2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

Towards Reliable, Performant Workflows for Streaming-Applications on Cloud Platforms

Daniel Zinn, Quinn Hart,
Timothy McPhillips, Bertram Ludäscher
University of California
Davis CA 95616
{*dzinn,qjhart,tmcphillips,ludaesch*}@*ucdavis.edu*

Yogesh Simmhan, Michail Giakkoupis,
Viktor K. Prasanna
University of Southern California
Los Angeles CA 90089
{*simmhan,giakkoup,prasanna*}@*usc.edu*

***Abstract*—Scientific workflows are commonplace in eScience applications. Yet, the lack of integrated support for data models, including streaming data, structured collections and files, is limiting the ability of workflows to support emerging applications in energy informatics that are stream oriented. This is compounded by the absence of Cloud data services that support reliable and performant streams. In this paper, we propose and present a scientific workflow framework that supports streams as first-class data, and is optimized for performant and reliable execution across desktop and Cloud platforms. The workflow framework features and its empirical evaluation on a private Eucalyptus Cloud are presented.**

I. Introduction

Scientific workflows have gained a firm foothold in modeling and orchestrating data intensive scientific applications by scientists and domain researchers [1]. Despite advances in workflow systems, the diversity of data models supported by workflows remains inadequate. Directed acyclic graphs (DAGs), and control and data flows operating on simple value types and files form the most common programming model available. Workflow systems that support collections or structured objects [2], [3] are more the exception than the rule.

While existing workflow data models are sufficient for a number of legacy applications that were originally orchestrated as scripts operating on files, an emerging class of scientific and engineering applications needs to actively operate on data as it arrives from sensors or instruments, and react to natural or physical phenomena that are detected.

In addition, these novel data and compute intensive applications are well suited to be targeted for Cloud platforms, whether public or private [4], [5]. The elastic resources available on the Cloud fit with the non-uniform resource needs of these applications, and the on-demand nature of the Cloud can help with their lower latency requirements. However, the native data services offered by many public Clouds – files, queues and tables – do not yet include high-performance, streaming-friendly services.

For example, consider the energy informatics domain and *smart power grids*[1] in particular. Data continuously arriving from 1.4 million smart meters in Los Angeles households will soon need to be continuously analyzed in order to detect impending peak power usage in the smart power grid and notify the utility to respond by either spinning up additional power sources or by triggering load curtailment operations to reduce the demand [5]. This closed loop cyber-physical application, modeled as a workflow, needs to combine streaming data arriving from sensors with historic data available in file archives along with structured collections of weather forecast data that help the large scale computational model make an energy use prediction in near real time. A workflow framework that supports this data model diversity, including streaming data, structured collections and files, and the ability to execute reliably and scalably on elastic computational platforms like the Cloud is currently absent.

In this paper, we address this lacuna by proposing a scientific workflow framework that supports the diverse data models required by these emerging scientific applications, and evaluate its performance and reliability across desktop and Cloud platforms. Specifically, we make the following contributions:

1) We motivate and present a workflow architecture that natively supports the three common data models found in science and engineering applications – *files*, *structured collections* and *data streams* – with the ability to seamlessly transition from one data model to another;
2) We incorporate and evaluate techniques in the workflow framework to ensure *high performance* of streaming applications across desktop and Cloud platforms; and
3) We describe architectural features that enhance *reliability* of the dataflows in distributed, Cloud environments for streaming applications.

The rest of this paper is organized as follows: Section II motivates the need for workflow support for diverse data models using applications from energy informatics and identifies desiderata, Section III introduces the data model and dataflow primitives used by our workflow framework, Section IV describes the workflow framework architecture, Section V highlights features that support high performance

[1] www.smartgrid.gov/projects/demonstration_program

978-0-7695-4395-6/11 $26.00 © 2011 IEEE
DOI 10.1109/CCGrid.2011.74

streaming dataflows on Cloud and hybrid platforms, Section VI discusses reliability of streaming applications during distributed execution, Section VII experimentally evaluates the performance and reliability of the framework for an energy informatics application on the Eucalyptus Cloud platform, Section VIII presents related work and we summarize our conclusions in Section IX.

II. Background and Motivation

A. *Energy Informatics Applications*

Pervasive deployment of sensors and instruments is allowing fine-grained monitoring of the environment about us. These range from orbiting satellites and rain gages on the field, to smart meters at households. While scientists have been dealing with this data deluge by storing and processing data periodically, there is a growing need to analyze these data as they arrive. In addition to the smart grid application introduced before, we motivate this need using the GOES satellite data processing application for solar radiation estimation.

NOAA's Geostationary Operational Environmental Satellite (GOES) generates continuous, real-time earth observations in multiple spectral bands that provide information for developing meteorological parameters like, cloud coverage over large regions of the earth [6]. Cloud cover maps are combined with clear sky radiation models and used to generate actual net solar radiation (R_{ns}) intensity maps [7]. Such maps can be used to estimate power generation from solar panels over the course of a day, for example, and to plan an appropriate power usage schedule.

These satellite derived maps can be combined with other streams of sensor information to develop more sophisticated parameters. For example, the California Irrigation Management Information System (CIMIS) program[2] combines R_{ns} with spatially interpolated estimates of temperature, wind speed, and relative humidity to create reference evapotranspiration (ET_0) maps [6]. They post these estimates online to help California farmers and water managers plan their daily water needs [8].

Algorithms for calculating ET_0 and R_{ns} maps are computationally costly and data intensive. Though data from the satellite arrives continuously at the UC-Davis campus, lack of programming and data models that support both stream and static file processing cause these data to be processed in batches, using files to buffer time windows of stream data. This, combined with the limited compute resources available locally at CIMIS, means that these maps are currently only generated once a day. More frequent map updates – every hour – and at a finer spatial resolution will be beneficial to both water and solar power managers.

