placement in term of locality, bandwidth access, security or reliability, applications or systems also have the initiative to create or delete data replica which in turn increases the complexity of the life cycle. Finally, unpredictable events such as failures may alter the life cycle and adding fault-tolerance behavior significantly increases the complexity of the data life cycle management system. As we target more efficient usage of the infrastructures, there will be a growing need for a stronger interaction and flow of information between the infrastructure and the DLM systems.

To alleviate the complexity of data life cycle, solutions are needed to automate and improve the expressiveness of data management operations. The first challenge lies in the gigantism of these data sets which requires distributed storage and parallel processing [3]. Recently, several popular programming languages have emerged, such as MapReduce [8] or Dryad [17], which offer simple yet high-level data-centric parallel interfaces. However these languages focus more on processing large data sets rather than specifying management operations on the data. The second challenge is to capture the dynamicity of the data, i.e the fact that data may be produced incrementally, regenerated, modified or temporarily unavailable. Percolator [24] is an example of a language and its implementation, which takes into consideration data arrival, and incrementally updates the result of a computation according to the modification of the data sets. The last challenge relies in the fact that data are distributed not only within a single infrastructure but also across a large variety of infrastructures and systems. Thus effective Big Data applications should be able to coordinate the various systems handling the data and react both to events happening to the data and to events happening to the infrastructure. Event-based programming [22] is a great concept to program distributed applications which require a high level of reactiveness and flexibility. However, although the paradigm is appealing, it lacks a data-centric flavour that

would make intuitive and comprehensible the management of large, distributed and dynamic data across heterogeneous distributed computing infrastructures.

The solution we propose is Active Data, a programming model to allow code execution at each stage of the data life cycle. Intuitively, Active Data borrows from Active Message the idea of executing user-provided code when certain events occur (message reception in the case of Active Message [31]). With Active Data, once the data life cycle is known and formally defined, routines provided by programmers are executed when a set of events (creation, replication, transfer, deletion) happen to any data item. This programming model would allow to develop a broad range of DLM applications such as automated tiered storage, processing at any stage of the life cycle, coordination between acquisition mechanisms and remote storage, content delivery networks, deep storage archive, energy efficient storage, incremental workflow and so forth.

Our contribution is the following: we first present a formalism inspired by Petri Networks allowing to model data life cycles in distributed systems. We show that this model is able to capture the main stages of data life cycle, namely creation, deletion, scheduling, transfer and replication as well as transient unavailability. Next, we propose a new programming model called Active Data. We report on the design of the Active Data execution runtime and present an implementation based on the BitDew data management middleware [12]. To evaluate Active Data, we present three use cases which illustrate the versatility of the framework to program DLM applications which manage distributed and dynamic data. The first one shows how to program a local cache to a remote storage service in few lines of code allowing both an increase in performance and reducing the cost. The second scenario is a distributed sensor network which cooperate to implement data acquisition throttling. The last example shows that an existing MapReduce runtime augmented with Active Data can turn into an incremental MapReduce. We run these scenarios using the experimental platform Grid'5000 [5] and report on the experiments.

The rest of this paper is organized as follows. In Section 2, we introduce the model for data life cycle and the Active Data programming model inspired by Petri Nets. We present the runtime system and an implementation Active Data on top of BitDew in section 4. Section 5 presents several use cases with experiments on real distributed infrastructures. We discuss related works in section 6, we conclude in 7 and open discussions about further topics of interest.

2 The Active Data Programming Model

In this section, we introduce our model of data life cycle and we present the Active Data programming model.

2.1 Requirements

The life cycle of data is the course of operational stages through which data pass from the time when they enter the system to the time when they leave it. Data enter the system when they are acquired by an instrument, or created from some other data already present in the system and leave the system when they are physically erased, or when they are moved to a storage outside of the system. Between this two points in time, data progress through a serie of different stages of development, e.g. migration, duplication, archive, transfer... In the remaining of the paper, we will use the terms *data life cycle* to denote the possible state of a data item when handled by a particular system or by a user application, e.g. created, duplicated, deleted, backed-up, and *data life cycle management*, the sequence of data operations, e.g copy, file transfer, reading, performed on a data item during its lifetime. Our objective is to provide a programming model which facilitates for developers the writing of applications implementing data life cycle management. Requirements for the programming model and its implementation are the following:

- Simplify the programming of applications that implement data life cycle management. This involves 1) giving applications the ability to operate on data and 2) informing applications about operations happening to data.
- Allows to reason about data life cycle. A model is required to capture the essential life cycle stages and properties (faults, replication), to allow error checking and to mix the life cycles of various systems.
- Allows to reason about data handled by heterogeneous softwares that were not designed to collaborate.
- Easy to implement from scratch or from an existing system, which requires a clear methodology for implementation. Integration of the model within a system ignoring data life cycle should provide optimization of these systems as well.
- The implementation should have scalable performances and minimum performance overhead over existing systems.

2.2 Active Data at a glance

Our response to the above requirements is Active Data: a programming model that allows programmers to observe data during their life cycle. Programmers supply code to the Active Data runtime, specifying at what step of a data life cycles the code must be executed. At the same time, programs that make data progress through their life cycles, when using or altering data, must inform the runtime. Any data item has a corresponding life cycle model that documents everything that can be done with it.

Modeling life cycles Thus at the root of Active Data is a way to model life cycles. Life cycle models are based on Petri Networks [23]. Petri Networks are a formalism and a graphical tool widely used for the analysis of systems with concurrency and resource sharing. Each data item is tagged with a state, corresponding to its current stage in its life cycle; we represent all possible data states with Places. Petri Net Transitions represent all possible operations on data that changes their state. As it is common for distributed systems to deal with data replicas, a Petri Net token represents a single replica of the same data item. Several tokens on different places represent the individual states of several data replicas distributed on different nodes.

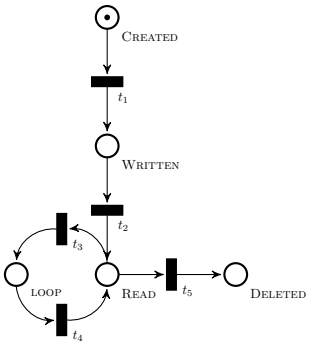


Figure 1: Active Data model for the *"write-once, read many"* life cycle.

