A Continuous Workflow Scheduling Framework [1]

Panayiotis Neophytou, Panos K. Chrysanthis, Alexandros Labrinidis
Department of Computer Science, University of Pittsburgh
Pittsburgh, PA, USA
{panickos, panos, labrinid}@cs.pitt.edu

ABSTRACT

Traditional workflow management or enactment systems (WfMS) and workflow design processes view the workflow as a one-time interaction with the various data sources, i.e., when a workflow is invoked, its steps are executed once and in-order. The fundamental underlying assumption has been that data sources are passive and all interactions are structured along the request/reply (query) model. Hence, traditional WfMS cannot effectively support business or scientific monitoring applications that require the processing of data streams such as those generated nowadays by sensing devices as well as mobile and web applications.

Our hypothesis is that WfMS, both in the scientific and business domains, can be extended to support data stream semantics to enable monitoring applications. This includes the ability to apply flexible bounds on unbounded data streams and the ability to facilitate on-the-fly processing of bounded bundles of data (window semantics). In our previous work we have developed and implemented a Continuous Workflow Model that supports our hypothesis. This implementation of a CONtinuous workFLow ExeCution Engine (CONFLuEnCE) led to the realization that different applications have different performance requirements and hence an integrated workflow scheduling framework is essential. Such a framework is the main contribution of this paper. In particular, we designed and implemented STAFiLOS, a STreAm FLOw Scheduling for Continuous Workflows framework within CONFLuEnCE and evaluated STAFiLOS based on the Linear Road Benchmark.

Categories and Subject Descriptors

H.2.8 [**DATABASE MANAGEMENT**]: Database Applications—*Scientific databases*; H.4.1 [**INFORMATION SYSTEMS APPLICATIONS**]: Office Automation—*Workflow management*

General Terms

Algorithms, Design, Data Streams, Continuous Queries

[1]This research was supported in part by NSF grant IIS-053453, NSF career award IIS-0746696 and NSF grant OIA-1028162.

SWEET'13, June 23, 2013, New York, NY, USA.

ACM 978-1-4503-2349-9/13/06 $15.00
http://dx.doi.org/10.1145/2499896.2499898.

1. INTRODUCTION

Many enterprises use workflows to automate their operations and integrate their information systems and human resources. Workflows have also been used to facilitate outsourcing or collaboration beyond the boundaries of a single enterprise, for example, in establishing Virtual Enterprises [6, 25]. In the context of scientific exploration and discovery, workflows have been used to orchestrate simulations and large scale and distributed data analyses [15, 23, 31, 10] as well. More recently, they have also been used towards supporting collaborative interactions among scientists [24, 2, 22].

A common class of applications, in both business and scientific domains, is monitoring and reactive applications that involve the processing of continuous streams of data (updates). Examples include financial analysis applications that monitor streams of stock data to support decision making in brokerage firms and environmental analysis applications that collect and analyze sensor data to support discovery of air or water pollution. The use of Continuous Queries (CQs) is the current popular approach in monitoring data streams, both in research (e.g., [7, 4, 8, 30, 9]) as well as in industry (e.g., [13, 16, 29]). However, CQs have three drawbacks if we consider them to support a workflow execution model: (1) they are stateless beyond a window operator's scope, (2) have a static configuration and (3) are unable to facilitate user interaction. These make CQs unsuitable as a complete solution for enabling monitoring and reactive workflow applications.

In our previous work [19, 18], we proposed a shift towards the idea of "continuous" workflows (CWfs) . The main difference between traditional and continuous workflows is that the latter are continuously (i.e., always) active and continuously integrating and reacting on internal streams of events and external streams of updates from multiple sources, at the same time and in any part of the workflow network. We have implemented our proposed CWf model as a prototype system, called CONFLuEnCE [21], which is short for CONtinuous workFLow ExeCution Engine. CONFLuEnCE was built on top of Kepler, an existing workflow management or enactment system (WfMS) [15], and can integrate backwardly with traditional workflows, so that it can support both traditional data sources, such as Data Base Management Systems (DBMS), and data stream sources, such as Data Stream Management Systems (DSMS) (Figure 1).

