Active Data: A Programming Model to Manage Data Life Cycle Across Heterogeneous Systems and Infrastructures

Anthony Simonet, Gilles Fedak, Matei Ripeanu

Active Data: A Programming Model to Manage Data Life Cycle Across Heterogeneous Systems and Infrastructures

Anthony Simonet[a,**], Gilles Fedak[a,*], Matei Ripeanu[b]

[a] *Inria, University of Lyon, LIP – ENS Lyon, 46 allée d'Italie, 69007 Lyon, France*
[b] *Kaiser Building, The University of British Columbia, Vancouver BC V6T 1Z4, Canada*

Abstract

The Big Data challenge consists in managing, storing, analyzing and visualizing these huge and ever growing data sets 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, etc. Indeed, data-intensive applications span over a large variety of devices and e-infrastructures which implies that many systems are involved in data management and processing. We propose Active Data, a programming model to automate and improve the expressiveness of data management applications. We first define the concept of data life cycle and introduce a formal model that allow to expose data life cycle across heterogeneous systems and infrastructures. The Active Data programming model allows code execution at each stage of the data life cycle: 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 four use cases: a storage cache to Amazon-S3, a cooperative sensor network, an incremental implementation of the MapReduce programming model and automated data provenance tracking across heterogeneous systems. Altogether, these scenarios illustrate the adequateness of the model to program applications that manage distributed and dynamic data sets. We also show that applications that do not leverage on data life cycle can still

*Corresponding author
**Principal corresponding author
Email addresses: anthony.simonet@inria.fr (Anthony Simonet), gilles.fedak@inria.fr (Gilles Fedak), matei@ece.ubc.ca (Matei Ripeanu)

benefit from Active Data to improve their performances.

Keywords: Parallel Programming Model, Distributed and Heterogeneous Systems, Data Life Cycle

---

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 [1]. 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 [2].

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 (DLCM)*, i.e. the various operations performed on data: transfer, archiving, replication, processing, deletion... One can observe two constraints influencing data life cycles. The first one is due to users and applications that explicitly express operations on data. For example, a common pattern for the production of scientific data is the sequence which consists of the following steps: acquisition by a scientific instrument, pre-processing to reduce the size of the data and storage for further analysis. 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 sequences 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. Applications or systems also have the initiative to create or delete data replica in order to optimize the data placement in term of locality, bandwidth access, security or reliability, which in turn

increases the complexity of the life cycle. Finally, unpredictable events such as failures may alter the life cycle as well. 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 DLCM systems.

To alleviate the complexity of data life cycles, 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 and models have emerged, such as MapReduce [4] or Dryad [5], that offer simple yet high-level data-centric parallel interfaces. However these languages focus more on processing large data sets than on 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 [6] 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 modifications of the data sets. The last challenge relies on data integration, i.e. 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 get a complete end-to-end view of data life cycles across systems and infrastructures, integrating the many copies of the same data under a single namespace. This would allow cross-system optimizations by enabling 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 [7] 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 formal model for distributed data life cycles and a programming model to allow code execution at each stage of the data life cycle. On the one hand, the formal model allows to describe the life cycle of distributed data; the model can be shared with other scientists to build DLCM applications, dynamically check that an execution is valid in regards to the model and integrate dispersed data under a unique namespace. On the other hand, the programing model borrows from Active Message the idea of executing user-provided code when certain events occur (message reception in the case of Active Message [8]). 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 DLCM 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 workflows, logs processing and monitoring and so forth.

Our contribution is the following: we first present a formalism inspired by Petri Networks allowing to expose data life cycles in heterogeneous distributed systems. We show that this model is able to capture the main stages of data life cycles, 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, on its integration with data management systems and on performance evaluation, namely throughput, latency and overhead of the system. To evaluate Active Data, we present four use cases that illustrate the versatility of the framework to program DLCM applications targeted at distributed and dynamic data sets. The first one shows how to program a local cache to a remote storage service in few lines of code allowing both performance improvement and cost reduction. The second scenario features a network of distributed sensors that cooperate to implement data acquisition throttling. The third example shows that an existing MapReduce runtime augmented with Active Data can turn into an incremental MapReduce. The last use-case features data provenance tracking across heterogeneous systems. We run these scenarios using the experimental platform Grid'5000 [9] and report on the experiments.

The rest of this paper is organized as follows. We introduce the Active Data programming model and life cycle model in sections 2 and 3 respectively. The execution runtime system is described in section 4, and experiments in section 5. We discuss related works in section 6, and in section 7 we conclude and open discussions about further topics of interest.

2. Active Data Programming Model

In this section, we introduce 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 a 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; they leave the system when they are physically erased, or when they are moved to a storage outside of the system. Between these two points in time, data progress through a serie of different stages of development, e.g. migration, duplication, archive, transfer and so forth. We call *data life cycle model* a formal representation of all the possible states and all the valid state transitions of a data item, when handled by a particular system or by an user application, e.g. created, duplicated, deleted, backed-up. *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 formal and infrastructure agnostic model to describe data life cycles in distributed systems and a programming model which facilitates data life cycle management for developers. We detail the requirements for the life cycle model, the programming model and its implementation:

- *Allow to reason about data life cycle:* the formal model must capture the essential life cycle stages and properties: creation, deletion, faults, replication and error checking. It must allow system developers to share the data life cycle model as documentation with their users.

- *Allow coordination between multiple systems in different data centers:* Datasets are heavily distributed, making collaboration between sites and systems necessary, even when those were not designed to collaborate.

- *Allow to reason about data sets handled by heterogeneous softwares:* The multiple systems involved in the management and the processing of datasets may not have been designed to collaborate (this is quite often the case with legacy software). The data management system must allow to *integrate* the multiple copies and identifiers of the same data concurrently living in various systems.

- *Allow data management systems to be extensible:* Data life cycles in scientific applications are not fixed in time, in particular because datasets can be used years after they have been acquired or generated. The data life cycle management system must be extensible in order to handle every future treatments that scientists may want to execute.

- *Incremental processing:* Many scientific and commercial applications treat streams of data, or datasets that are often partially modified. It

is most of the time unnecessary to run a complete processing workflow again to incorporate new data and update the results. The data life cycle management system must offer a granularity of tasks fine enough to allow treating newly arrived data only, and a system to allow users to discriminate the updated data from the rest of the data.

- *Simplify the programming of applications that implement data life cycle management:* This involves: *i*) giving applications the ability to operate on data, *ii*) informing applications about data life cycle progression and *iii*) making their development easy with implicit parallelism.

- *Be easy to implement from scratch and to integrate to an existing system:* This requires a clear methodology for system developers to construct the life cycle model of their systems. The data management system must also make it easy for scientists to implementation data management tasks. Integrating the model within a system ignoring data life cycles should provide optimization of these systems as well.

- *Have scalable performances and minimum performance overhead over existing systems:* The data life cycle management system must not impair the normal system operations. This property is also essential to encourage the adoption of the system.