A Petri Net represents a model, and different data items can share a single model; such case implies that the same operations can be performed with them. Moreover, because readability is essential, a single Petri Net visual representation only shows the replicas of the same data item.

As a first example, we modeled a regular "write once, read many" life cycle with Active Data, as shown in figure 1. In this model, a data item starts in the CREATED place. The only data replica, represented by a single token, can be written only once. From the READ place, the token can loop through t_3 and t_4. This implies that the data item can be read multiple times. Eventually, the token can pass through t_5 and finnish its life cycle on the DELETED place.

This theoretical basis enables us to express more complicated things, like the composition of several Active Data models, as detailed in section 3.

Programming applications The most important concept in the above subsection is the one of *data transitions*. Transitions in an Active Data model give precious information. They tell programmers what actions on data are to be expected, and where they can *plug* code to be executed.

The Active Data runtime offers a client API that allows programmers to supply code to be executed when data transitions are triggered. The supplied code is called a *handler*. The usual course of events that leads to code execution is as follow:

1. The programmer writes and compiles code into an Active Data handler;
2. The programmer use the client API on machine A to register the handler for a specific transition t;
3. A remote program uses or alters a data item on machine B and changes its state, matching transition t;
4. The remote program reports to the runtime that transition t occurred;
5. The user provided code is run on machine A.

The last two points are repeated as long a the handler subscription remains, every time the transition is triggered, on any data, on any node manipulating data.

Architecture The paradigm used by Active Data to propagate transitions is based on Publish/Subscribe [11]. In our implementation, every node can be publisher and subscriber at the same time. Clients of the Active Data runtime publish transitions to a centralized service called *Active Data Service*. Clients pull transition information from the Active Data Service as well. In this paper we preferably use the word "client" to refer to software using the Active Data client API, and "node" to refer to a machine running such software.

Transition handlers considerations It is possible for programmers to subscribe to several transitions, with one or more handlers. Thus, handlers can be shared by subscriptions. Handlers are executed serially on each subscribing machine, in the order they were published. Moreover, transition handlers can be stateless or stateful.

2.3 Example

With the write once, read many model we defined above, we present a short code sample that illustrates how to automatically perform additional treatment when data get written in the

system. Whenever transition t_2 is triggered, i.e. an arbitrary node wrote data, we want a specific node to compute the file's md5 sum and write it in a file.

The programmer first creates a handler that contains the necessary code:

```
TransitionHandler md5Handler = new TransitionHandler() {
   public void handler(ActiveData data, String transitionName, boolean isLocal) {
      MessageDigest md = MessageDigest.getInstance("MD5");
      String path = getPath(data.getId());
      InputStream input = new FileInputStream(path);
      byte buffer[] = new byte[2048];

      int n = 0;
      while((n = input.read(buffer)) > 0)
         md.update(buffer, 0, n);

      byte[] digest = md.digest();
      BigInteger bigInt = new BigInteger(1,digest);
      String hash = bigInt.toString(16);
      while(hash.length() < 32 )
        hash = "0" + hash;

      OutputStream output = new FileOutputStream(path + ".md5");
      output.write(hash.getBytes());
      output.close();
  }
};
```

Listing 1: md5 sum transition handler

The handler in listing 1 is a Java object that implements the `TransitionHandler` interface. The first argument provides information about the data that was written; we use it to get the file path. Then we compute the file's md5 sum and write it to a new file.

The second argument is the name of the transition that was triggered; the last argument indicates whether the transition was triggered on the same node. Here we do the same thing whether the transition was triggered locally or remotely.

The programmer further passes the code to the Active Data runtime, specifying it should be run after transition t_2 was triggered:

```
client.subscribeTo(t2, md5Handler);
```

2.4 Application scope

The Active Data programming model offers the opportunity to develop a broad range of applications covering a wide range of scenarios. However the scope and the methodology differs whether the data life cycle is known a priori or has to be defined, and if Active Data is to be implemented or not. We now review some of the application domains according to the implementation of Active Data.

- Active Data is implemented for a particular data management software, such as a file storage, a data-flow scheduler or a file transfer service. In this case, the set of transitions is known by the programmer, so they can express their program as a set of transition handlers implementing data operations. For instance, the implementation of Active Data for BitDew (see Section 4.2) would allow to program a wide range of DLM applications such as backup system, distributed checkpoint servers, collective file operation (scatter/gather, alltoall), data-intense applications, execution runtimes such as MapReduce or Allpairs, automated-tiered storage systems etc.

- Particular data management system which lacks DLM features. Active Data can be implemented after an analysis of the system to extract the data life cycle. This would provide

either additional programming functionalities, such as in the previous case, or permit specific optimizations to this system.

- Users also have the possibility to implement from scratch their data life cycle based on their own needs, that is without any data management substrate. In this case, the application is expressed as set of operations on data and the developer takes care of generating the transition whenever operations are actually performed. In this case, the benefit of using Active Data is to have a clear specification of the data life cycle.

3 Life Cycle Model

In this section we first give notations and formal definitions for Active Data models, based on Petri Nets. Then, we define how to compose a unified Active Data model from various models of heterogeneous legacy systems that were not designed to collaborate.

Notations A Petri Net is 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 requires and how many token it produces.

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, it is generally omitted in the visual representation.

Data identifiers and state A single Petri Net represents the life cycle model of a single data item, with as many replicas as tokens. To identify data replicas, we tag each token with a triplet $\delta_{(id,i,p)}$ where id is a unique data identifier, identical for all the replica of a single data item, i is the replica number and $p \in P$ is a place. We elaborate by stating that the state S_δ of data item δ is the set of all of the data replicas individual states distributed over the system: $S_\delta = \{S_{\delta_{id},1}, S_{\delta_{id},2}, \ldots, S_{\delta_{id},r}\}$ where r, called the distributivity of data items, is the number of replica of data δ. Informally, this implies that several data replicas may live at the same time on several nodes, but each replica progresses through its own life cycle independently.