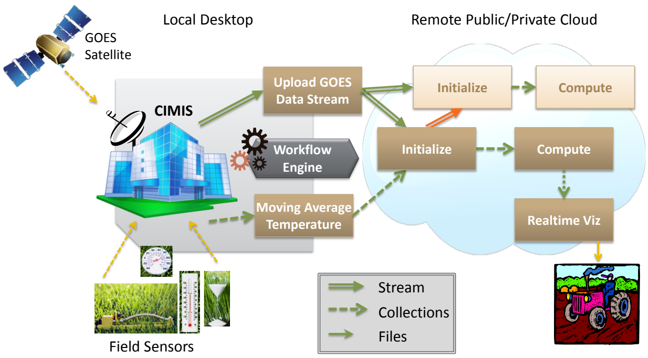


Figure 1. Interactions in the GOES Solar Radiation Workflow

Figure 1 shows an ideal workflow that uses streams, files and collections to generate R_{ns} and ET_0 maps. Local computers retrieve and reformat the specialized satellite and sensor stream data format. Standardized streams are then made available to Cloud computing systems for image processing, parameter creation and aggregation of disparate datasets. The results are high level environmental indicators.

B. *Cloud and Hybrid Platforms*

Clouds are gaining acceptance as a viable platform for data intensive computing in the sciences and engineering. Besides their well-known benefits, Cloud computing offers particular advantages for data driven eScience applications, such as elastic scale-out of computation and on-demand provisioning of resources with a pay-as-you-go model. For many novel, loosely-coupled applications that are being developed, Clouds provide a compelling alternative to clusters and Grids, with the option of public (Amazon AWS[3], Microsoft Azure[4], Google AppEngine[5]) or private (Eucalyptus[6], Nimbus[7], OpenNebula[8]) hosting. Even national labs are beginning to evaluate the advantages of Cloud platforms[9].

Public Clouds often provide reliable and scalable data structures such as message queues (Amazon Simple Queue Service, Microsoft Azure Queue Service), file and collection storage (Amazon S3, Microsoft Azure Blob Service), and tables (Amazon SimpleDB, Microsoft Azure Table Service, Google BigTable [9]). These can be used as building blocks for higher order applications. Some of the Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) Cloud providers also allow direct TCP socket access to Cloud Virtual Machines (VMs) from the Internet, with restrictions (E.g. limited number of public IPs in Amazon, passing through a load balancer for Azure). While these features

[2] http://www.cimis.water.ca.gov/cimis/data.jsp

[3] aws.amazon.com

[4] www.microsoft.com/windowsazure

[5] code.google.com/appengine

[6] www.eucalyptus.com

[7] www.nimbusproject.org

[8] www.opennebula.org

[9] newscenter.lbl.gov/press-releases/2009/10/14/scientific-cloud-computing/

can help with higher-performance for streaming applications [10], there is no public or private Cloud provider that supports native data structures that meets the needs of streaming applications.

C. Requirements Summary

The energy informatics applications we have identified are well suited to run on public or private Clouds. Since many of the data sources are from scientific instruments or sensors, the applications will have to span desktop clients or workstations, which receive the instrument data, and Clouds, where the majority of computation takes place. Scientific workflow frameworks provide an ideal starting point to compose and execute such applications in the Cloud given their maturity for eScience domains.

These applications also, however, highlight the need for streaming data models within workflows that can work effectively across desktop and Cloud platforms. Users need stream programming abstractions in workflow tasks, just as file access is taken for granted. These logical stream abstractions have to be more robust than simple TCP sockets, given the unreliability and opaqueness introduced by operating in a distributed environment across desktop and Cloud with different characteristics from a typical local area network. Reliability of VMs hosting workflow tasks is another concern to be addressed. Too, there has to be intelligence to avoid costly (both in time and money) duplicate movement of the same logical stream. Some of these existing shortcomings have been exposed in our recent work [10].

Abstractions in the workflow must also hide the need to explicitly construct trivial transformations across data models, such as from streams to files, or files to collections. Many of these steps should be automated with limited domain knowledge. This need to move between models is common in legacy code that do not support streams natively and operate on time windows of streams as files.

III. Workflow Data Model

Our proposed workflow framework supports the three data models that were motivated, viz., files, collections and streams. Tasks in a workflow support these models as first class input or output ports or parameters. We describe these models in this section along with specialized dataflow primitives to operate on them in a transparent manner.

A. Data Model Characteristics

Files: Files are a popular data model for scientific applications and commonly supported in workflows. They are *bounded-sized* data that reside on *local disk* or a shared file system, and can be accessed through standard file system primitives by workflow tasks. A file's content may change over time, though scientific data tends to be *static* as workflows create updated copies of files rather than change them in place, for provenance tracking. File storage mediums are also typically *persistent*. Files may expose either a well-defined structure (e.g. HDF or XML) or use an opaque binary format with *random access*.

Structured Collections: Collections contain an *ordered set* of items with well-defined *structure*. They are typically *bounded* in size. Exposing their structure allows interesting queries and *access patterns*. The itemized nature of collections makes them well suited for tasks to *iterate* over them. Collections can be *nested* and items can potentially refer to the other two data models – files and streams. Collections may also have data to *object mapping* in higher level languages for access by workflow tasks. Workflow systems such as Kepler/COMAD [11] and Taverna [3] provide support for collections.

Data Streams: Streams are a continuous series of *binary data*. They are often *unbounded* in size – a key distinction – and accessed as a logical *byte stream*. The continuous nature of streams also makes them *transient* unless mapped to another data model. Streams often need to be handled at *high rates* of flow, but these rates can vary. Streams may have *landmarks* [12] within them, that act as a point of reference and serve to delineate them. Landmarks for a stream from an instrument may be the instrument starting and stopping.