2.2. Active Data at a glance

Our response to the above requirements is Active Data, a formal model, a programming model and a runtime environment, which allow to program applications by specifying for each step of the data life cycle, the code that will be executed. In this subsection we give a first overview of Active Data and a first example, before detailing every aspect of the system in Section 3 and 4.

In a nutshell, Active Data works as follows: data management systems expose their intrinsic data life cycle according to a well-specified formalism. We consider each creation, modification or deletion of a data item by the data management system as a progression of the data item through its life cycle. We call *transition* the move of a data item from one state to another state.

Informally, the programming model proposed by Active Data could be called of *transition-based* programing model. A programmer provides a routine or code that we call *transition handler* which is executed whenever a transition is triggered. In other words, when a transition happens to a data

item, a message (or event) is sent to the other nodes of the system to notify them about the transition and this causes the transition handler code to be executed. The paradigm used by Active Data to propagate transitions is based on Publish/Subscribe [10]. Data management systems publish transitions to a centralized service called *Active Data Service*.

Thus, at the root of Active Data is a representation of data life cycles. We base our model on Petri Networks [11], which is a formalism and a graphical tool widely used for the analysis of systems with concurrency and resource sharing. Petri Nets can depict very intuitively data life cycles: *Places*, represented by circles are the states of the life cycle; *Transitions*, represented by rectangles are the operations that happen on data items; *Tokens*, represented by • in Places, are data items in a particular state of the life cycle. It is common for distributed systems to deal with data replicas. Each data replica is represented by a single Petri Net token. Several tokens on different places represent the individual states of several data replicas distributed on different nodes.

2.3. Example

To illustrate a data life cycle model, Figure 1 shows a representation of the "Write-Once, Read-Many" life cycle. A data item starts its life in the CREATED place, then is written once (WRITTEN place), may be read once or several times (loop between the READ place and t_3 transition) and finally deleted (TERMINATED place). In this example, the transition t_1 corresponds to the action of writing data, t_3 is triggered when reading data, and t_4 corresponds to a data item deletion.

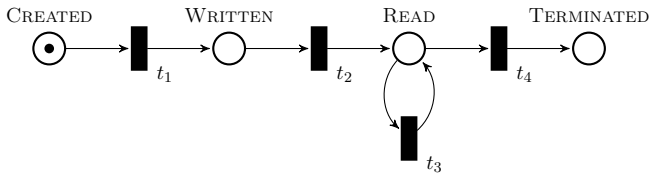


Figure 1: Representation of the *"Write-Once, Read-Many"* data life cycle.

We present a short code sample that illustrates how to automatically perform curation when data get written in the system. Whenever transition t_2 is triggered, the application creates a md5 signature file.

The programmer first writes the handler that contains the checksum computation:

```
TransitionHandler md5Handler = new TransitionHandler() {
   public void handler(Transition transition, bool isLocal, Token[] ↘
       →inTokens, Token[] outTokens) {
      MessageDigest md = MessageDigest.getInstance("MD5");
      String path = getPath(inTokens[0].getUid());
      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 `handler()` method is invoked by the system when the transition is triggered. The first argument is the transition that was triggered; the second 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 last two arguments provides information about the tokens that went through the transition; we use them to get to the file path of the data item that was written. Then we compute the file's md5 sum and write it to a new file.

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);
```

Now, every time a process somewhere triggers the transition t_2 for any data item, the handler will be executed asynchronously and the signature file will be created.

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, our approach differs from (and complements) existing workflow and dataflow systems [12]; Table 1 summarizes the main differences between Active Data,

	Workflows	Dataflows	Active Data
Data centric	No	Yes	Yes
Execution model	Task sequence	Data dependency	Event based
Control flow	Explicit	Implicit	None
Monitoring	Task completion	Data production	Data state change
System integration	Difficult (ad-hoc solutions)		Easy
Integration responsibility	Workflow/dataflow developer or user		System developer

Table 1: Comparison of Active Data with workflow and dataflow systems

workflows and dataflows using six criteria: *1)* whether the approach is data centric; *2)* how the sequence of activities is determined; *3)* how activities are chained and how control is passed from one to the next; *4)* what events are usually monitored in such approaches; *5)* how easily different system representations can be integrated; and *6)* who is responsible for integrating an external system (or tool), for example in the case of Active Data this is the responsibility of the system developer to make a system "Active Data enabled".

Moreover, the scope and the methodology for application development differs whether the data life cycle is known a priori or has to be defined. We now review some of the application domains of Active Data.

- Active Data is integrated into 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 4.3) would allow to program a wide range of DLCM 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 systems which lacks DLCM features. Active Data can be integrated 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. Data Life Cycle Metamodel

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.

In this section we first give our notations and formal definitions for data life cycle models and present several developments: i) decoration of tokens to allow unique identification of data replicas, ii) data life cycle termination rules, that allow to detect errors and illegitimate operations and iii) how Active Data exposes a unified view of data life cycles even though they involve several heterogeneous systems.

3.1. 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 metamodel, 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, the state of a data item that is distributed in a system matches the marking on its Petri Network.

3.2. Definitions

We now formally define our life cycle metamodel by extending Petri Networks.

Definition 1. *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: $\textsc{Created} \in P$ is the start place, representing the creation state of the data item; $\textsc{Terminated} \in P$ is the end place, representing the state of the data after it has been permanently deleted. In addition, Terminated is a sink:

$$\forall t \in T, \nexists (p,t) \in (P \times T) \mid p = \textsc{Terminated} \tag{1}$$

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:

$$M_0(p) = \begin{cases} 1 & if\ p = \textsc{Created} \\ 0 & if\ p \in P \setminus \{\textsc{Created}\} \end{cases} \tag{2}$$

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. Thus they are neither represented graphically nor exposed to users.

3.3. 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 identifiers that links to the real-life data.

Systems discriminate data items they manage by assigning a unique identifier to each of them. 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 [13] assigns a property named `R_DATA_ID` of type `Long` to each file; HDFS [14] 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).

Definition 2. *The* data identifier *is the word that uniquely identifies a data item in a given system. The* replica identifier *is the word that uniquely identifies all the replicas of a single data item.*

When a data management systems uses replication, it must discriminate the several replicas of the same data item in addition to discriminating different data items. 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 (see Definition 2).

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.

Now, we can adjust our metamodel to assign the two identifiers we discussed to tokens. 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 metamodel 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 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 2. 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 2, transition t_2 consumes one token and produces none (it has no outgoing arc). Thus firing it effectively removes a token from p_1.

3.4. 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.

On one system, the end of a life cycle is represented in the life cycle model with a token on the TERMINATED place. 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:

$$\forall t \in T,\ \exists(\textsc{Terminated}, t) \in G \tag{3}$$

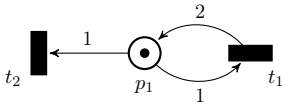


Figure 2: Creation and deletion of data replicas. t_1 creates a new replicas and t_2 deletes one.