A Transition is a change that occurs when a data state is updated, i.e. a transition is the function $T : S_\delta \longrightarrow S'_\delta$ where S_δ and S'_δ are respectively the previous and the new state of δ.

Data creation, replication, and deletion We now focus on different stages in the life cycle that require special attention. As stated before, a data item is created as a token in the CREATED place and labelled with the triplet $(id, 1, \text{CREATED})$. The CREATED state has the special property that no transition have an outgoing arc pointing to that place, which can be formulated as $\forall t \in T, \nexists (t,p) \in (T \times P) \mid p = \text{CREATED}$.

To create additional data replicas, we use a pattern similar to the one presented in Figure 2. From the place P1, a single token fires the transition t_1. This causes two tokens to be present

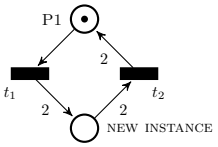


Figure 2: Data instance creation.

on the place NEW INSTANCE, because the weight of the outgoing arc from t_1 is 2. Finally, the old and new tokens, the later representing the new data replica, move back to the P1 place so that additional data replicas can be created.

Deletion of data is represented using a special state labelled DELETED. This state has two special properties: *i)* it has no outgoing arc, i.e. $e\forall t \in T, \nexists (p,t) \in (P \times T) \mid p = \text{DELETED}$ and *ii)* when it is not empty, every transition that is not connected to DELETED is disabled. The last property implies that once a token has reached the deleted place, every other token must proceed to the deleted place. In the system, it implies that data life cycle management operations on data that have a replicate in the deleted state are illegal.

Proposition 1. Data life cycle termination *We say that a data item* terminates *its life cycle represented by a Petri Net, when all of its tokens reach the* DELETED *place.*

Proof. A data item being deleted implies that all data replicas should be deleted as well i.e., all tokens with the same *id* must reach the DELETED place. We use an extension of Petri Nets called inhibitor arcs; when a transition is connected to a place by an inhibitor arc, it is disabled whenever the place contains one or more tokens. The DELETETED place is connected to every transition by inhibitor arcs, except transitions already leading to DELETED. Let G be the set of inhibitor arcs for PN:

$$\forall t \in T, \text{ if } (t, \text{DELETED}) \notin F, \text{ then } \exists (\text{DELETED}, t) \in G \tag{1}$$

Once a token reaches the DELETED place, all the other transitions are disabled, and all remaining tokens are forced to transit to the DELETED place. We make sure that tokens can always transit to the deleted place by adding the necessary transitions:

$$\forall p \in P, \exists t \in T | P \neq \text{DELETED} \wedge (p,t) \in F \wedge (t, \text{DELETED}) \in F \tag{2}$$

□

Because readability is essential for Active Data, inhibitor arcs and transitions added by equations 1 and 2 are usually not visually represented.

Composing data life cycles Large distributed applications often involve several unrelated middleware and tools that were not designed to work together. Different tools treat data in different ways and as such create different life cycles. Obtaining a high-level unified view of data in the various layers that compose the whole system is a valuable asset for developing and maintaining it. Active Data offers such a view and the ability to programmatically react to data transitions transparently across Petri Nets. Here we describe how to associate and compose different life cycle models to enable distinct Petri Nets to contain tokens with identical *id*.

Let us define $P_s \subset P$ a set of special places, called *start places*. A start place is linked to a set of Petri Nets; we note $C(p)$ the set of Petri Nets linked to the start place p. We further consider

two independent life cycles A and B, each with their own Petri Net representation, respectively PN_A and PN_B. Then, we define two functions A *starts* B and A *stops* B; when applied to a start place $p \in P_{S,A}$, and a Petri Net $PN \in C(p)$, A *starts* B creates in B a replica of the data item in A, if p is not empty. A new token placed in the start place of B represents the new replica; this token is labeled with the quadruplet (id_A, id_B, r_B, q_B) where id_A is the data identifier in the Petri Net A, id_B, r_B and q_B are respectively the new identifier, token number, and place in the Petri Net B. Conversely, calling the function A *stops* B adds a token in the DELETED place of B; this function is automatically called when A terminates. Informally, this implies that any life cycle created from A must terminate when A terminates. We note (A, B) the life cycle composed of life cycles A and B. We say that A is the *parent* life cycle of B, and B is a *child* life cycle of A.

Definition 1. Composing data life cycles *We say that a data life cycle model PN_A represented by a Petri Net is* composed with *a life cycle model PN_B when $\exists p \in P_{S,A} | PN_B \in C(p)$.*

Composition can be symmetric, i.e. there can be two start places p and p' such as: $p \in P_A | PN_B \in C(p)$ and $p' \in P_B | PN_A \in C(p')$. In such case, any of the two composed life cycles can terminate the other.

Proposition 2. Composed life cycles termination
We say that a composed life cycle (A, B) represented by two Petri Nets PN_A and PN_B terminates *when A and B terminate.*

Proof. Let us examine the situation where A is composed with B. If A is terminated, a token is present in the DELETED place of A and the function A *stops* B has been called. This implies that a new token has been placed in the DELETED place of B, causing B to terminate. Therefore the composed life cycle (A, B) terminates. If B is also composed with A, the composed life cycle terminates when either A or B terminates. □

4 Implementation

Here we give more details on aspects that have been mentioned previously. We also discuss particular implementation considerations.

4.1 The Active Data API

Active Data provides a Java API to construct models and interface with the Active Data runtime to publish and subscribe to transitions.

Defining a model Programmers must build an object-oriented copy of their life cycle models in order to use them with Active Data. The API provides classes that matches the model vocabulary: `Place`, `StartPlace`, `Transition` and `Arc`.

Instantiating a life cycle From the constructed model, the programmer can create life cycles. As such, a life cycle model can be seen as a mold for life cycles. The programmer creates a `LifeCycle` object for each domain-specific data object that must be integrated in Active Data with a unique identifier and a link to a model.