In the traditional workflow model, the goal is to generally process data in a data transformation pipeline, or perform a set of remote tasks which are interdependent and may involve multiple disparate, and local resources (computational, data, or human). This is achieved generally without worrying about performance, since the processes could be easily spawned in multiple systems and each request could be handled independently. Also the application requirements for those workflows do not have any time-critical constraints.

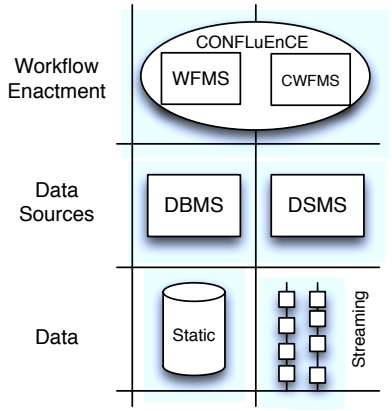

Figure 1: CONFLuEnCE Ecosystem

Scheduling in the traditional workflow model is mostly concerned with resource allocation and availability, and is typically delegated to the underlying Operating System.

As opposed to traditional workflow applications, CWf applications are more time-sensitive and have different performance requirements. These Quality of Service (QoS) requirements range from specifying a delay target, to keeping a fraction of results below a response time target, to minimizing tardiness. We also realized this fact, when implementing two applications on top of CONFLuEnCE, one from the business domain (a Supply Chain Management System [20]) and one from the scientific domain (AstroShelf [22]). The ability to facilitate *on-the-fly* processing of data that arrives at different rates and produces results within predefined time windows mandate better resource management than traditional WfMS. This is precisely the challenge that this work addresses by means of an integrated scheduling framework in CONFLuEnCE.

Specifically, in this work we focus on how to enable scheduling of the actors/tasks in a workflow, in order to better utilize the system resources and improve some metric (e.g., latency in the production of results). As opposed to Kepler's approach which implements a different model of computation (called *Director*) for every execution and recourse management model, our approach is to enhance CONFLuEnCE's CWf Director with the ability to support multiple scheduling policies in a *plug-and-play* manner. Towards this we have designed STAFiLOS, the *STreAm FLOw Scheduling for Continuous Workflows framework* within CONFLuEnCE. Basically, STAFiLOS implements an abstract scheduler that enables developers of CWf applications to easily incorporate new scheduling policies by implementing their abstract methods. The framework exposes many types of runtime statistics to the abstract scheduler such as actor runtime per invocation, input and output rates etc. These runtime statistics can be used by the developers to design new effective scheduling policies that implement smart resource allocation decisions.

Contibution: The contributions of this paper are summarized as follows:

1. Examine the current scheduling techniques used in traditional workflow systems.

2. Design a generic framework, called STAFiLOS, to enable the implementation of multiple scheduling policies for Continuous Workflows, and specifically for CONFLuEnCE.

3. Evaluate and compare the native Operating System thread-based execution with multiple STAFiLOS-based schedulers from the realm of Operating Systems and Data Stream processing, on a continuous workflow implementation of the Linear Road Benchmark.

Roadmap: In the next section, we provide background details on our CWf model and its implementation. Then, in Section 3 we describe how we designed and implemented STAFiLOS. In Section 4 we present our experimental evaluation of STAFiLOS. In Section 5, we discuss scalability and multiple CWf scheduling. In Section 6 we overview related work. Finally we conclude in Section 7.

2. BACKGROUND

In this section we first introduce the main aspects of the continuous workflow model and then go into some implementation details of the model into a working system. Both of these topics are described in full detail in [18].

2.1 Continuous Workflow Model

A *Continuous Workflow* is a workflow that is able to support enactment on multiple streams of data, by parallelizing the flow of data and its processing into various parts of the workflow. Continuous workflows can potentially run non-stop for an unlimited amount of time, constantly monitoring and operating on data streams.