One common and implicit data model that is supported by workflows are value parameter types such as strings, numbers and booleans. These are well understood and commonly supported. Their discussion is omitted for brevity.

B. Workflow Primitives for Streaming

1) Always-On Workflows: Traditionally, when a workflow is executed, tasks in the workflow are orchestrated to execute in a certain order. Tasks typically execute once, as for example in the DAGman[10] workflow model, or several times in case control flows like iterations are allowed, and the workflow eventually stops. Introducing a streaming and collection notion in a workflow also allows tasks in the workflow to be invoked multiple times. While collection oriented workflows have used it earlier to introduce control flow into a pure data flow workflow by iterating over a collection [3], the use of unbounded streams brings the possibility of *always-on workflows* that are alive and executing constantly. This provides a more natural execution model [13] for workflows that are constantly responding to environmental conditions based on stream outputs from sensors.

2) Transforming between Models: The need to transform from one data model to another is common as part of workflow composition and execution. This helps to support legacy applications, to match the output type from a previous workflow activity to a required input type by a subsequent activity, and sometimes even to rewrite workflow patterns (e.g., from sequential to pipeline parallel) and better leverage

[10] www.cs.wisc.edu/condor/dagman

available computational resources. A legacy application may, for example, require temperature values to be present in a file to use it as input type while a previous activity generates a structured collection. This requires a form of "materialization" of the data.

Traditionally, activities called "shims" [14] have been used to explicitly map between data structures and types in a workflow. However, when using different data models, it is possible for the workflow framework to automate translation from one model to another while conforming to certain rules.

Mapping from files to collections and back is possible when the files have structure – possibly domain dependent. For example, a NetCDF file containing a 2-dimensional array of temperature values can be mapped to a collection of 1-dimensional arrays of temperatures, or a doubly-nested collection of temperatures. For workflows used in environmental sciences, providing such a NetCDF mapping function can help implicitly translate between the models, and just the row- or column-major transformation need be specified. A similar argument can also be made for XML based files that naturally fit a collection model [15].

Transforming from streams to files and back is more easily managed since both operate intrinsically on bytes, but it may provide limited benefits when done naively. It is possible to trivially write bytes from a stream to a file, and chunk by the number of bytes to control size of a single file. But this works in practice for only the simplest of cases where a stateless task operates on each byte in the stream or file independently. A more useable notion of capturing streams to files comes from landmarks defined in the streams [12]. Landmarks form the logical boundaries between portions of the stream and can be specific events, such as an instrument going on or off, logical or real timestamps, or byte-boundaries for a series of fixed-sized data structures. Capturing data between two landmarks, in a single file or a collection of files with offsets from the head, will prove more useful for mapping data models between two tasks.

Going from collections to streams also becomes tractable with the use of landmarks. While data between landmarks can be treated as one item in a collection, the continuous nature of streams means that the size of items or the number of items can grow large. Collections need to be bounded by the number of items. Mapping collections to streams also requires thought on the serialization to be performed. This may again be domain specific, but the workflow framework can provide the hooks for automation.

3) Pipeline Parallelism: A streaming model innately allows pipeline parallelism among workflow tasks. Once a task completes processing a particular region of the stream, it can generate an output in its stream output port and continue processing the next region in the stream. Subsequent tasks can start and continue operating on the output stream values. Combined with a Cloud platform, this allows pipelined tasks to be run on different VMs and scaleout on available resources. The logical streaming model – as opposed to a physical TCP socket – also allows elasticity of tasks, by permitting stateless tasks to migrate to other VMs and scaleout computation, or to gather in a single VM and conserve resources. The benefits of pipeline parallelism, however, only extend to those tasks that are linked together by streams.

4) Data Parallelism: Data parallelism is often exploited by scientific workflows [16] and is a predominant way to achieve efficient execution on distributed resources. Trivial data parallelism using streams is inhibited since streams arrive over a period of time and the workflow framework needs to provide the necessary logic to distribute streams to stateless tasks that can operate on the streams in parallel.

Using the concept of landmarks, there are two ways to achieve data parallelism for stateless tasks operating on streams. One, the streams can be mapped to bounded collections (or files) using transformations discussed before, bounded by number of items or a time window per collection, and the items in the collection be executed data-parallel by instances of the same task. This is an explicit materialization of the stream and the time to buffer the stream is overhead unless it is pipelined.

Alternatively, the stream can be duplicated and passed to multiple instances of a task, with each task responsible for processing beyond the i^{th} landmark. The tasks would either track and skip regions between landmarks that are not of interest, which has limited overhead if the tasks are collocated in the same Cloud VM, or the streaming framework can perform an implicit filter by landmark for optimization.

However, certain tasks need to maintain state between invocations. These tasks range from computing simple averages, maxima, or minima, up to performing complex stream analysis such as determining frequent itemsets [17]. Here, the data units (e.g., single data items, collections or files) need to be processed *sequentially*, precluding the use of data-parallel approaches altogether.

IV. Architecture

Figure 2 summarizes our proposed Workflow architecture. The workflow engine orchestrates the overall execution and is often located on a computer outside the Cloud (laptop, desktop, or server for long-running workflows). We chose to extend the RestFlow [18] workflow engine since it provides, besides the regular DAG workflow execution model, the ability to invoke workflow tasks multiple times and to manage collections passed between task invocations. The workflow tasks inside the RestFlow system orchestrate workers in the Cloud. Work requests and responses are communicated through a queue, provided by the Cloud infrastructures. This allows for automatic load balancing and fault tolerance since work requests are only "leased" by a task and destroyed

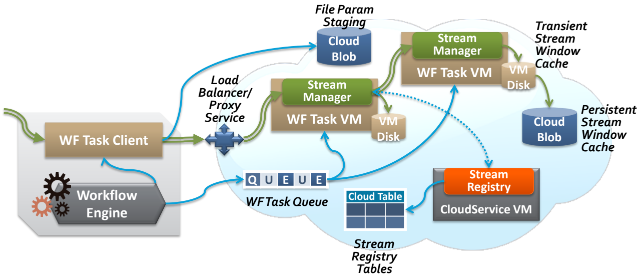


Figure 2. Architecture of Workflow Framework across Cloud Platforms. Green arrows represent high-performance streaming data flow; blue arrows represent smaller control data flow.

only after the work has been performed successfully. Besides existing support for accessing and operating on files and collections provided by RestFlow, we provide the additional modeling abstraction of named, fault-tolerant, shareable, and persistent streams. To facilitate inter-workflow operability and re-use, we implement a stream-management system outside the workflow engine to provide this abstraction.