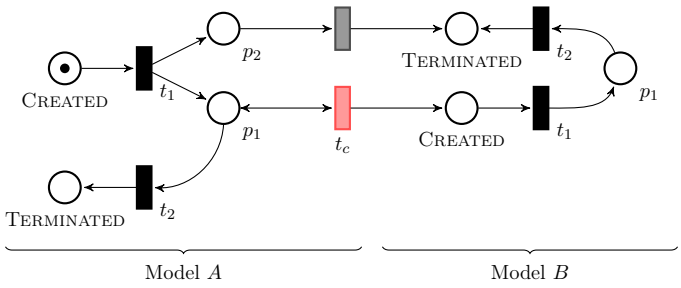


Figure 3: Example of life cycle model composition: a token in model A can be consumed by t_c on start place p_1 in order to start a life cycle in B; an other token can be consumed on place p_2 by t_t to stop the life cycle in B. Both life cycle models are valid on their own.

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.

3.5. Life Cycle Composition

Until now, our metamodel only allows to represent the life cycle of data in a single system. In order to represent the complete life cycle of data when it involves several systems, 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.

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 features a first example of composition.

Definition 3. *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:*

- $P = P_A \cup P_B$

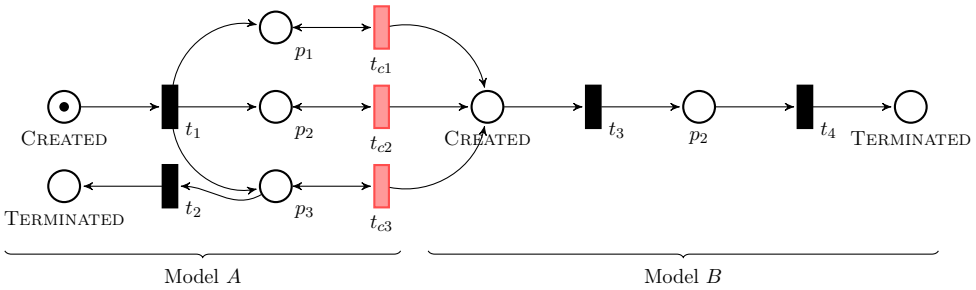


Figure 4: Example of life cycle model composition: any of the three composition transitions (in red) can create a new life cycle in B from a life cycle in A. The choice of the place and composition transition used to create the new life cycle represent meaningful information to the users.

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

$T_c(A, B)$ *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 3 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 4. 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.

This definition is very flexible as it allows a life cycle model to be composed with several other models, and even itself; the former case allowing data derivation.

Definition 4. *We say that a life cycle model A is* composed *with another life cycle model B iff* $T_c(A,B) \neq \emptyset \wedge \forall t \in T_c(A,B)\ \exists\ (p,t) \in F(A,B) \wedge (t, Created_B) \mid p \in P_A$.

Composition and Token Identification. According to the definition given in subsection 3.3, 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 $\delta(id,i,p)$ is actually a set of identifiers. Figure 5 illustrates how the set of identifiers is maintained in both the already existing and the newly created token when a composition takes place.

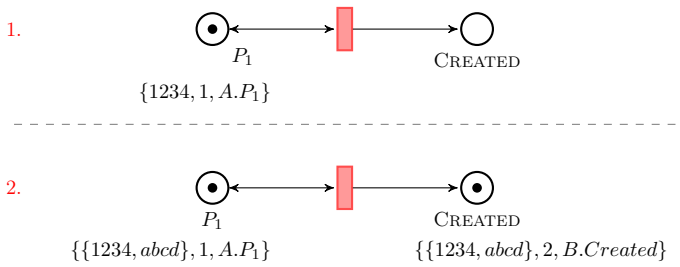


Figure 5: Composition process: the figure presents the state of a portion of life cycle model around a composition transition. Before the composition transition is triggered (1.), a token is present on the transition's source place (P_1); after the transition is triggered (2.), a new token is present 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.

This way of identifying tokens also offers a strong link between copies of the same data distributed in non-cooperative systems. Thanks to the composition transition, it is now possible for a system to have a view of the whole data life cycle, beyond the scope of the system.

4. System Design

Active Data is composed of two parts: *i*) the Active Data Service stores and manages life cycles, receives transition publications and inform clients

of transitions they have subscribed to; and *ii*) the client API that executes code in reaction to transitions and allows systems to inform the Service of operations they apply on data (transitions). In this section we discuss the implementation and integration of the Active Data programming model.

4.1. Client Application Programming Interface

We first describe how system developers and users can use Active Data through its Application Programming Interface (API).

4.1.1. Constructing a new life cycle model

The first step for using Active Data is to construct an object-oriented representation of the life cycle of data managed by a system. The root of the model is a class called `LifeCycleModel`; objects of this class link to `Place` and `Transition` objects. Constructing a `LifeCycleModel` requires to decide of a unique system identifier (SID) for the system. This class has methods for adding places and transitions, that also require names.

```
LifeCycleModel myModel = new LifeCycleModel("My System");
Place p1 = myModel.addPlace("not executed");
Place p2 = myModel.addPlace("executed");
Transition t = myModel.addTransition("execute");

myModel.addArc(p1, t);
myModel.addArc(t, p2);
```

4.1.2. Publishing a new life cycle

Active Data users 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.

All applications participating in the life cycle of a data item must create `LifeCycle` objects to link the files they manipulate to Active Data. After that, the service saves the state of the life cycles and is able to receive transition publications regarding them. 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.

```
ActiveDataClient client = ActiveDataClient.getInstance();
LifeCycle lc = client.createAndPublishLifeCycle(myModel, "xyz");
```

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();
```

4.1.3. Publishing transitions

Newly created life cycles have only one token on their CREATED place. Applications must move to other places to reflect the progress of data items through their life cycle.

All applications 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. Publishing a transition is a single call:

```
LifeCycle lc = client.createAndPublishLifeCycle(myModel, "xyz");
...
Transition t = myModel.getTransition("My system.t");
client.publishTransition(t, lc);
```

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 in the sense that it returns immediately, regardless of whether someone has subscribed to the transition or not.

In more complex cases, the transition has several input and/or output places; in this case (such as the one depicted on Figure 6), the publishing code must specify: *i*) What tokens are consumed, and from which input places; *ii*) 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 and implements the `doDeal(LifeCycle, Transition)` method. When implementing the dealer, the publisher has access to three primitives, which indicate how tokens are consumed and produced:

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

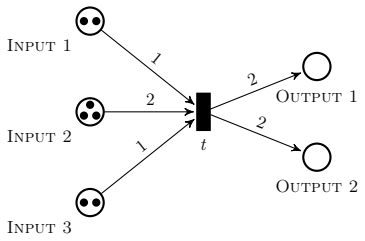


Figure 6: Transition requiring the publishing process to specify which input tokens are consumed, and how they are laid out on the output places.

`produce(Token, Place)` indicates 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 dealer —which gets a chance to examine every place and token and decide which to pick— is called internally by Active Data on the client side 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. The behavior of the default transition dealer is not defined for complex cases such as the one we are discussing; it is only guaranteed to consume the right number of tokens from the input places, and to produce the right number of tokens 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.

4.1.4. Publishing a composition transition

To inform Active Data of data passing from one system to another, a process (usually the one that initiates the operation) must publish a composition transition every time a data item is sent to an other system.

Publishing a composition transition is similar to publishing a regular transition, except it requires an additional argument: the unique identifier of the

```
// Publish the new life cycle and transition for Model A
String uid = "1234";
LifeCycle lcA = client.createAndPublishLifeCycle(modelA, uid);
...
Transition t1 = modelA.getTransition("Model A.t1");
client.publishTransition(t1, lcA);

// Effectively send a the data object from A to B, getting the new id from B
String newId = sendDataToB(...)