Our proposed Continuous Workflow model exhibits the above characteristics by means of:

- Active queues on the inputs of activities which support windows and wave functions to allow the definition of synchronization semantics among multiple data streams.
- Pipelined concurrent execution of sequential activities.
- The ability to support push communication, i.e., receiving *push* updates from data stream sources.

A *wave* is a set of internal events associated with an external event and as such these internal events can be synchronized at different points of the workflow. A wave is initiated when an external event e_i enters the system, and is associated with a wave-tag which is e_i's timestamp t_i. When the external event e_i or any internal event in its wave is processed by a task, any new internal events produced by this task become part of the wave as well. Specifically, if the processing of the event with wave-tag t_i creates n events then these resulting events will have wave-tags $t_i.1, t_i.2, ..., t_i.n$. The wave-tag of the last event of the wave is marked as such. This is useful when a task downstream needs to synchronize all of the events belonging to a single wave. Moreover, a sub-wave may be formed when an event which is part of a wave is processed by a task. In this case a wave hierarchy is formed where an extra serial number is attached to the wave-tag. For example, if $t_i.3$ is involved in a task then the resulting m events will have wave-tags $t_i.3.1, t_i.3.2, ..., t_i.3.m$.

A *window* is generally considered a mechanism for setting flexible bounds on an unbounded stream of data events in order to fetch a finite, yet ever-changing set of events, which may be regarded as a logical bundle of events. We have introduced the notion of windows on the queues of events in workflows, which are attached to the activity inputs. The windows are calculated by a *window operator* running on the queue. The window operator will try to produce a window whenever it is asked by the attached workflow activity. When events expire they are pushed to an *expired items* queue which are optionally handled by another workflow activity. Five parameters are required to define the window semantics

Director	Actor Interaction	Computation Driver	Scheduling	Time based	QoS
SDF	Director: Topology-driven	Pre-compiled	Pre-compiled	N/A	N/A
DDF	Push	Data-driven	Iterative/Consumption Based	N/A	N/A
PN	Push	Data-driven	Thread/OS	N/A	N/A
DE	Director: Event Queue	Event-driven	Event Order	Yes (global)	N/A
CN	Director: Topology-driven	Pre-compiled	Pre-compiled	Yes (global)	N/A
CI	Push/Pull	Data-driven	Thread/OS	N/A	N/A
CSP	Push Synchronous	Data-driven	Thread/OS	Yes (global)	N/A
DT	Director: Topology-driven	Pre-compiled	Pre-compiled	Yes(global or local)	N/A
HDF	Director: Topology-driven	Pre-compiled	Multiple Pre-compiled	N/A	N/A
SR	Synchronous Reactive	Pre-compiled	Pre-compiled	Yes(global tick)	N/A
TM	Director: Priority Queue	Priority-based	Pre-emptive Priority-based	N/A	Priority
TPN	Push	Data-Time-driven	Thread/OS	Yes (global)	N/A
PNCWF	Push-Windowed	Data-Windowed-driven	Thread/OS	Yes (local)	N/A

Table 1: Taxonomy of Directors found in Kepler (first group) and ProlemyII (second group) as well as our PNCWF Director

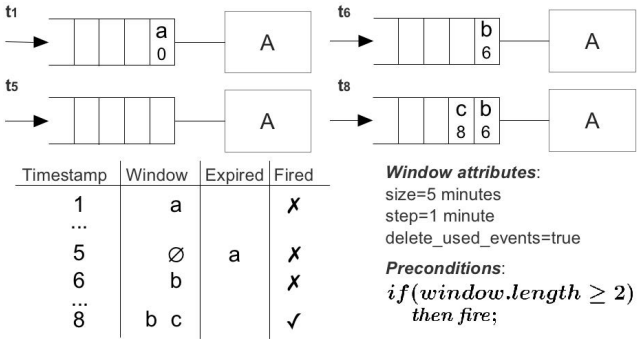

Figure 2: Window operator example.

for that operator: *size*, *step*, *window_formation_timeout*, *group-by* functionality, and *delete_used_events*. The window semantics definition along with the deleted_used_events flag can execute the hybrid window and consumption modes described in [1], such as *unrestricted*, *recent* and *continuous*, as those are combined with *tuple, time*, and *wave-based* windows. Even though not currently supported, wave-based windows can also be calculated according to a more generic framework, which defines nested sets according to nested relational algebra expressions [12]. A concrete example of a window definition with the *delete_used_events* flag usage is depicted in Figure 2.