Publishing transitions Publishing each transition is a necessary step for Active Data to work correctly. This is done by a single call to the client API that relays the publication to the Active Data runtime. For error checking matters, publishing a transition may fail. Thus, it is the programmer's responsibility to either actually trigger a transition after it was successfully published, or to be able to undo a transition that occurred if its publication fails. We discuss error checking in more details later in this section.

Subscribing to transitions For more flexibility, Active Data provides two main ways to subscribe to transitions: *i*) subscribe to run code when a specific transition is triggered on any life cycle or *ii*) subscribe to run code when a specific transition is triggered on a specific life cycle. Listing 1 demonstrates how to implement a transition handler.

A third form of subscription does not run code; instead it tells the Active Data runtime to automatically update a local life cycle object to reflect the global Petri Net that is maintained by the Active Data service. With this subscription, the client receives all transitions regarding this life cycle and replays them in the order they were published to the service. This enables clients to *see* how all the tokens are laid out on the places and to react to the global state data.

Subscribing to data creation Active Data by default enables to react to data creation, but data life cycles always start with a start place that contains at least one token. As Petri Nets do not allow transitions with no incoming arcs, handlers should not be able to react to data creation. To address this issue, a phony transition called CREATE is automatically inserted in every `ActiveDataModel` upon construction. This transition leads to the model's start place. When a new life cycle is integrated in Active Data, the `ActiveDataClient` triggers and publishes the CREATE transition.

Error checking Another valuable feature of the Active Data runtime implementation is that it is able to dynamically enforce the model. The Active Data service will throw an exception whenever a client tries to:

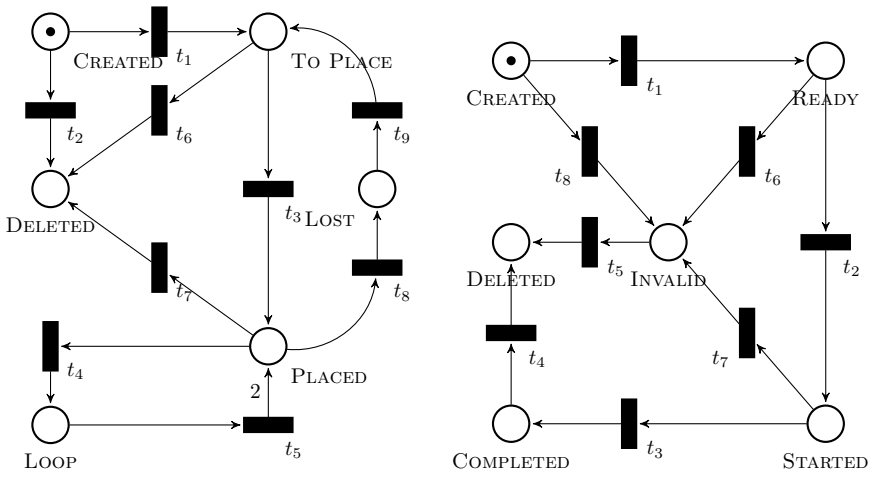
- publish a transition that is not in the model, ensuring the new state assigned to a data item is valid with respect to the previous state;
- publish a transition that is not enabled, knowing about the position of every token in the system;
- publish a transition for a life cycle that has terminated.

4.2 Active Data for BitDew

In this sub-section we apply the concepts presented in the paper to BitDew [12], a distributed data management framework. We show Active Data models are able to express the most complex features of BitDew.

BitDew is a middleware developed by INRIA for easy data management on various distributed infrastructures; it offers a programmable environment for automatic and transparent data operations.

BitDew relies on a set of data *attributes* to drive key data management operations such as indexing, distribution, placement, replication and fault-tolerance, with a high level of abstraction. The BitDew framework has a flexible service-based architecture that integrates modular P2P components such as a distributed data catalog based on DHTs, collaborative transport protocols for data distribution and asynchronous and reliable multi-protocol transfers.



(a) Data model (b) Transfer model

Figure 3: The two Active Data models made from the observation of BitDew. One regards only data, and the other file transfers.

Because of all these features, BitDew creates complicated data life cycles that make it a good case study for Active Data. In the remaining of this section we explain how we modeled BitDew with Active Data and discuss some particular aspect required to carry out the integration of Active Data in BitDew.

Data life cycle model The first step to integrate Active Data in BitDew is to observe how data are created, treated and deleted by the framework; from these observation we are able to construct BitDew's data life cycle. In the documentation and source code of BitDew, we observe that user data and metadata are managed by a class `Data`; this class contains a `status` property that holds the data item state. From the enumeration of possible states, we deduce the places of our model. We further examine the code to determine, each time a `Data` object status is updated, what the old and new status were; this indicates where to add transitions in the model.

The result of our observation is the model shown in figure 3a; A `Data` objects naturally starts in the Created place. If the user requests BitDew to place the data on a client node, it is moved to the To Place state, and then further to the Placed place when the request has been satisfied. If replication is requested, the `Data` object is placed on an additional client, a token goes through the loop, resulting in an additional token. If fault tolerance was requested, whenever a client on which a `Data` was placed disappears, BitDew triggers t_8 and moves the data item to the Lost place so it, will be re-placed on another client node. For readability, transitions from lost and loop to deleted that are required by the model are not represented in the figure.

File transfer life cycle model BitDew offers a single API for various file transfer protocols such as HTTP, FTP, BitTorrent etc. As `Data` objects in BitDew initially only hold metadata, BitDew's "Put" and "Get" operations are used respectively to add a payload to a `Data` and to read its payload. Put and get requests return `Transfer` objects that are used to manage file transfers. Like `Data objects`, they contain a status (with a different set of possible values) and

hold a reference to the `Data` being transfered. The file transfer life cycle model is presented in figure 3b.

Data identifiers To link `Data` objects managed by BitDew and life cycle objects managed by Active Data, we need common identifiers. Active Data makes this easy by accepting collections of strings as identifiers for a life cycle. BitDew's `Data` objects contain a string identifier that is unique amongst all objects managed by BitDew; we allocate this identifier to Active Data data life cycles. `Transfer` objects also contain a unique identifier and, as stated above, a reference to a `Data` object; we allocate both the `Data` and `Transfer` identifiers to transfer life cycles.