// Publish the composition transition with Model B
Place p2 = modelA.getPlace("Model A.p2");
Token token = lcA.getTokens(p2).get(0);
Transition comp = modelA.getTransition("Model A.tc");

client.publishCompositionTransition(comp, token, newId);
```

Listing 2: Publication of a composition transition. The identifier for the new token is provided by the destination system, then passed to Active Data.

data item in the new system. Listing 2 demonstrates how to publish the composition transition on figure 3.

Here, the composition transition consumes the first token present on the p_2 place. This token's unique identifier is "1234"; the publishing process initiates the transfer of the data item from A to B. B returns the identifier it has locally assigned to the data item. When the publishing process publishes the composition transition with this identifier, Active Data creates a new token on the CREATED place of system B.

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.

4.1.5. Transition handlers

Transition handlers are the code provided by the programmer, that are to be executed whenever DLC transitions happen. A first transition handler was presented in Listing 1.

A handler is an object that implements the `TransitionHandler` interface and 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.

The handler is able to examines the tokens consumed and produced by the transition, to extract their unique identifier and uses them to access the real data. 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 can be stateful. For example, a user may not want to be notified every time a transition is published; instead they might want the handler to notify them each n times the handler is executed instead.

4.1.6. Transition subscription

The transition handlers must be attached to a transition to be executed after the 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. However, users usually want their handlers to be executed only when it is relevant to them, 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 the example handlers in Listings 1, we notice that the value of the `Transition` argument is never tested; in these cases it is not needed, because these handlers are meant to be subscribed to a single transition only.

There is a second type of subscription that allows to be notified of every transition, but only for a particular life cycle:

```
|| TransitionHandler handler = new TransitionHandler() {
```

```
    public void handler(Transition transition, bool isLocal, Token[] ↘
        →inTokens, Token[] outTokens) {
.........
    }
};

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

4.1.7. Querying Life Cycles

The `LifeCycle` class allows to examine all places, transitions and tokens for any data item. Active Data offers a querying interface that allows any programmer to obtain the complete life cycle of any data item, no matter where their code is running.

Users query a life cycle with a couple (SID, UID). Through composition, a life cycle is identified by as many such couples as there are systems in the life cycle. However, only one is necessary, allowing code with a very partial view expand it to the whole life cycle.

For example, consider a storage system composed with an execution system. A client of the storage system would like to discover the state of the computation in which a particular file is involved. The client knows that the identifier of the storage system is "storage" and that locally, the identifier of their file is, for example, the integer `36253`.

Listing 3 shows how the client can use Active Data to discover *i*) the identifier of their file in the execution system, and *ii*) the state of the application.

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

ActiveDataClient client = ActiveDataClient.getInstance();
LifeCycle lifeCycle = client.getLifeCycle(storageSID, storageUID);