A more detailed description of the concepts described above as well as the complete definition of the Continuous Workflow Model can be found in [19, 21].

2.2 CONFLuEnCE

CONFLuEnCE (CONtinuous workFLow ExeCution Engine) is the implementation of our CWf model on top of Kepler [15]. Kepler is a free open-source scientific WfMS, which was built on top of PtolemyII, a software system for modeling, simulating, and designing concurrent, real-time systems. The suitability of Kepler, for implementing our CWf model, comes from the fact that it decouples the specification of a workflow and the models of computation that govern the interaction between components of a workflow. This means that a workflow can be specified once and executed under different runtime environments (i.e., models of computations) which Kepler inherited from its underlying PtolemyII system [11]. Also, Kepler's code is inherently extensible, by providing a modular design, and this is also proven by the fact that is being actively developed by nearly twenty different scientific projects. Furthermore, programming workflows in Kepler is made easy for domain experts without any knowledge of programming structures. Kepler provides an intuitive high-level visual language for building workflows, where the designer can drag and drop components and connect inputs with outputs quite easily. Configuring parameters is easily done using dialog boxes and it also gives useful displays for debugging the workflows. Finally, Kepler was implemented in Java which simplifies our implementation of CONFLuEnCE.

The advantage of Kepler over other WfMS, which is to distinguish between the specification of a workflow and the model of computation, enables us, on one hand, to easily expand Kepler's workflow specification language to capture workflow patterns of our CWf model and on the other hand, to develop a new CWf model of computation. A workflow in Kepler is specified as a composition of independent components called *actors*. Actors have parameters used to configure and customize their behavior, which can be set statically, during the workflow design, as well as dynamically, during runtime. Communication between them happens through interfaces called *ports*. These are distinguished into input ports and output ports and the connection between them is called a *channel*. As part of the communication between the two ports, a data item (referred to as token in Kepler) is propagated from the output port of one actor to the input port of the receiving actor. The receiving point of a channel has a *receiver* object, which controls the communication between the actors. The receiver object is not provided by the actor but by the workflow's controlling entity, called the *director*. The director defines the execution and communication models of the workflow. As such, the communication being synchronous or asynchronous (buffered) is determined by the designer of the director, not by the designer of the actor. Various models of computation which can be found in Kepler and PtolemyII are listed along with their characteristics in Table 1.

CONFLuEnCE was implemented within Kepler as a new model of computation (i.e., as another module). This module implements all the necessary constructs which enables Kepler to run continuous workflows. This includes a *Continuous Workflow Director*, a *Windowed Receiver* and *Timing components* (i.e., timestamped event objects and actor timekeepers). Our Continuous Workflow (PNCWF) director is based on Kepler's PN, CN and DE directors. PNCWF implements a generic interface to support the timing constructs (windowed receivers, timekeepers etc.) and support concurrent execution of sequential activities by means of OS threads. Specifically, it enables concurrent execution by wrapping every actor in its own thread, allowing them to run in parallel and blocking them whenever there are no more data to consume.