Placement and replication Part of the complexity of the data life cycle in BitDew comes from the Data Scheduler that places data on clients. A data item being placed on a client is equivalent to the creation of a new replica. We represent replication with the loop $\{\text{PLACED}, t_4, \text{LOOP}, t_5\}$ state; the loop creates an additional token every time a token traverses it.

Fault tolerance BitDew deals with many faults, especially in the context of Desktop Grids. In particular it must deal with host churn. When a data item is placed on a node, and the node disappears from the system, the data item is marked with a LOST state and is placed on an other node. This is represented by the loop $\{\text{PLACED}, t_8, \text{LOST}, t_9, \text{TO PLACE}\}$ in figure 3a.

File Transfer The implementation of reliable file transfers is another source of complexity, and transfers have a life cycle of their own. The class `Transfer` contains a state, and the possible states for a transfer is different as of the possible states for the `Data` class. Transfers' life cycle are represented by the Petri Net in Figure 3b. A `Transfer` object can be made for any existing data. A `Transfer` object always contains a reference to the `Data` object being transferred.

Composition of File Transfer and Data Scheduler In BitDew the Data Scheduler and file transfers 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 place as explained in section 3. In BitDew, a new file transfer can be started for a `Data` object in any state, except DELETED, LOST and LOOP. To represent this, we define all the places but the 3 mentioned above as start places and connect them to the transfer life cycle model.

From this model, we instrument BitDew's code to publish transitions anywhere an operation altering data is performed.

5 Case studies

We conduct three case studies to evaluate the ability of 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.

- 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 triggered during file transfers. 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 the programmer to have a unified view of data distributed across systems or infrastructures. To illustrate this, we present how to create a life cycle model based on the Linux Kernel extension `inotify` which notifies applications of local filesystem modifications. Based on this "distributed inotify", 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, such as a data set that dynamically grows or 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 with dynamic data. We modify an existing MapReduce implementation [26] so that it incrementally updates the result of a MapReduce computation when a subset of the input data is modified.

All experiments have been performed using the Grid'5000 experimental platform [5].

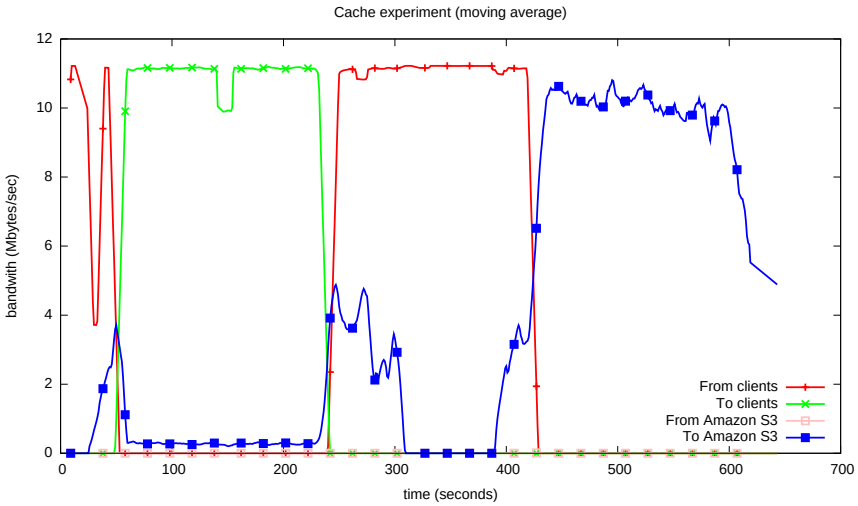
5.1 Storage cache

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 transitions. Storage caches are widely used by scientific applications to minimize cost, network bandwidth, latency and energy consumption.

We consider a cache between a computer infrastructure and the Amazon Simple Storage Service [1] (S3). S3 provides extensible cloud storage with high redundancy. S3 users pay according to the storage space used, the number of put and get requests performed and the amount of data transferred 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 decrease the S3 usage cost. 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 Active Data on BitDew and exploits the File Transfer life cycle.

Clients are connected to the cache application that runs on a local server node and uses a fixed portion of its local storage. 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 in order to have a durable copy of each data written in the cache. The cache application can be expressed with only two transfer transitions: t_1 (when a transfer begins) and t_4 (when a transfer ends). The algorithm reads as follow: *i*) after a client performs a put in the cache, the transition handler transfers the same data item to Amazon S3 and possibly deletes local data according to the cache eviction policy; *ii*) when a client begins a get request from the cache, the transition handler checks if the data item is in the cache. If the data item is already in the cache (*cache hit*), the handler serves it from the cache; if the data item is not in the cache (*cache miss*), the handler gets it from Amazon S3 and then serves the data item from the cache.

We evaluate the cache with a scenario which mimics a master/worker computation involving 10 clients, a 5Gb cache server and the Amazon S3 service. The master first transfers three files to the cache; they are to be distributed to all the clients:



(a) Bandwidth measured on the cache server in bytes per seconds

	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
Dollars	0.3	0.53	0.23

(b) Data transfers measured from and to Amazon S3 with and without a cache

Figure 4: Cache experiment evaluation

1. a 200Mb program to be run by all client nodes;
2. a 50Mb data-set;
3. a 100Mb data-set.

Once the files are available, each client downloads the program. The clients are evenly divided in two sets depending on their rank. Clients in the first set download the smallest data-set, the others download the largest.

To illustrate the cache behavior, we plot the network traffic between the cache and the clients and between the cache and Amazon S3 in figure 4a. We observe that the cache behaves as expected: data coming from the clients to the cache (*put*) are transferred to Amazon S3; most data to the clients come from the cache without querying Amazon S3. The two traffic peaks from the clients to the cache for $x \in [50, 130]$ correspond to the distribution of the program and the two data-sets. After the first burst of data sent to the clients, traffic appears from the cache to Amazon S3. From $x = 120$ to $x = 310$, clients download the program and a data-set from the cache, and in the meantime there is few traffic to Amazon S3. Then clients execute the program and return the result: from $x = 310$ to $x = 500$ we observe transfers from the clients to the cache, and again transfers from the cache to Amazon S3, due to the write-through policy.