The stream-management system runs predominantly inside the Cloud, currently as user-processes. In the future, Cloud providers may provide such streams as IaaS abstraction. Its main component is a registry service that maintains a list of known streams and the endpoints where particular streams are provided. The registry maintains this state in a persistent Cloud table. Stream managers run as separate processes on the Cloud VMs, and can function as stream providers. When a task creates an output stream, it contacts a nearby/local stream server that registers the stream. The data is requested by the stream manager if the stream does not exist already or if the *overwrite* flag is set. If the *persist* flag is set, the stream manager will also cache the data stream to BLOB storage as it is received. For uploading to BLOB cache, we chunk streams based on maximum chunk-size and time-out configurations, whichever occurs first. When a stream is accessed for reading inside the Cloud, a registry-lookup is performed to obtain a nearby stream-manager from which the stream can be read. When accessed from outside the Cloud, the contacted manager will transparently forward the stream to others who need it. This allows the registry to use Cloud-internal addresses for the managers. It further requires no modification to existing Cloud infrastructures, which provide load balancing mechanisms for TCP connections.

V. Stream Performance Features on Cloud

Unlike the performance of common Cloud data services like files, blobs, tables and queues, the performance of streams within Clouds is less studied. In our earlier work [10], we demonstrated the superior data transfer bandwidth using streams as a transport mechanism for moving files into the Azure public Cloud, relative to BLOB file transfers. Besides those transfer optimizations, several novel features have been incorporated on top of our workflow framework to make it performant for a streaming data model spanning desktop and Cloud platforms.

The use of named streams and landmarks allow streams to be shared with multiple destinations. This is of prime importance when a stream source at a task running in the desktop is shared by several workflow tasks in the Cloud. Duplicating this stream transfer will use up bandwidth and be punitive in terms of cost. It may also affect the latency of task execution since some Cloud vendors throttle the cumulative bandwidth for a single user account into their public Cloud. The peering stream managers we support in combination with the stream registry addresses this by sharing the stream within the Cloud while passing just one stream from desktop to Cloud. The empirical advantages of this are illustrated in Section VII-B1.

The ability of the stream manager to cache the streams locally on VM disk ensures that the performance benefits of shared streams will outlast the memory available in the VM. Additionally, the use of Cloud persistent storage to cache some of the streams will help them be reused within the Cloud beyond the lifetime of the VM, and also offload bandwidth or computation overhead on a VM caused by its sharing a stream.

Currently, our stream managers do not coordinate access to replicas of streams and it is likely under certain cases for a particular VM hosting a stream to be overloaded by requests. We are working on more intelligent and fair stream sharing.

VI. Reliability for Streaming Applications

The always-on nature of our applications and their use by a large user community means that the workflows should exhibit tolerance to faults. Our earlier work has identified fault recovery models for file based workflows [19]. Here, we restrict our attention to the reliability of workflows that use a streaming model.

There are two aspects of fault resistance: (1) transient or permanent loss of physical network, and (2) loss of virtual machines in the Cloud or services running on them. Transmitting streams across desktop and the Cloud over TCP sockets can be prone to error, particularly given the long lifetime of the logical streams. A network reconfiguration on the desktop workstation, migration of a laptop to a different wireless network or the restart of a desktop server stream source after installing patches can all cause unintentional loss of network connection between desktop and Cloud for various periods of time. The use of a logical stream model, exposed as a Java class implementing an interface similar to a byte stream, hides the underlying network transport and loss from the workflow application. A disconnect of the TCP socket due to transient network error can be recovered by reconnecting to the same source. A permanent network failure can be sidestepped by locating and connecting to a replicated stream source if available, or an optimistic attempt to reconnect with the original source. The protocols used by the stream managers to communicate with each other seamlessly recovers from the point at which the physical stream was broken, and translates to just a slight increase in latency for the receiver of the stream rather than a permanent failure. This is shown in our experiments in Section VII-C.

The loss of network connectivity between VMs in the Cloud is less frequent, but can be handled in the same manner as above. More of a concern is the loss of a VM instance due to, say, the loss of the physical host or a rolling upgrade to the Cloud fabric [20], [21]. One casuality in such a case could be the stream cached in the memory or local disk of the VM that was lost. We address this by trickling the stream from the VM memory/local disk to Cloud persistent store in a background thread. This ensures persistence of the stream window even if the VM is lost and limits the extent to which the stream has to be retransmitted from desktop client to the recovered VM instance or other VM instances accessing that stream.

VII. Evaluation

We investigate the feasibility of our streaming workflow framework, and study the performance and reliability features outlined in Section VI and V. Our experiments use synthetic workloads that are similar in data size and computational needs to the GOES solar radiation and evapotranspiration workflow, which we have examined in earlier work [10].