Although having queues on the inputs of actors to buffer data is a feature already implemented in certain models of computation in Kepler, window semantics on these queues do not exist in any model of computation. We have implemented a new generic type of receiver which can be associated with continuous workflow directors that implement the CWF model. This new type of receiver defines windows by size and step, as well as other parameters such as Group-by clauses. When adding a token into this receiver the generic `put()` method is used. This method encapsulates the token into a timestamped and wave-stamped event as they are dictated by the timekeeping components. Then it inserts the event into the appropriate queue, after evaluating the group-by clause. Within the same call it also checks to see if a new window is produced and if it does then it stores it into the output queue. When the actor, to whom this receiver belongs to, calls the `get()` method, a window from the output queue is returned. The timing between the `put()` and `get()` methods depend on the director's execution model.

Finally, in order to support push communications on continuous workflows, we have implemented various actors which are able to connect to external data streams (through TCP or HTTP connections). As data are pushed into those connections from the sources these actors pump it into the workflow's internal ports at a rate which is again dictated by the director's execution model.

3. STAFILOS: STREAM FLOW SCHEDULING FOR CONTINUOUS WORKFLOWS

The PNCWF director we described earlier, is thread-based, thus resource management and allocation to the various threads is handled directly by the Operating System. This leaves no margins for QoS based optimizations, which are suitable for monitoring applications. Since in Kepler execution is dictated by the director component, we could have implemented specific scheduling policies in different CWf director implementations. Instead of that we adopted a slightly different philosophy. We designed a framework to integrate scheduling through a generic and pluggable scheduled CWf director that can be plugged with different scheduling policies. We applied what we learned from implementing the PNCWF director, the Windowed Receiver, the time keeping method and the token encapsulation inside CWEvents, and reused these components within STAFiLOS, while extending them specifically to work in this new execution model while at the same time supporting the previous one. We also added a more generic actor statistics module, which can now be used by any CWf scheduler within STAFiLOS, to provide runtime statistics on a number of different metrics. Specifically the statistics module keeps track of the cost of each actor (i.e., time per invocation), actor input rates and actor output rates, which are in turn used to calculate the selectivity of the actor. These statistics are dynamically calculated during runtime and are updated with each actor's invocation.

STAFiLOS is composed of three main components:

- The Scheduled CWF Director.
- The TM Windowed Receiver.
- The Abstract Scheduler.

All three components and their interactions, described below, are depicted in Figure 3.

The *Scheduled CWF* (SCWF) director is the main component that interacts with the workflow model (i.e., actors, ports, subworkflows) and the management modules ran by Kepler. It is responsible for initializing the actors, ports, receivers and the scheduler, as well as transitioning the workflow model through the various execution stages within each iteration. The SCWF director is schedule-independent, thus a scheduling policy implementation, which extends the Abstract Scheduler, is being enacted by it.

The *TM Windowed Receiver* is based upon the TM Receiver of the TM PtolemyII domain, but extends our Windowed Receiver implementation described in Section 2. The TM Windowed Receiver interacts with the SCWF director as shown in Figure 4. When an upstream actor produces an event on its output port it broadcasts it to all the remote downstream receivers connected to it. The TM Windowed Receiver extends the `put()` and `get()` methods of the Windowed Receiver. When an event is passed to the `put()` method, it is propagated to the Windowed Receiver's `put()` method, which in turn is queued in the appropriate group-by queue. During the same call, the window semantics are evaluated on that queue and if a window is produced it is returned to the TM Windowed Receiver `put()` method. The produced window is then enqueued at the actor's ready queue at the SCWF director. When the director decides to run that actor (Actor B in the example), it dequeues the event and adds it to a buffer inside the TM Windowed Receiver, rendering it available at the next `get()` call by the `fire()` method of the actor. Besides the regular events being queued at the director, the windowed receivers that compute timed windows also register "window timeout events" which are used to produce timed windows before an event from the next window arrives to close and produce the current window.

The *Abstract Scheduler* component implements most of the basic functionality of a scheduler but it is not a complete scheduler. However, it can be extended and made fully functioning by an actual scheduler implementation. The Abstract Scheduler maintains a list of the workflow's actors, and maps them to queues of events (sorted by timestamp) that should be propagated to each actor's corresponding input ports when they are to be scheduled for execution. It also maintains a mapping between actors and their current state as well as a list of flags denoting whether a state is valid or not.