Table 4b shows that using the storage cache avoided performing 1.9 Gb of unnecessary data transfers from Amazon S3, cutting the costs by 43%.

5.2 Collaborative Sensor Network

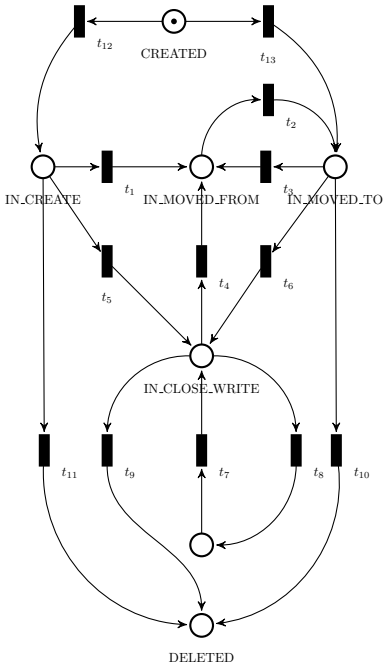


Figure 5: life cycle of a file in terms on notify events.

This case study illustrates: *i*) the adaptability to legacy data management systems, and *ii*) the ability to develop distributed applications that support independent data life cycles distributed over several local systems.

It is a common practice for applications acquiring data – for example from a sensor network – to apply some pre-processing before being pushed on 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 scripted using ad-hoc languages or programs. Data throttling is also a common practice to reduce the amount of data injected in the system at a given time. Decentralize data throttling enables to reduce the load on the system by dropping data before they are injected. However, it requires coordination between the sensors, which can be made easier when expressed with Active Data.

Here 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. Then the resulting JPEG files are 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.

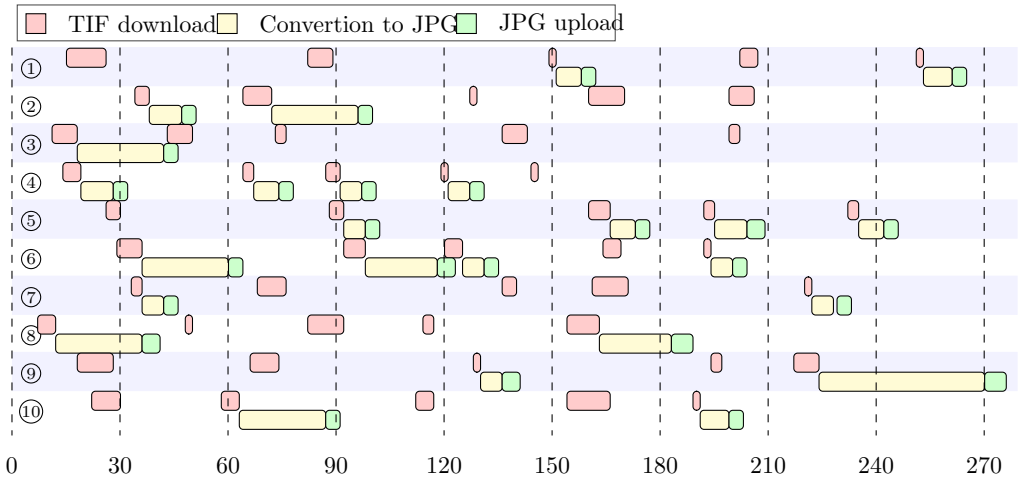


Figure 6: Collaborative network of 10 sensors: the x axis plots the time in seconds, for a window size $w = 30$ seconds.

Files on nodes' filesystems are dynamic data and can be represented with an Active Data model. 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. Then, programs can be expressed in terms of file transitions. To capture these transitions on files, we use the inotify Linux kernel subsystem [20]. 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 inotify events with an Active Data model. Figure 5 presents the inotify Active Data model, constructed using the method described in section 4.2. The combination of Active Data and inotify creates a *distributed inotify*: all nodes can now coordinate based on transitions happening on other nodes' filesystems.

Now that sensor nodes can react to remote filesystem transitions, we can express our problem in terms of Active Data transitions. Nodes locally run a program that reads inotify events from their Linux kernel and publishes the corresponding Active Data transition to all the other nodes. Each node also independently runs a program to react to two types of Active Data transitions:

- t_{12}: we check if the transition is local or remote: if it is remote, and if the associated file is a JPEG image, it implies that a TIFF image has been pre-processed on a remote sensor and we increment the counter p.
- t_5: if the transition is local, we check the associated file type: if it is TIFF, we compare p to n and pre-process the file only if $p < n$.

Every w seconds, each node sets its counter p to 0.

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 Active Data handlers for $n = 3$ and $w = 30$ *seconds*.

Figure 6 presents the Gantt chart of the scenario which lasts 279 seconds, where each numbered pair of lines represent the activity of one sensor. Red bars plot data acquisition times, yellow bars plot data pre-processing and the green bars plot the upload time of pre-processed

data. 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 have been downloaded on the nodes, but only 3 have been pre-processed. The other images have been dropped.

5.3 Incremental MapReduce

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

One of the strongest limitations of MapReduce 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 the MapReduce framework becomes *aware* of the life cycle of the data involved, it can dynamically adapt the computation to data modification.

We consider the MapReduce implementation made on top of the BitDew storage system [26]. In this implementation, a master node places the input data chunks in the BitDew storage and launches a MapReduce execution, whose map and reduce tasks are respectively executed by mappers and reducers. However, input data can be updated directly in the storage by external applications. To make the MapReduce implementation incremental, we simply add a "dirty" flag to the input data chunks. When a chunk is flagged 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's dirty flag, we need the master and the mappers to react to transitions in the life cycle of the chunks. More precisely, nodes listen to two transitions triggered by the storage system, thanks to Active Data:

- t_3 is observed by the master node. When this transition is triggered, 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 the storage system before launching the job. If both conditions are true, the transition handler flags the chunk as dirty.