A. Experimental Setup

We used a private Eucalyptus Cloud [22], running at University of Southern California. The Debian Linux VMs have a 2GHz CPU core and 2GB of RAM each, and the underlying host machines are interconnected with Gigabit Ethernet. Throughout our experiments, there was no other load on these machines. Each VM instance used in our tests ran on a different host machine to ensure uniform network speeds between VM machines. We chose such a controlled private Cloud environment to "micro-benchmark" our proposed features. In public Clouds, where network and host load is much more inconsistent, we expect to see the same general trends, however, overlaid with "unpredictable" noise. Our workflows are orchestrated and obtain input data from a Debian machine `lore` located at University of California at Davis, that acts as a user "desktop". The network bandwidth between `lore` and the Eucalyptus Cloud head node is 10MBit/s. Neither CPU nor data intensive computation is performed on `lore`; its particular specification is thus irrelevant. We performed each experiment at least three times and show performance average as well as the minimum and maximum measurements as error bars.

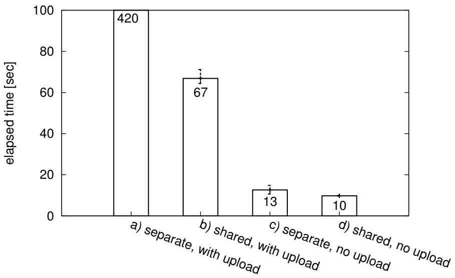


Figure 3. Data Sharing Between Cloud Workflow Tasks

B. Streaming Application Performance

Here, we investigate the workflow performance for streaming applications, focused on data stream sharing and pipeline parallelism.

1) Shared Streams: A major bottleneck in Cloud architectures is the data movement from local resources to remote Clouds and back. We evaluate the effect of stream sharing using a workload that consists of 7 identical workflows with 14 tasks each. Each workflow deploys 7 independent streams by having 7 sender and 7 receiver tasks. The sender tasks 1-7, which run on the desktop machine, create a stream and each write 10MB of data into it. The receiving tasks 8-14, open these streams and consume the data, i.e., task 1 streams to task 8, etc. To evaluate stream sharing, we consider four scenarios: *a)* all 7 × 7 streams are distinct, and the data is available only on the desktop machine. *b)* the 7 workflows operate on the same input data (i.e., there are only 7 distinct streams), which is available on the local desktop machine. In *c)* and *d)*, we similarly consider 49 and 7 distinct input streams, however here, the input streams are available in the Cloud already. In all cases, two logical data movements occur: from desktop to a stream manager in the Cloud, and from the stream managers to Cloud VM. Furthermore, since the mechanisms for selecting a stream manager (i.e., TCP load-balancer) and for selecting a Cloud VM for a task (i.e., Cloud message queue) are independent of each other, a transfer from stream manager to the worker is likely to occur. For each of the 4 cases, we launch all 7 workflows in parallel.

Figure 3 shows the total wall-clock execution times for our workloads, i.e., we measure the time from starting the workflows until the last workflow has finished. Execution time in *a)* is 420 seconds reflecting the uplink bottleneck (7×7×10MB×8Bit/420s=9.3MBit/s). In *b)* we achieve a speedup of 7x since the shared input streams are detected by the stream subsystem avoiding redundant data movement to the Cloud. This data sharing is achieved transparently, by the stream subsystem utilizing the stream registry and the fact that streams are registered with an identifying name. We also consider cases *c)* and *d)*, in which the input data is already available in the Cloud. Here, only the data movement from stream managers to worker is performed. In *c)* and *d)*, we achieve a data movement bandwidth of around 300MBit/s

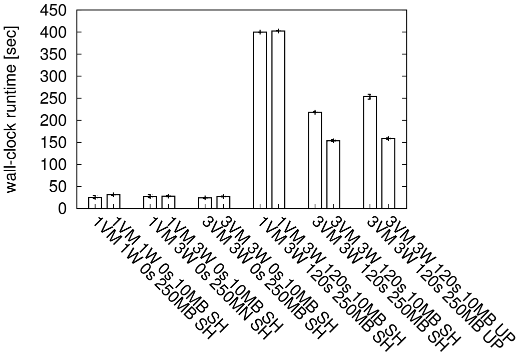


Figure 4. Investigating Pipeline Parallelism

and 400Mbit/s, respectively. While the same amount of data is moved in *c)* and *d)*, we attribute the better performance of *d)* to the fact that less data has been loaded from disk by the stream managers.

2) Pipelined Parallelism: In the next experiment, we investigate pipeline parallelism with relatively large data movements and computational load. Our workflow

$$\boxed{\text{S}} \xrightarrow{s_0} \boxed{\text{A}} \xrightarrow{s_1} \boxed{\text{B}} \xrightarrow{s_2} \boxed{\text{C}} \xrightarrow{s_3}$$

consists of four tasks in a pipeline. The source task S, running on the desktop machine, produces a stream of 100MB size. Tasks A, B, and C (running on the Cloud) each transform their input stream and produce an output stream. While s_0 has a size of 100MB, s_1, s_2, and s_3 are sized 1GB each. This is a common pattern: the first Cloud task increases the data size by, for example, decompressing input. All three transformation tasks are streaming, that is they can produce a (prefix of the) output data stream after having seen only a prefix of the incoming data stream. The "streaming degree" of a task is characterized by the granularity in which it can produce output from seen input. In our experiment, we consider the case of (i) very stream-friendly tasks that operates on the input stream in chunk-sizes of 10MB (1MB for task A) resulting in 100 chunks; and the case (ii) in which 25% increments of the stream have to be consumed before the respective 25% of output stream is produced. Chunk-sizes here are 250MB (25MB for task A). Note, that the task implementation itself decides when output data can be produced. In general, tasks will have dynamically varying "chunk-sizes" during a workflow run. Besides the chunk-size, we also vary the computational work that a task has to perform per chunk from a no-op (no workload) to a busy wait of 1.2s for the smaller 10MB chunks and 30s for the larger 250MB chunks; note that both workloads add up to 120s for the whole stream, which is performed by each of the three tasks.