Three states are defined: ACTIVE which denotes that the actor can be considered for firing at the current iteration, WAITING which denotes that the actor is waiting for something to happen within the scheduler before it can be run, and INACTIVE which denotes that the actor currently has no events to process. State transition rules are implemented within each scheduler implementation. Finally, the Abstract Scheduler keeps two priority queues. One for the active actors (active queue) and one for the actors who are waiting (waiting queue). Basically when an actor switches state from being ACTIVE to WAITING, it is removed from the active queue and it is placed in the waiting queue. If an actor is INACTIVE it is not placed in any of the priority queues. The `getNextActor()` method returns the next actor from the active priority queue. The priority queues are sorted based on a function implemented inside a QueueComparator object which is provided by the scheduler implementation. This comparator could be based on actor priorities defined by the workflow designer or some kind of dynamic priorities calculated at runtime based on the actor statistics.

The Abstract Scheduler also provides hooks where the director can signal the scheduler for the director's state changes, such as the start and end of a director's iteration, the start and end of an actor's iteration etc.

3.1 Implemented CWF Schedulers

As first case studies, we used the STAFiLOS scheduling framework to implement three schedulers with different characteristics; the *Quantum Priority Based scheduler*, the traditional fair *Round Robin* scheduler and a *Rate Based* scheduler proposed for CQs. Our aim was to assert the expressive power of STAFiLOS in implementing different scheduling policies.

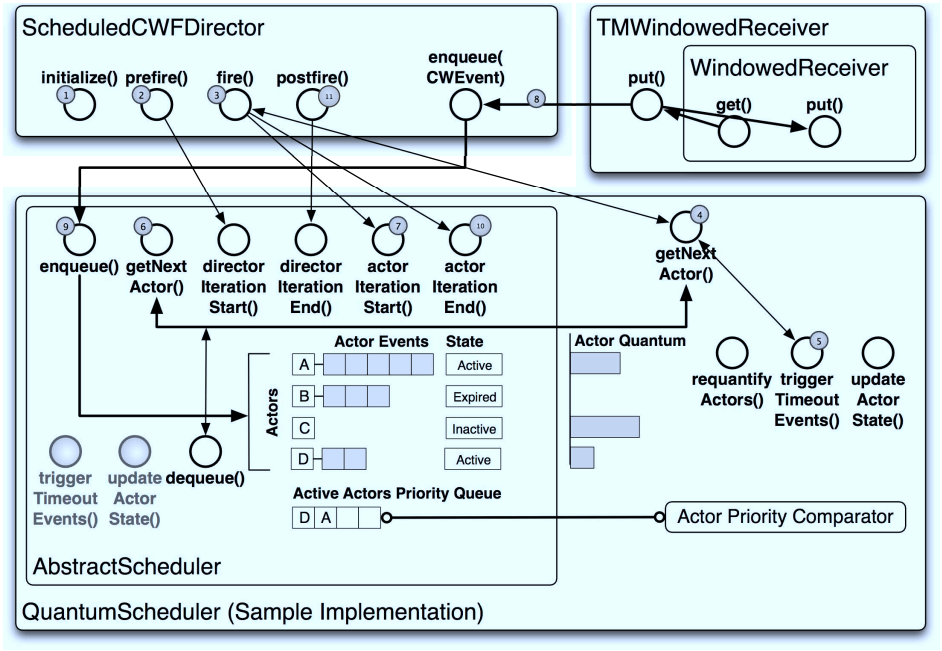

Figure 3: The STAFiLOS scheduler framework in CONFLuEnCE.