for(Place p: modelB.getPlaces())
   for(Token t: lifeCycle.getTokens(p))
      print("Token with id " + t.getUid() + " is on place " + p.getName();
```

Listing 3: Querying the complete life cycle of a data item from a partial knowledge. Active Data returns the complete `LifeCycle` object with all the tokens in every system.

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 process.

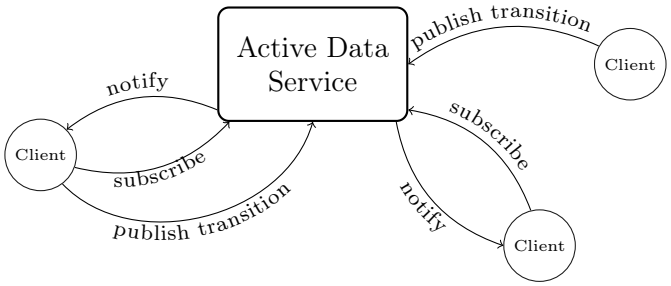


Figure 7: Architecture of Active Data

4.2. System Architecture

The architecture of Active Data is illustrated by Figure 7; it is composed of the centralized Active Data Service and of clients communicating with the service using the client API.

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. The fact that the service is centralized allows it to easily maintain the consistency of life cycles.

Clients of the Active Data Service are of 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 code.

4.2.1. Active Data Service

The Active Data programing model being based on two main operations that are publishing transitions and subscribing handlers, it seems natural to use the Publish/Subscribe paradigm [10] 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. 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.

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. 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. However, the 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.

4.2.2. Execution model

The client regularly pulls events from the service, that contain the transitions published, and the corresponding life cycle. Once the transitions list is received, the client executes the associated handlers. The handlers are run serially, 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 therefore, it is recommended to perform any lengthy operation in a separate thread. In the case when a handler would take a long time to run, it would not block the entire system, but only the local execution queue.

4.2.3. Verification

Detecting incorrect behavior of processes in large distributed systems is very difficult in nature. 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, or runtime verification. Every time a client attempts to publish a transition through the API, the Active Data Service checks the request against the life cycle 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.

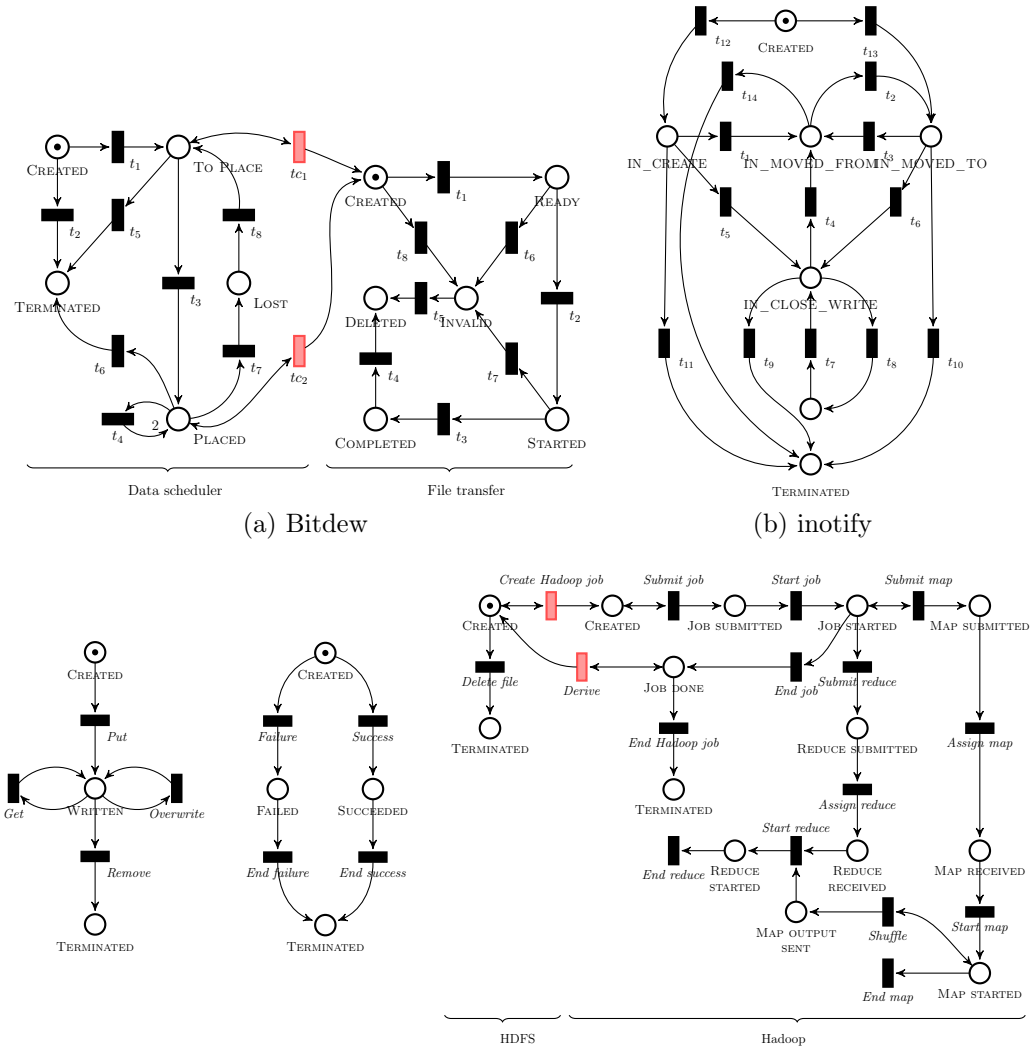
4.2.4. Consistency

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.

4.3. Systems Integration

We report now on the integration of Active Data with five widely used data management systems. BitDew [15] 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 [16] allows to *watch* a directory and receive events about the files it contains, such as creation, modification, write, movement and deletion. iRODS [13] 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 [17] 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 [18] is a project maintained by the Apache Software Foundation that includes a widely used implementation of the MapReduce [4] programming model and a distributed file system called HDFS [14].

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



(a) Bitdew (b) inotify

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

Figure 8: Data life cycle models for five data management system.

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 8d 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 5.3.4). This model is represented in Figure 8c.

Conversely, two data life cycles are complete: inotify, BitDew. Figure 8b 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 8a). 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 3.5. 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 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 8e 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.

5. Experimental Evaluation

In this section we report on experimental evaluation of the prototype using micro-benchmarks and use-case scenarios. Experiments are run on a cluster of the Grid'5000 experimental platform [9] composed of 92 2-CPU nodes. Each CPU 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 2.6.32.

5.1. Performance Evaluation

To evaluate the performance in terms of throughput, latency and overhead, we conduct a set of benchmarks based on the version 0.1.2 of the

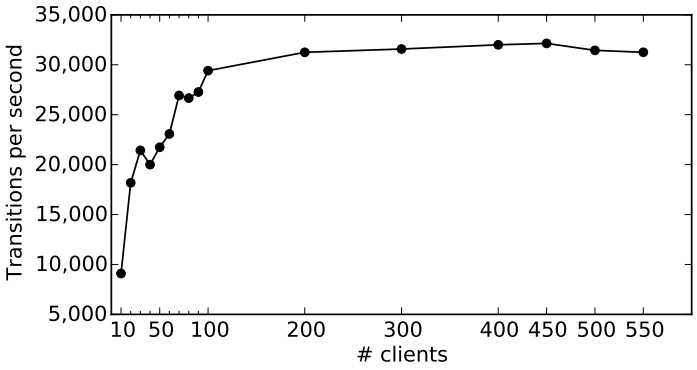


Figure 9: Average number of transitions per second handled by the Active Data Service

prototype[1].

Throughput is measured as the number of transitions that Active Data is able to handle per second. In order to stress the system we run one Active Data server and a varying number of clients (one per cluster node), which publish 10,000 transitions in a loop without a pause between the iterations. Figure 9 plots the average number of transitions processed per second against the number of clients. Once saturated, the Active Data Service can handle up to 32,000 transitions per second.

We evaluate the latency, i.e the time to create of a new life cycle and to publish a transition, and the overhead, i.e the additional time incurred by the Active Data runtime environment. We use the BitDew file transfer operation as a reference to measure the overhead. The experiment consists in creating and uploading 1,000 files (1KB to stress the system) in a single burst. 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 2 shows the median, 90^{th} centile and standard deviation for the latency in milliseconds, when the service and the client run on the same node (local) and on two different machines (Ether). The overhead is given in seconds and as a percentage compared with the vanilla BitDew. We can observe that latency 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 our prototype

[1]Active Data is free software, available under the GPL licence http://active-data.gforge.inria.fr

		med	90^{th} centile	std dev
Latency	Local	0.77 ms	0.81 ms	18.68 ms
	Eth.	1.25 ms	1.45 ms	12.97 ms
Overhead	Eth.	w/o AD	with AD	
		38.04 s	40.6 s (4.6%)	

Table 2: Latency in milliseconds for life cycle creation and transition publication and overhead measured using BitDew file transfers with and without Active Data.

implementation performs well enough to provide reactivity and scalability, and is fully able to handle the case studies presented in the next section.

5.2. 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 dataset. Active Data must be able to handle the many transition publications induced by the map and reduce tasks.

We execute the Terasort benchmark on a cluster of 45 nodes, each having two 4-core Intel Xeon L5520 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.

The Terasort benchmark runs in two phases: a random dataset 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 10 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.

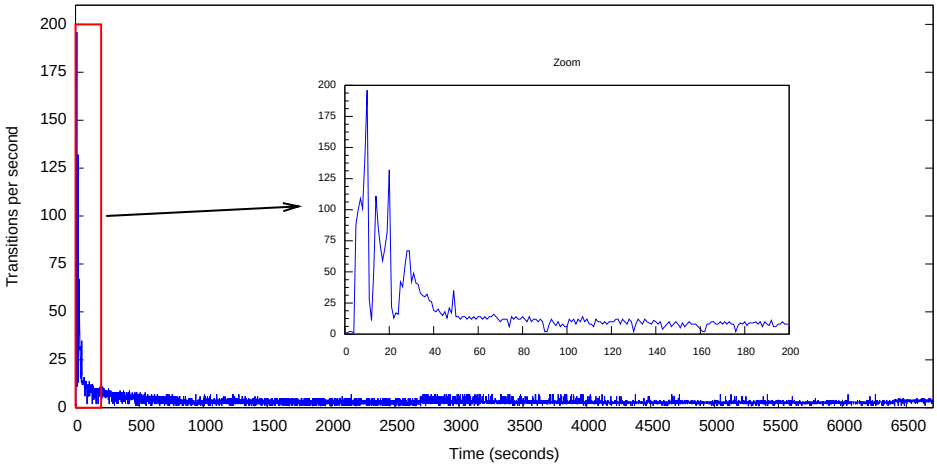


Figure 10: Number of transitions published each second during the Hadoop Terasort execution.

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.

5.3. Case studies

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.

- 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 manage data sets distributed across systems or infrastructures. To illustrate this, we implement a "distributed" `inotify` thanks to Active Data. 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 [19] so that it incrementally updates the result of a MapReduce computation 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 Globus Online. Thanks to this model, 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.