Workflow end-to-end wall-clock execution times are shown in Figure 4; on the X-axis, we vary the number of used VMs (1VM or 3VM), the number of concurrently running workers on the VMs (1W or 3W), the computational load (0s or 120s), the stream chunk size used (10MB or 250MB), and whether the input stream of 100MB is already available in the Cloud (SH), or has to be uploaded by the desktop (UP). Figure 4 shows a subset of the possible combinations that show interesting results. In cases without CPU load and without desktop-Cloud upload (0s and SH), the workflow execution is fast; using 1VM or 3VMs has comparable performance, that is streaming from one VM to the next is comparable to data streaming within the same VM. Note, that since an output stream in the Cloud is streamed to the stream manager on the same host, the stream managers are co-located with the data producer. When the workflow is executed completely in series (1VM 1W), using 100 chunks is 24% slower than using only 4 chunks. We attribute this slow-down to the increased amount of work to manage the smaller blocks individually. The observation that the execution time for (3VM 3W 10MB) is 12% faster than for (1VM 3W 10MB) reinforces this hypothesis.

More interesting results are obtained when not only data is moved, but also CPU load is performed. We note that a serial execution of the workload has a lower bound on the execution time of 3×120s=360s. Letting 3 workers run concurrently on 1VM does not improve performance, which is expected since the tasks perform CPU intensive busy waiting. Here, the penalty of having a smaller chunk-size is reduced to about 0.5%. Once three VMs are used, pipeline parallelism is exploited. In the case of 4 chunks, the workflow has a speedup of 1.8x compared to the execution on one host. Furthermore, when the chunk-size is reduced to 10MB, we obtain a speedup of 2.6x – only 13% short of a perfect speedup of 3x.

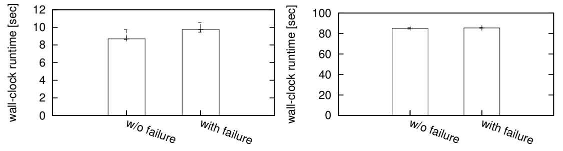
Pipeline parallelism mitigates the impact of adding additional stages. The overall execution time varies significantly only if the added stages have a much lower throughput. This is demonstrated by the case in which the input data is not available in the Cloud, but has to be streamed into the Cloud first. Although uploading the data takes around 80s (see earlier examples; 10MBit/s), the execution time increases only by 35s for the 250MB chunks and 4.8s for chunks sized 10MB.

C. Testing Reliability

We investigate the fault-resistance of our stream abstraction with a simple workflow that contains one task reading a stream. We consider two cases: a) the task is run in the Cloud, and b) the task is run on the desktop (i.e., the data is downloaded from the Cloud). We use a stream size of 1GB and 100MB respectively. After 10% of the stream has been received, we simulate a failure of the stream manager by killing the stream manager process. The kill command is triggered from inside the task via an asynchronous ssh-connection to the stream manager's VM[11]. In both cases,

[11] We used a fixed port-forwarding from Cloud head node to the VM in the Cloud–desktop use-case to reach the Cloud VM from the desktop.

the stream is replicated at two stream managers such that the second manager can also serve the stream. The second stream manager does not monitor the first one, and the failover is completely performed by the stream implementation inside the task.



(a) Cloud — Cloud (b) Cloud – Desktop

Figure 5. Runtimes while tolerating failures

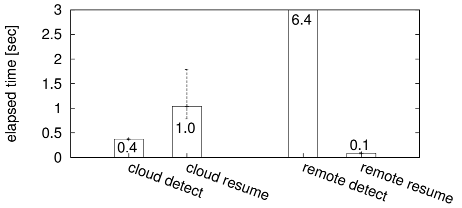


Figure 6. Timings for failure detection and resume

Our experiments demonstrate that the failure is hidden by our stream abstraction, i.e., the task can successfully download and operate on the stream *as if* there was no error. Figure 5 shows that while the failure introduces a slowdown of around 1 second inside the Cloud (slowing transfer speed from 919Mbit/s to 840MBit/s), the difference for Cloud–desktop is only 350ms on average causing a slowdown of less than 0.4%. In Figure 6, we show the time to detect the failure (measured as the time span from initiating the remote kill command to receiving an exception while reading from the TCP socket), and the time to resume reading (the time span from exception to when new data has arrived). As expected, the detection is much faster inside the Cloud than it is from Cloud to desktop. Interestingly, however, the time to detect the failure is not wasted: Although the server process is killed within a fraction of a second (manual observation), the desktop client is still receiving data for another 5 seconds. This behavior is due to the implicit buffering of data packets as sent through the (inter-) network.

Furthermore, note that the desktop-Cloud resume is much faster than the Cloud-internal resume. This is because inside the Cloud, we first try to contact the same stream manager again, and then contact the registry to obtain a list of stream managers providing the stream, from which we randomly select one. From outside the Cloud, we simply connect to the load-balancer on the Cloud-framework, without doing a round-trip to the registry. Since we had only two managers running, the round-robin connect was successful immediately. In case there are more stream managers running, and the load balancer selects one that does not provide the stream, then the stream manager itself will contact the registry, and pull the stream from an appropriate other manager to forward it to the client.

This fail-stop of the stream-manager was easy to detect by our system since the still running VM will reset the TCP connection. In case of permanent network errors or crashes of the whole VM, appropriate mechanisms for detecting failure need to be deployed. Since network outages and complete VM failures are indistinguishable by the client, a balance between tolerance against shorter outages and declaring a VM as lost need to be found. One observation was that TCP does not reset the connection even after 2 minutes of network outage (tested via iptables DROP and DENY rules).

VIII. Related Work

There are several frameworks and platforms, spanning workflows, stream processing and Cloud programming models, which support a subset of the features that we require and have presented in this article.