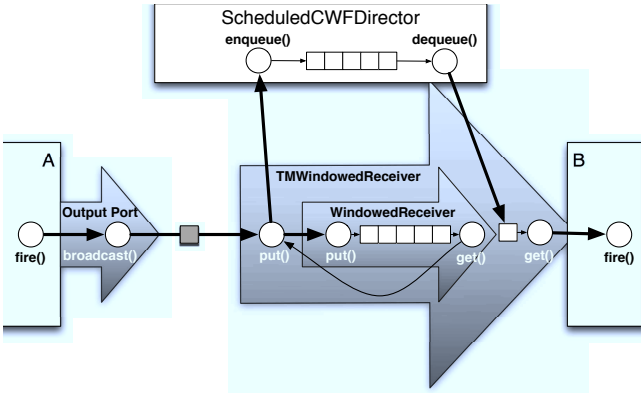

Figure 4: Event flow in the STAFiLOS scheduling framework.

3.1.1 The Quantum Priority Based Scheduler (QBS)

The Quantum Priority Based Scheduler is largely based on the Linux process scheduler [5]. The actors are assigned priorities by the workflow designer and based on those priorities the scheduler assigns a number of basic quanta, as given by Equation 1.

$$q = \begin{cases} (40 - p) \times b, for\ p >= 20 \\ (40 - p) \times 4b, for\ p < 20 \end{cases} \quad (1)$$

Where, p is the actor's priority, b is the basic quantum (a scheduler static parameter), and q is the quantum to be given to the actor whenever a re-quantification process is initiated. Source actors are treated independently of the rest of the actors in order to regulate better the flow of data coming into the workflow. Correctly tuning the scheduling policy regarding the source actors can play a significant role in the overall behavior of the QoS metrics. In the case of QBS the source actors are being scheduled in regular intervals (i.e., after x internal actor invocations).

The quantum value for each actor represents its allowance in microseconds that can run before the next re-quantification period. Actors that have events ready to be processed are divided into *active* and *waiting* depending on wether they have a positive quantum or not. The active actors are sorted by ascending priority. If two actors have the same priority then they are treated as FIFO. When an active actor runs for a while and suddenly runs out of quantum it is moved into the waiting queue. Once all the actors with events run out of quanta and are moved into the waiting queue, the scheduler initiates a re-quantification process and swaps the two queues (i.e., the waiting queue becomes the active queue and vice-versa). There is a possibility that an actor consumed more than its remaining quantum in its last iteration and ended up having a negative quantum. If that negative value is significant, there is a chance that even after re-quantification, it still has a negative quantum value. In that case it stays in the waiting queue. An actor that processed all of its ready events transitions into the *inactive* state and its quantum value is preserved until new events become available. The state conditions for an actor A in QBS are shown in Table 2.

We now explain the interactions between the components as depicted in Figure 3 which are also applicable to any scheduler implemented within the STAFiLOS framework. When the execution of the workflow begins, the director carries out the *initialization* of all the components. It also signals the scheduler, in order for it to carry out its own initialization. As part of the step, the source actors are being registered to the scheduler, which, depending on the implementing policy, it decides how to treat them. After that, the director enters the director iteration cycle, first with the *pre-fire* state, again while signaling the scheduler about it. Next, at the *fire* state the director calls the scheduler's `getNextActor()` method to get the next actor to be fired. At this point the scheduler polls the next actor from the active queue, which is sorted using the Comparator attached to the active queue. The Comparator implements the scheduler's priority function. The selected actor might be a source, an

		QBS and RR Schedulers	RB Scheduler
ACTIVE	A is not a source actor	Has events waiting in its queue **AND** has a positive quantum value	Has events waiting in its queue
	A is a source actor	Has a positive quantum value **AND** has not fired yet in the current director iteration	Has not yet fired in the current period
WAITING	A is not a source actor	Has events waiting in its queue **AND** has a negative quantum value	Has no events waiting in its queue **AND** has events waiting in the next period buffer
	A is a source actor	Has a negative quantum value **OR** has fired in the current director iteration	Has fired in the current period
INACTIVE	A is not a source actor	Has no events waiting in its queue.	Has no events waiting in its queue or buffer.
	A is a source actor	*A source actor does not transition into this state.*	

Table 2: State conditions for an actor A in the different schedulers.

internal or an output actor. If it is an internal or output actor then an event from the corresponding actor's event queue is dequeued and placed on the actor's input port. Then the director pre-fires the actor and if that returns true, it goes on to fire the actor while starting the necessary timers to measure the cost of the actor.