- t_1 is observed by the mappers. When this transition is fired, mappers check whether the transfer is distant and if one of their input chunks is modified. In this case, the transition handler on the mapper flags the chunk as dirty.

To evaluate the performance of incremental MapReduce, we compare the time to process the full data set compared with the time to update the result after modifying a part of the dataset. 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 chucks. Table 1 presents the time to update the result with respect to the original computation time when a varying fraction of the dataset is modified. As expected, the less the dataset 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

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

Table 1: Incremental MapReduce: time to update the result compared with the fraction of the dataset modified.

significant speedup with a patch that impacts less than 2% of the MapReduce runtime source code.

6 Related Work

In this section, we review existing programming models for data-centric and distributed applications as well as popular distributed storage systems and argue that they are inadequate for supporting DLM applications.

The Big Data challenge has renewed the approach to parallel programming in many aspects. MapReduce [8], introduced by Google in 2004, is a good example of this quest for new paradigms to simplify the processing and generation of large data sets. MapReduce borrows from functional programming, where a programmer can define both a Map task that maps a data set into another data set, and a Reduce task that combines intermediate outputs into a final result. Although MapReduce was originally developed for use by web enterprises in large data-centers, this technique has gained a lot of attention from the scientific community for its applicability in large parallel data analysis (including geographic, high energy physics, genomics, ...). In the wake of MapReduce, several other data-centric languages have emerged, either as alternative, e.g, Dryad [17] for dataflow parallel computing, Allpairs [6] to perform massive pair-wise comparisons in large data sets, Swift [32] to script and automate the manipulation of large parallel scientific dataflow, or as evolution of the paradigm, e.g., PigLatin [9] to provide high level query interface on top of MapReduce, Twister [10], a framework for iterative MapReduce computations, to cite a few of them.

Because data sets are dynamic, i.e., may grow or shrink in time, or be partly modified, there is a need for frameworks which adapt to data change and are in particular optimized for incremental computation, i.e. where a single change in the data set does not trigger the whole computation re-execution. Percolator [24] presents a programming model and an implementation for incrementally process multi-petabytes of continuously mutating data sets. This model relies on events: when a data is updated, a workflow is implicitly created and it triggers a method that updates depending data. Nephele [4] and MapReduce-Online [7] are two frameworks specialized for parallel processing of large data-streams. Chimera [13] addresses the problem of complex data flows in scientific applications by defining data as derivations of other data. By keeping track of data history and dependencies, Chimera allows to know how data have been computed in order to reproduce them, to re-compute a set of data when its dependancies have been updated, and to re-compute locally data from their meta-data instead of transferring them whenever this would be more efficient. Although, these languages outline the necessity of high level data-centric paradigms, none of them explicitly address the key issues associated with DLM.

Phoenix [27] is an inspiring example of a parallel programming model which takes as a foundation principle the characteristics of the infrastructure. To address the problem of highly dynamic environments, where nodes can join and leave the computation at any time, Phoenix constructs a set of virtual nodes, on top of the physical nodes, where computations communicate through a message passing semantic. Event-based programming is a paradigm which is strongly

gaining in popularity, as witnesses the emergence of mature or innovative frameworks such as Mace [19], libasync [21] or recently Incontext [33]. However, such systems lack the abstractions specific to data-intense computing that would allow to mix advanced data operations and data processing.

There exist countless DLM systems where Active Data could be implemented aside, but systems which provide high level interface to data management would be the one that would benefit the most. We list some of them and give hints about possible improvement. Data attributes is a feature proposed by BitDew which allows the user to specify data behaviour such as fault-tolerance, replication, file transfer protocol or affinity placement. The implementation of Active Data and BitDew [12] together tha we have proposed in this paper would allow the programmer to take advantage of the BitDew data attribute to wisely control and steer the distribution of data, while using Active Data to program applications that could react on each event happening during the distribution of the data set. Similarly, Chirp [28] is aimed at data intensive applications in computational grids. It provides a flexible file system that can be run and accessed by any user without kernel changes or other administrator privileges. MosaStore [30] is a file system optimized for the main collective file pattern operations (gather/scatter, reduce, broadcast) that can be found in workflows. Combined with MosaStore, Active Data would allow to implement a workflow system that takes advantage of these features.

7 Conclusion

We described Active Data, a programming model for supporting data life cycle management applications. The starting point is the formal definition of the data life cycle either within existing system or defined by end-user application. The Active Data execution runtime system permits user-provided code to be executed at each stage of the data life cycle (creation, replication, transfer, deletion). An implementation of the model with BitDew has been presented, which provides both guidelines for future implementation with other systems, but also a functional prototype allowing to program life cycle transitions-based applications. Several applications studies have been described, demonstrating that this model is a general model that will facilitate the development of complex applications to manage large, dynamic and distributed data sets. Applications investigated, although developed in few dozens lines of codes, already demonstrate significant performance and cost improvement. Scenarios have also outlined more specific features of Active Data: a unified view of data across heterogeneous systems, low overhead implementation, API to publish transitions and execute transition handlers, ability to plug-in into existing systems.

Future works will focus on several aspects. The model can be extended on the following directions: advanced representation of computations that would investigate consumption and production of data items; representation of collection of data items that would allow collective operations on data sets. Concerning the implementation of Active Data, we plan to investigate rollback mechanisms for fault-tolerant execution of applications and evaluate distributed implementations of the publish/subscribe substrate. Finally, several application prototypes are being developed using Active Data: a MapReduce runtime which mixes low power mobile devices (tablets, set-top boxes, smartphones) and online Cloud storage [2], a distributed and cooperative content delivery network to distribute virtual appliance images embedding large HEP applications to Internet Desktop Grid resources [15, 29, 18] and a distributed network of checkpoint image server featuring server selection using network distance [25].

References

[1] Amazon simple storage service. `http://aws.amazon.com/s3/`, 2010.