5.3.1. 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 [20] (S3). 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 decreases 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 and BitDew and exploits the File Transfer life cycle in Figure 8a.

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 to the cache. The cache application can be expressed with only two transfer transitions:

- t_1 (transfer begins) is observed by the cache. If the handler detects the transfer is a get from the cache, it checks if the data item is in the cache. If it is (*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.

- t_9 (transfer ends) is observed from the cache as well. If the handler detects it is a put in the cache, it transfers the same data item to Amazon S3; local data can be deleted according to the cache eviction policy.

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 Amazon S3: a 200MB program to be run by all client nodes and 2 input data-sets (50MB and 100MB). Once the files are available, each client downloads the program. Half of the clients download the smallest data-set, the others download the largest. Table 3 shows that using the storage cache avoided performing 1.9 Gb of unnecessary data transfers from Amazon S3, cutting the costs by 43%.

This scenario illustrate the ability to rapidly prototype data management application with Active Data: the source code is less than 100 lines of code

	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

Table 3: Cache experiment evaluation

and can be developed in a day. It is distributed and yet requires no synchronization, no thread spawning and no forking.

5.3.2. 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 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 possible 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.

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 these 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 inotify events with an Active Data model. Figure 8b presents the inotify Active Data model, constructed using the method described in section 4.3. 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, then a TIFF image has been pre-processed on a remote sensor and we increment the local 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 11 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.

This scenario illustrate a powerful feature : Active Data can easily turn into a distributed system, any local system that is able to expose its local

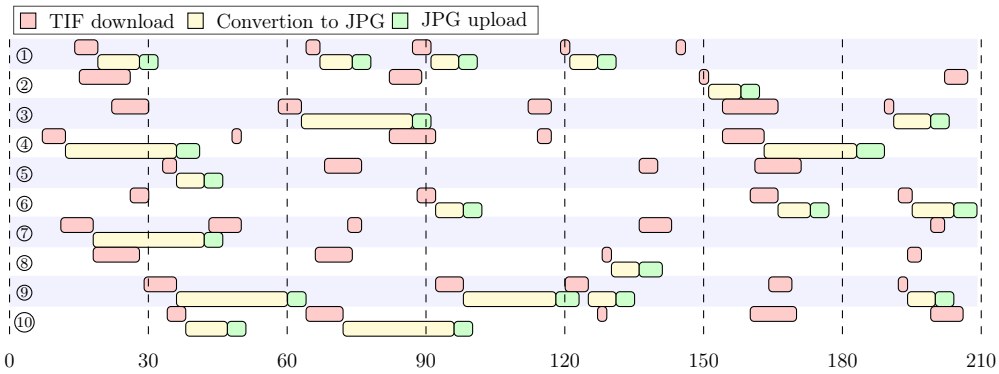


Figure 11: Collaborative network of 10 sensors: the x axis plots the time in seconds, for a window size $w = 30$ seconds (illustrated by dashed lines).

data life cycle. It also demonstrates the ease with which distributed data life cycle management tasks can be expressed with Active Data.

5.3.3. 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 [19]. 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

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

Table 4: Incremental MapReduce: time to update the result compared with the fraction of the dataset modified.

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 chunks. Table 4 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 how the system was optimized to reach a significant speedup with a patch that impacts less than 2% of the MapReduce runtime source code.

5.3.4. Data Provenance

Data provenance constitutes the complete history of derivations and treatments throughout data life cycle, and is essential to preserve the quality of scientific data asset over time. Reconstructing this history gets very complicated when multiple data management systems and infrastructures cooperate. It is even more challenging when the data management systems involved are loosely coupled or do not communicate at all, because we lose the causality between data management operations.

Research on data provenance is heading towards automated provenance collection and Provenance-Aware Storage Systems (PASS) [21]. A PASS is a storage system that records and maintains provenance automatically, based on the analysis of client requests. A strong limit of PASS is that the provenance information they provide is bound to what the storage system sees of data life cycles.

We consider a scenario where Active Data receives events regarding data evolving in two systems completely independent. 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).

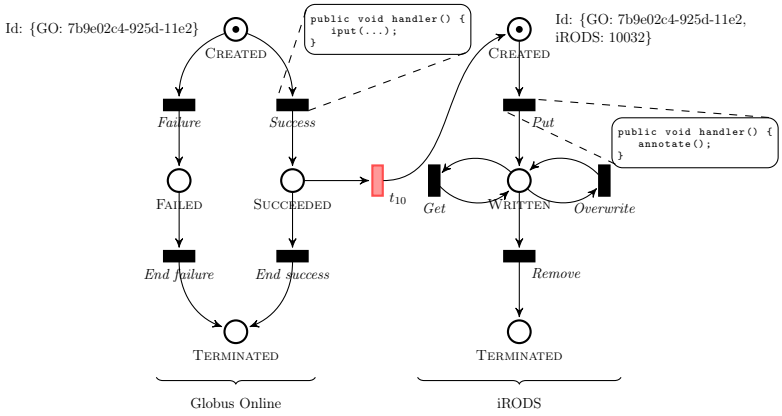


Figure 12: Composition of Globus Online and iRODS: the figure presents the two life cycle models connected by a composition transition as well as the token identifiers and transition handlers.

In our scenario, for any data file in iRODS, we want to record file transfer provenance: the transfer endpoints, start and completion times, transfer failures. Active Data is the glue that enables both Globus Online 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 8d represents the life cycles of data in Globus Online and iRODS. When a Globus Online file transfer starts, a transfer task is created and the returned `Task Id` becomes the token identifier.

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

A second transition handler is attached to iRODS's *Put* transition: it is executed when any iRODS data is written the first time, right after its creation. This handler requests the life cycle from the Active Data Service to see if it contains a Globus Online identifier. In such case, it uses the token identifier to query Globus Online and get file transfer information.

To demonstrate our solution, file transfers are launched from a remote Globus Online 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 Online `Task Id` and meta-data information (endpoint, completion date, request time). Listing 4 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: Metadata associated to an iRODS data file transferred with Globus Online

The global and unique namespace provided by Active Data of datasets 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.

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 DLCM applications.

The Big Data challenge has renewed the approach to parallel programming in many aspects. MapReduce [4], 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. In the wake of MapReduce, several other data-centric languages have emerged, either as alternative, e.g, Dryad [5] for dataflow parallel computing, Allpairs [22] to perform massive pair-wise comparisons in large data sets, or as evolution of the paradigm, e.g., PigLatin [23] to provide high level query interface on top of MapReduce, Twister [24], 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 [6] 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 [25] and MapReduce-Online [26] are two frameworks specialized for parallel processing of large data-streams. Chimera [27] 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 DLCM.

The complexity of handling Big Data often necessitates to assemble several class of heterogeneous infrastructures. For instance, [28, 29, 30] investigate several options to efficiently deliver data to Cloud, Grids and Desktop Grids infrastructures using P2P systems. More recently, several works have investigated the possibility to execute MapReduce applications on Clouds and Desktop Clouds [31, 32]. Our preliminary work around Active Data [33, 34] have explored issue of representing data sets when there are distributed on hybrid infrastructures.