A. *Scientific Workflows*

Scientific workflows allow composition of applications using a control and/or data flow model [23]. Scientific Workflows have been well studied [3], [16], [24]–[26], and, more recently, adapted to run on Cloud platforms [26], [27]. The data models supported by workflows have grown to include value parameters, files and collections [2]. However, one key data model that has been absent from workflows is streams. While [23] mentions collections as being a type of stream, we make the distinction that streams are unbounded, often have opaque structure, and require high performance to keep up with the generating instrument or sensor. Some workflow systems [28] have also used the streaming transport in GridFTP [29] for data transfers in the Grid. We distinguish this use of socket streaming for transport of files (also used in our recent work [10]) from the logical data streams we introduce in this paper and support at the workflow data model level.

The closest comparable work to ours is by the StreamFlow model for workflows [30]. StreamFlow incorporates complex event processing (CEP) into WS-BPEL workflows by introducing a StreamFlow edge into the data flow model, and specialized tasks that perform event processing. While similar in some respects, we make distinct contributions. The CEP model used by StreamFlow data edges is more similar to unbounded collections than our treatment of logical streams [31]. Consequently, it makes structural assumptions of streams comparable to collections, such as a time series of events and the ability to perform filters on event streams. Our streams are intentionally more basic since our structured collection data model provides many features of StreamFlow, except unboundedness. Also, our logical streams demonstrate reliability features and performance optimization for Clouds absent in StreamFlow, which uses the Esper CEP engine for event processing. In addition, we reduce complexity by providing a single workflow execution model that combines file, collection and stream processing and executes tasks multiple times as necessary in an always-on fashion, rather than separate the workflow into a pure CEP workflow and a pure static workflow.

B. Stream and Complex Event Processing

Both stream processing systems and complex event processing engines are well established areas. Stream processing allows continuous queries to execute on a moving window of data, and has its roots in data processing in sensor network. Salient streaming systems include TelegraphCQ [12], and Aurora [32]/Borealis [33], and a Continuous Query Language (CQL) [34] inspired by SQL have been proposed. Stream processing has also been studied in Grid computing as part of the OGSA-DAI project [35]. Complex event processing (CEP) attempts to detect event patterns that occur across disparate event streams. CEP has been used in the financial industry to predict stock market behavior, and several vendors [36]–[39] provide technology solutions. While our logical stream data model is similar to the streams used in stream processing systems, our unique contribution comes from combining streams into a scientific workflow environment and allowing it to coexist with the other data models: files and collections.

C. Map-Reduce Platforms

The Map-Reduce programming model and its Hadoop implementation have been popular for composing applications in the Cloud. Several scientific applications are also starting to use it. A recent work, Map-Reduce Online [40] extends the batch oriented Map-Reduce model to include a streaming model to allow pipelining between Map and Reduce tasks. However, Map-Reduce by itself is not expressive enough compared to scientific workflows. In fact, some workflow systems have even included an optimized Map-Reduce pattern as a task available for workflow composers [41].

IX. Conclusions

In this paper, we have shown the need for streaming support in scientific workflows to support the next generation of scientific and engineering applications that respond to events in the environment at real time. We propose a data model for streams that can coexist with collections and files that are currently supported by workflows. Our implementation of this abstraction for the RestFlow workflow system shows it to be performant and reliable for operating across desktop and the Cloud. We plan to further build on this initial framework to implement the energy informatics applications we motivated and address novel data optimization challenges that emerge.

Acknowledgment. This work is supported by the Department of Energy sponsored Los Angeles Smart Grid Demonstration Project. The authors would like to thank the Los Angeles Department of Water and Power (LDWP) for discussions on the Smart Grid domain challenges. This work was also supported by NSF awards OCI-0722079 (Kepler/CORE), and AGS-0619139 (COMET).

References

[1] E. Deelman, D. Gannon, M. Shields, and I. Taylor, “Workflows and e-Science: An overview of workflow system features and capabilities,” *Future Generation Computer Systems*, vol. 25, no. 5, pp. 528–540, 2009.

[2] T. M. McPhillips and S. Bowers, “An approach for pipelining nested collections in scientific workflows,” *SIGMOD Record*, vol. 34, no. 3, pp. 12–17, 2005.

[3] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe, “Taverna: Lessons in creating a workflow environment for the life sciences,” *Concurrency and Computation: Practice & Experience*, pp. 1067–1100.

[4] L. Ramakrishnan, Y. Simmhan, and B. Plale, “Realization of dynamically adaptive weather analysis and forecasting in lead: Four years down the road,” in *Computational Science – ICCS 2007*, ser. Lecture Notes in Computer Science, 2007, vol. 4487, pp. 1122–1129.

[5] Y. Simmhan, S. Aman, B. Cao, M. Giakkoupis, A. Kumbhare, Q. Zhou, D. Paul, C. Fern, A. Sharma, and V. Prasanna, “An informatics approach to demand response optimization in smart grids,” Computer Science Department, University of Southern California, Tech. Rep., 2010.

[6] Q. Hart, M. Brugnach, and S. Ustin, “Calculation of daily reference evapotranspiration for California using GOES satellite measurements and CIMIS weather station interpolation,” California Department of Water Resources, Tech. Rep., 2005.

[7] C. Rigollier, O. Bauer, and L. Wald, “On the clear sky model of ESRA(Euorpean solar radiation atlas)with respect to the Heliosat method,” *Solar Energy*, vol. 68, no. 1, pp. 33–48, 2000.

[8] B. Temesgen, “CIMIS - past, present, and future,” *Water Conservation News*, October 2003, California Department of Water Resources. [Online]. Available: http://www.owue.water.ca.gov/news/news.cfm

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, “Bigtable: A distributed storage system for structured data,” in *Symposium on Operating System Design and Implementation (OSDI)*, 2006.