[2] Gabriel Antoniu, Julien Bigot, Cristophe Blanchet, Luc Boug, Franois Briant, Franck Cappello, Alexandru Costan, Frdric Desprez, Gilles Fedak, Sylvain Gault, Kate Keahey, Bogdan Nicolae, Christian Prez, Anthony Simonet, Frdric Suter, Bing Tang, and Raphael Terreux. Towards Scalable Data Management for Map-Reduce-based Data-Intensive Applications on Cloud and Hybrid Infrastructures. In *The 1st International IBM Cloud Academy Conference (ICA CON 2012)*, North Carolina, USA, April 2012.

[3] LA Barroso, J Dean, and U Holzle. Web search for a planet: The google cluster architecture. *IEEE MICRO*, 23(2):22–28, MAR-APR 2003.

[4] Odej Kao Bjorn Lohrmann, Daniel Warneke. Massively-parallel stream processing under qos constraints with nephele. In *HPDC'12, ACM Symposium on High PErformance PArallel and Distributed Computing*, Delft, Nederland, June 2012.

[5] Raphael Bolze and all. Grid5000: A large scale highly reconfigurable experimental grid testbed. *International Journal on High Peerformance Computing and Applications*, 2006.

[6] J. Bulosan, D. Thain, and P.J. Flynn. All-pairs: An abstraction for data-intensive cloud computing. In *International Symposium on Parallel and Distributed Processing*, Miami, FL, USA, 2008.

[7] 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*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51:107–113, January 2008.

[9] Jeffrey Dean and Sanjay Ghemawatta. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008.

[10] 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.

[11] 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.

[12] Gilles Fedak, Haiwu He, and Franck Cappello. Bitdew: A data management and distribution service with multi-protocol file transfer and metadata abstraction. *Journal of Network and Computer Applications*, 32(5):961 – 975, 2009.

[13] Ian Foster, Jens Vöckler, Michael Wilde, and Yong Zhao. Chimera: Avirtual data system for representing, querying, and automating data derivation. *Scientific and Statistical Database Management, International Conference on*, 0:37, 2002.

[14] Geoffrey Fox, Tony Hey, and Anne Trefethen. Where does all the data come from? Technical report, Indiana University, November 2011.

[15] Haiwu He, Gilles Fedak, Peter Kacsuk, Zoltan Farkas, Zoltan Balaton, Oleg Lodygensky, Etienne Urbah, Gabriel Caillat, and Filipe Araujo. Extending the EGEE Grid with XtremWebHEP Desktop Grids. In *Proceedings of CCGRID'10, 4th Workshop on Desktop Grids and Volunteer Computing Systems (PCGrid 2010)*, pages 685–690, Melbourne, Australia, May 2010.

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

[17] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conferenceon Computer Systems 2007*, New York, NY, USA, 2007. ACM.

[18] P. Kacsuk, Z. Farkas, and G. Fedak. Towards Making BOINC and EGEE Interoperable. In *Proceedings of 4th IEEE International Conference on e-Science (e-Science 2008), International Grid Interoperability and Interoperation Workshop 2008 (IGIIW 2008)*, pages 478–484, Indianapolis, USA, December 2008.

[19] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: language support for building distributed systems. In *In PLDI'07: Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation*, New York, USA, 2007.

[20] Robert Love. Kernel korner: intro to inotify. *Linux Journal*, 2005(139):8–, November 2005.

[21] David Mazieres. A toolkit for user-level file systems. In *Proceedings of USENIX Annual Technical Conference*, Berkeley, CA, USA, 2001.

[22] Rene Meier and Vinny Cahill. Taxonomy of distributed event-based programming systems. *The Computer Journal*, 48(5):602–626, 2005.

[23] Tadao Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, pages 541–580, April 1989.

[24] 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.

[25] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-aware overlay construction and server selection. In *INFOCOM 2002. Conference of the IEEE Computer and Communications Societies. Proceedings.*, 2002.

[26] Bing Tang, Mircea Moca, Stphane 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.

[27] Kenjiro Taura, Kenji Kaneda, Toshio Endo, and Akinori Yonezawa. Phoenix: a parallel programming model for accommodating dynamically joining/leaving resources. *SIGPLAN Notice*, 38:216–229, June 2003.

[28] Douglas Thain, Christopher Moretti, and Jeffrey Hemmes. Chirp: a practical global filesystem for cluster and grid computing. *Journal of Grid Computing*, 7:51–72, 2009.

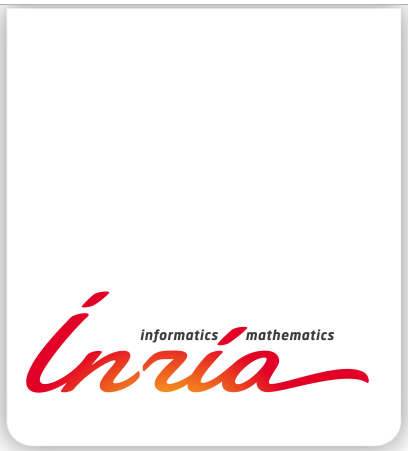
[29] E. Urbah, P. Kacsuk, Z. Farkas, G. Fedak, G. Kecskemeti, O. Lodygensky, A. Marosi, Z. Balaton, G. Caillat, G. Gombas, A. Kornafeld, J. Kovacs, H. He, and R. Lovas. EDGeS: Bridging EGEE to BOINC and XtremWeb. *Journal of Grid Computing*, 7(3):335–354, September 2009.

[30] Emalayan Vairavanathan, Samer Al-Kiswany, Lauro Costa, Zhao Zhang, Daniel Katz, Michael Wilde, and Matei Ripeanu. A workflow-aware storage system: An opportunity study. In *12th International Symposium on Clusters, Cloud, and Grid Computing (CCGrid'12)*, Ottawa, Canada, 2012.

[31] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. *SIGARCH Computing Architectures News*, 20:256–266, April 1992.

[32] Michael Wilde, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz, and Ian Foster. Swift: A language for distributed parallel scripting parallel computing. *Parallel Computing*, 2011.

[33] Sunghwan Yoo, Hyojeong Lee, Charles Killian, and Milind Kulkarni. Incontext: simple parallelism for distributed applications. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 97–108. ACM, 2011.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399