There have been few works around the concept of Data Life Cycle to improve the management of e-infrastructures. In [35], 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 [36], 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 [37] leverages the DLC concept to provide description of application-specific storage requirement and use this information for storage planning and data deployment on Cloud infrastructures.

Scientific workflow systems such as Taverna [38], Galaxy [39], and LONI [40] have been used in many domains including bioinformatics, genomics, astronomy, and medical imaging. Scientific workflows allow developers to describe and coordinate their multi-step computational tasks, for example data acquisition, data conversion, and data analysis. Swift [41] allows developers to script and automate the manipulation of large parallel scientific dataflows. Pegasus [42] targets multiple kinds of infrastructures, such as Desktops, campus Clusters, Grids, and Clouds. With Pegasus, scientists develop workflows in high level terms without having to deal with the details and the particularities of the underlying resource management middleware (Condor, Globus, or Amazon EC2). These systems allow researchers to orchestrate a series of tools and services into a cohesive workflow while the workflow system handles the flow of data between tools. In most cases the workflow is described as a directed acyclic graph (DAG), where the nodes are tasks and the edges denote the task dependencies. In contrast to Active Data, workflow users must manually "wrap" tools and services before they can be used in a workflow and they typically focus only on the analysis components of the workflow.

Phoenix [43] 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 [44], libasync [45] or recently Incontext [46]. However, such systems lack the abstractions specific to data-intense computing that would allow to mix advanced data operations

and data processing.

An emerging research challenge in e-Science is the question of data provenance, i.e. maintaining the historical and causal record of data derivation [47]. Two issues when establishing data provenance are relevant to our problematic. The first one is the question of the representation of data lineage. Several approaches have been proposed in the literature [48], which eventually lead to standardization: the Open Provenance model for instance, which is a form of annotated causality graph [49]. The second question is how to gather provenance information? One approach, illustrated by the Provenance Aware Storage System [21, 50] is to capture events from the storage system so that operations on files can be recorded and stored as meta-data annotations. Although these works outline the necessity for investigating formal representations of data-sets, our context differs and objectives diverge as well. We want to capture the state of a dataset when it is deployed on the entire infrastructure as well as all legitimate possible future operations.

There exist countless DLCM systems where Active Data could be implemented aside, but systems that provide high level interfaces to data management would be the ones 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 [15] together that 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 [51] 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 [52] 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 systems or defined by end-user applications. 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 and runtime system have been presented, along with its integration into five data management systems. Several applications studies have been described, demonstrating that the model genericity will facilitate the development of complex applications to manage large, dynamic and distributed data sets. Applications investigated, although some were developed in few dozen 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, an API to publish transitions and execute transition handlers, the ability to plug into legacy 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 collections 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 [31], a distributed and cooperative content delivery network to distribute virtual appliance images embedding large HEP applications to Internet Desktop Grid resources [53] and a distributed network of checkpoint image server featuring server selection using network distance [54].

Acknowledgements

Some of the experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the Inria ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies. This work is partly funded by the French National Research Agency within the MapReduce project under contract ANR-10-SEGI-001.

[1] T. Hey, S. Tansley, K. Tolle (Eds.), The Fourth Paradigm: Data-Intensive Scientific Discovery, Microsoft Research, Redmond, Washington, 2009.

[2] G. Fox, T. Hey, A. Trefethen, Where does all the data come from?, Tech. rep., Indiana University (November 2011).

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

[4] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Communications of the ACM 51 (2008) 107–113.

[5] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. 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, ACM, New York, NY, USA, 2007.

[6] D. Peng, F. 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, USENIX Association, Berkeley, CA, USA, 2010, pp. 1–15.
URL http://portal.acm.org/citation.cfm?id=1924943.1924961

[7] R. Meier, V. Cahill, Taxonomy of distributed event-based programming systems, The Computer Journal 48 (5) (2005) 602–626.

[8] T. von Eicken, D. E. Culler, S. C. Goldstein, K. E. Schauser, Active messages: a mechanism for integrated communication and computation, SIGARCH Computing Architectures News 20 (1992) 256–266. doi:http://doi.acm.org/10.1145/146628.140382.
URL http://doi.acm.org/10.1145/146628.140382

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

[10] P. T. Eugster, P. A. Felber, R. Guerraoui, A.-M. Kermarrec, The many faces of publish/subscribe, ACM Computing Surveys 35 (2003) 114–131. doi:http://doi.acm.org/10.1145/857076.857078.
URL http://doi.acm.org/10.1145/857076.857078

[11] T. Murata, Petri nets: Properties, analysis and applications., in: Proceedings of the IEEE, 1989, pp. 541–580.
URL http://www-cad.eecs.berkeley.edu/~{}polis/class/papers/PetriNets.pdf

[12] E. Deelman, D. Gannon, M. Shields, I. Taylor, Workflows and e-science: An overview of workflow system features and capabilities, Future Generation Computer Systems 25 (5) (2009) 528–540. doi:http://dx.doi.org/10.1016/j.future.2008.06.012.

[13] A. Rajasekare, R. Moore, C.-Y. Hou, C. A. Lee, R. Marciano, A. De Torcy, M. Wan, W. Schroeder, S.-Y. Chen, L. Gilbert, P. Tooby, B. Zhu, iRODS primer: Integrated rule-oriented data system, Synthesis Lectures on Information Concepts, Retrieval, and Services 2 (1).

[14] K. Shvachko, K. Hairong, S. Radia, R. Chansler, The hadoop distributed file system, in: Proceedings of the 26th Symposium on Mass Storage Systems and Technologies, MSST 2010, IEEE, 2010, pp. 1–10. doi:10.1109/MSST.2010.5496972.