[10] D. Zinn, Q. Hart, B. Ludäscher, and Y. Simmhan, “Streaming satellite data to cloud workflows for on-demand computing of environmental data products,” in *5th Workshop on Workflows in Support of Large-Scale Science (WORKS)*, 2010.

[11] L. Dou, D. Zinn, T. McPhillips, S. Köhler, S. Riddle, S. Bowers, and B. Ludäscher., “Scientific Workflow Design 2.0: Demonstrating Streaming Data Collections in Kepler,” in *Data Engineering (ICDE), 2011 IEEE 26th International Conference on*. IEEE, 2011, p. to appear.

[12] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah, “Telegraphcq: Continuous dataflow processing for an uncertain world.” in *CIDR*, 2003.

[13] *Workflows for e-Science: Scientific Workflows for Grids*. Springer, 2007, ch. Adapting BPEL to Scientific Workflows.

[14] D. Hull, R. Stevens, P. Lord, C. Wroe, and C. Goble, “Treating shimantic web syndrome with ontologies,” in *AKT Workshop on Semantic Web Services*, 2004.

[15] D. Zinn, S. Bowers, and B. Ludäscher, “XML-based computation for scientific workflows,” in *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*. IEEE, 2010, pp. 812–815.

[16] J. Frey, “Condor DAGMan: Handling inter-job dependencies,” 2002.

[17] G. Grahne and J. Zhu, “Fast algorithms for frequent itemset mining using fp-trees,” *IEEE Transactions on Knowledge and Data Engineering*, pp. 1347–1362, 2005.

[18] T. McPhillips and S. McPhillips, “Restflow system and tutorial,” https://sites.google.com/site/restflowdocs/, 09 2010.

[19] Y. Simmhan, C. van Ingen, A. Szalay, R. Barga, and J. Heasley, “Building reliable data pipelines for managing community data using scientific workflows,” *e-Science and Grid Computing, International Conference on*, vol. 0, pp. 321–328, 2009.

[20] W. Lu, J. Jackson, J. Ekanayake, R. Barga, and N. Araujo, “Performing large science experiments within a cloud architecture: Pitfalls and solutions,” in *IEEE International Conference on Cloud Computing (CloudCom)*, 2010.

[21] A. Ruiz-Alvarez, Z. Hill, M. Mao, J. Li, and M. Humphrey, “Early observations on the performance of windows azure,” in *Workshop on Scientific Cloud Computing*, 2010.

[22] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, “The eucalyptus open-source cloud-computing system,” in *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE Computer Society, 2009, pp. 124–131.

[23] E. Deelman, D. Gannon, M. S. Shields, and I. Taylor, “Workflows and e-science: An overview of workflow system features and capabilities,” *Future Generation Comp. Syst.*, vol. 25, no. 5, pp. 528–540, 2009.

[24] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. C. Laity, J. C. Jacob, and D. S. Katz, “Pegasus: A framework for mapping complex scientific workflows onto distributed systems,” *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[25] B. Lud
”ascher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. Lee, J. Tao, and Y. Zhao, “Scientific workflow management and the Kepler system,” *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039–1065, 2006.

[26] Y. Simmhan, R. Barga, C. v. Ingen, E. Lazowska, and A. Szalay, “Building the trident scientific workflow workbench for data management in the cloud,” in *Proceedings of the 2009 Third International Conference on Advanced Engineering Computing and Applications in Sciences*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 41–50.

[27] G. Juve and E. Deelman, “Scientific workflows and clouds,” *Crossroads*, vol. 16, pp. 14–18, March 2010.

[28] J. Frey, T. Tannenbaum, M. Livny, I. T. Foster, and S. Tuecke, “Condor-g: A computation management agent for multi-institutional grids,” *Cluster Computing*, vol. 5, no. 3, pp. 237–246, 2002.

[29] T. G. Alliance, “Gridftp,” dev.globus.org/wiki/GridFTP, 2010.

[30] C. Herath and B. Plale, “Streamflow – programming model for data streaming in scientific workflows,” in *International Symposium on Cluster, Cloud and Grid Computing*, 2010.

[31] T. Bass, “Mythbusters: event stream processing versus complex event processing,” in *DEBS*, 2007, p. 1.

[32] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, “Aurora: a new model and architecture for data stream management,” *The VLDB Journal*, vol. 12, pp. 120–139, 2003.

[33] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker, “Fault-tolerance in the borealis distributed stream processing system,” vol. 33. New York, NY, USA: ACM, March 2008, pp. 3:1–3:44.

[34] A. Arasu, S. Babu, and J. Widom, “The CQL continuous query language: semantic foundations and query execution,” *The VLDB Journal*, vol. 15, pp. 121–142, June 2006.

[35] C. S. Liew, M. P. Atkinson, J. I. van Hemert, and L. Han, “Towards optimising distributed data streaming graphs using parallel streams,” in *Workshop on Data Intensive Distributed Computing (DIDC)*, 2010, pp. 725–736.

[36] E. Inc., “Esper - complex event processing,” esper.codehaus.org, 2010.

[37] Oracle, “Complex event processing - oracle,” www.oracle.com/technetwork/middleware/complex-event-processing/overview/complex-event-processing-088095.html, 2010.

[38] StreamBase, “Streambase complex event processing (cep),” www.streambase.com, 2010.

[39] M. Corp., “Microsoft streaminsight,” msdn.microsoft.com/en-us/library/ee362541.aspx, 2010.

[40] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, and R. Sears, “Mapreduce online,” in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*. USENIX Association, 2010, p. 21.

[41] X. Fei, S. Lu, and C. Lin, “A mapreduce-enabled scientific workflow composition framework,” *Web Services, IEEE International Conference on*, vol. 0, pp. 663–670, 2009.