[15] G. Fedak, H. He, F. Cappello, Bitdew: a programmable environment for large-scale data management and distribution, in: SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, IEEE Press, Piscataway, NJ, USA, 2008, pp. 1–12.

[16] R. Love, Kernel korner: intro to inotify, Linux Journal 2005 (139) (2005) 8–.
URL http://dl.acm.org/citation.cfm?id=1103050.1103058

[17] I. Foster, Globus online: Accelerating and democratizing science through cloud-based services, Internet Computing, IEEE 15 (3) (2011) 70–73.

[18] Apache Hadoop, http://hadoop.apache.org/.

[19] B. Tang, M. Moca, S. Chevalier, H. He, G. Fedak, Towards MapReduce for Desktop Grid Computing, in: Fifth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC'10), IEEE, Fukuoka, Japan, 2010, pp. 193–200.

[20] Amazon simple storage service, http://aws.amazon.com/s3/ (2010).

[21] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, M. Seltzer, Provenance-aware storage systems, in: Proceedings of the 2006 USENIX Annual Technical Conference, 2006, pp. 43–56.

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

[23] J. Dean, S. 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.

[24] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, G. Fox, Twister: a runtime for iterative mapreduce, in: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC '10, ACM, New York, NY, USA, 2010, pp. 810–818.

[25] O. K. 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, 2012.

[26] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, R. Sears, Mapreduce online, in: Proceedings of the 7th USENIX conference on Networked systems design and implementation, NSDI'10, Berkeley, CA, USA, 2010, pp. 21–21.

[27] I. Foster, J. Vöckler, M. Wilde, Y. Zhao, Chimera: Avirtual data system for representing, querying, and automating data derivation, Scientific and Statistical Database Management, International Conference on 0 (2002) 37. doi:http://doi.ieeecomputersociety.org/10.1109/SSDM.2002.1029704.

[28] F. Costa, L. Silva, G. Fedak, I. Kelley, Optimizing the Data Distribution Layer of BOINC with BitTorrent, in: Proceedings of the 2nd IPDPS'08 Workshop on Desktop Grids and Volunteer Computing Systems, PCGrid 2008, Miami, Florida, 2008, pp. 1–8.

[29] B. Wei, G. Fedak, F. Cappello, Towards Efficient Data Distribution on Computational Desktop Grids with BitTorrent, Future Generation Computer Systems 23 (7) (2007) 983–989.

[30] F. Costa, L. Silva, G. Fedak, I. Kelley, Optimizing Data Distribution in Desktop Grid Platforms, Parallel Processing Letters 18 (3) (2008) 391–410.

[31] G. Antoniu, J. Bigot, C. Blanchet, L. Bougé, F. Briant, F. Cappello, A. Costan, F. Desprez, G. Fedak, S. Gault, K. Keahey, B. Nicolae, C. Pérez, A. Simonet, F. Suter, B. Tang, R. Terreux, Scalable Data Management for MapReduce-Based Data-Intensive Applications: a View for Cloud and Hybrid Infrastructures, International Journal on Cloud Computing 2 (2-3).

[32] B. Tang, H. He, G. Fedak, HybridMR: A New Approach for Hybrid MapReduce Combining Desktop Grid and Cloud Infrastructures, Concurrency Practice and Experience.

[33] A. Simonet, G. Fedak, M. Ripeanu, S. Al-Kiswany, Active Data: A Data-Centric Approach to Data Life-Cycle Management, in: Proceedings of the 8th Parallel Data Storage Workshop, PDSW 2013, ACM, Denver, USA, 2013, pp. 39–44.

[34] A. Simonet, K. Chard, G. Fedak, I. Foster, Active Data to Provide Smart Data Surveillance to E-Science Users, in: Proceedings of IEEE 23rd Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2015, Turku, Finland, 2015, to appear.

[35] Y. Demchenko, P. Grosso, C. de Laat, P. Membrey, Addressing big data issues in scientific data infrastructure, in: Proceedings of the 2013 International Conference on Collaboration Technologies and Systems, CTS 2013, IEEE, 2013, pp. 48–55.

[36] T. Ho, D. 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, 2007, pp. 39–46.

[37] L. Ramakrishnan, D. Ghoshal, V. Hendrix, E. Feller, P. Mantha, C. Morin, Storage and Data Life Cycle Management in Cloud Environments with FRIEDA, in: X. Li, J. Qiu (Eds.), Cloud Computing for Data Intensive Applications, Springer, 2015.

[38] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva

de la Hidalga, M. P. Balcazar Vargas, S. Sufi, C. 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) (2013) W557–W561. arXiv:http://nar.oxfordjournals.org/content/41/W1/W557.full.pdf+html, doi:10.1093/nar/gkt328.
URL http://nar.oxfordjournals.org/content/41/W1/W557.abstract

[39] J. Goecks, A. Nekrutenko, J. Taylor, The Galaxy Team, Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences, Genome Biology 11 (8) (2010) R86.

[40] D. Rex, J. Ma, A. Toga, The LONI pipeline processing environment, NeuroImage 19 (3) (2003) 1033–1048.

[41] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, I. Foster, Swift: A language for distributed parallel scripting parallel computing, Parallel Computing.

[42] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, et al., Pegasus: A framework for mapping complex scientific workflows onto distributed systems, Scientific Programming 13 (3) (2005) 219–237.

[43] K. Taura, K. Kaneda, T. Endo, A. Yonezawa, Phoenix: a parallel programming model for accommodating dynamically joining/leaving resources, SIGPLAN Notice 38 (2003) 216–229. doi:http://doi.acm.org/10.1145/966049.781533.
URL http://doi.acm.org/10.1145/966049.781533

[44] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, A. M. Vahdat, Mace: Language support for building distributed systems, in: Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation, PLDI'07, ACM, New York, USA, 2007.

[45] D. Mazieres, A toolkit for user-level file systems., in: Proceedings of the 2001 USENIX Annual Technical Conference, Berkeley, CA, USA, 2001.

[46] S. Yoo, H. Lee, C. Killian, M. Kulkarni, Incontext: simple parallelism for distributed applications, in: Proceedings of the 20th international symposium on High performance distributed computing, HPDC '11, ACM, 2011, pp. 97–108.

[47] J. Cheney, S. Chong, N. Foster, M. Seltzer, S. Vansummeren, Provenance: a future history, in: Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications, OOPSLA '09, ACM, New York, NY, USA, 2009.

[48] Y. L. Simmhan, B. Plale, D. Gannon, A survey of data provenance in e-science, SIGMOD Rec. 34 (3). doi:10.1145/1084805.1084812.

[49] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, et al., The open provenance model core specification (v1. 1), Future Generation Computer Systems 27 (6) (2011) 743–756.

[50] K.-K. Muniswamy-Reddy, P. Macko, M. Seltzer, Provenance for the cloud, in: Proceedings of the 8th Conference on File and Storage Technologies (FAST'10), San Jose, 2010.

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

[52] E. Vairavanathan, S. Al-Kiswany, L. Costa, Z. Zhang, D. Katz, M. Wilde, M. Ripeanu, A workflow-aware storage system: An opportunity study, in: 12th International Symposium on Clusters, Cloud, and Grid Computing (CCGrid'12), Ottawa, Canada, 2012.

[53] H. He, G. Fedak, P. Kacsuk, Z. Farkas, Z. Balaton, O. Lodygensky, E. Urbah, G. Caillat, F. Araujo, A. Emmen, Extending the EGEE Grid with XtremWeb-HEP Desktop Grids, in: Proceedings of the 4th Workshop on Desktop Grids and Volunteer Computing Systems, CCGrid 2010, Melbourne, Australia, 2010, pp. 685–690.

[54] S. Ratnasamy, M. Handley, R. Karp, S. Shenker, Topologically-Aware Overlay Construction and Server Selection, in: Proceedings of the 25th Annual Joint Conference of the IEEE Computer and Communications Societies, Vol. 3 of INFOCOM 2002, IEEE, 2002, pp. 1190–1199.