During the actor's firing, new events will be produced at its output ports. The events go through the flow we explained in Figure 4, and end up being enqueued at the scheduler. An event has a reference to its corresponding actor and, based on that, it is enqueued on the actor it belongs to. At the same time, the actor's input rate as well as the producing actor's output rate statistics are being updated. At this point the actor's state is updated. If it was inactive, the scheduler will re-evaluate its state (e.g., assign a quantum to it and put it in the active queue). Once the actor post-fires, the director notifies the scheduler in order for it to calculate its cost and other statistics it needs to function.

The director's iteration cycle ends when a call to the method `getNextActor()` returns `null`. That's when the director post-fires, notifies the scheduler and restarts the iteration. At this point the scheduler usually performs some maintenance tasks (e.g., re-quantify the actors, recalculate their states, update statistics etc.)

3.1.2 Round-Robin Scheduler (RR):

The Round Robin scheduler works in similar manner with the QPB scheduler. It does not take into account any priorities though. At each scheduling period it gives the active actors a time slice (quantum) on which they are allowed to run. They are then scheduled to process their available events in a round robin manner. If they manage to process all of their current events they transition to the *inactive* state and give up any remaining slice. If they consume their slice they transition to the *waiting* state, and remain in that state until the next period, to process the remaining of their available events. New events can be added to an actor's ready queue even within the current period. The actor processes them if it has enough time to do so during the current period. If an actor is inactive and new events arrive, a slice is assigned to it and the actor is placed at the end of the Round-Robin queue. The state conditions for an actor A in the RR Scheduler are shown in Table 2.

3.1.3 Rate Based Scheduler (RB)

The third scheduler we have implemented is the Rate Based Scheduler which is based on the Highest Rate scheduler described in [28]. The Highest Rate scheduler is the best performing scheduler for CQs with respect to average response time.

The actors are once again divided into active and waiting, and their priorities are dynamically calculated based on their selectivity and cost as shown in Equation 2.

$$Pr(A) = S_A \,/\, \overline{C}_A \tag{2}$$

$Pr(A)$ is the dynamic priority of actor A. S_A is the actor's global selectivity, and $\overline{C}_A$ is the actor's global average cost, as they are defined in [28]. When an actor is shared among multiple workflow paths (i.e., is connected to more than one downstream actor) then we add up the downstream global costs and global selectivities of each path.

Event processing in this scheduler is divided into periods. At each period the scheduler processes all the events that have been enqueued during the previous period. Any newly enqueued events are kept in a buffer and are put into their corresponding actor's queues once the current period is over. The end of a period is signaled by the director's end of iteration, which happens when the active actors queue becomes empty. The active actors queue is empty when all the actors have no more events to process and all the source actors have executed once during the current period. The dynamic priorities are re-evaluated at the end of each period. The state conditions for an actor A in the RB Scheduler are shown in Table 2.

4. EXPERIMENTAL EVALUATION

The primary goal of our preliminary evaluation is to determine if there are any performance penalties in implementing scheduling policies using our STAFiLOS framework. The secondary goal was to get a better inside on the effectiveness of a QoS-based scheduler, compared to a native OS scheduler being used in traditional WfMS.

Almost all QoS-based schedulers utilize an optimization metric which is defined in terms of a *delay or latency target*. Broadly, these require that a specified fraction (0-100%) of results be produced under the delay target. For example, in keeping average response time below delay target, no fraction is specified (best effort) whereas in meeting strict deadlines, the fraction is 100%.

As a first approximation for all these metrics, we carried out a stress test to identify the supported delay target space of each scheduler by varying the input rate in the context of the Linear Road benchmark [3].

4.1 Experimental Setup

We evaluated the STAFiLOS framework by running the various schedulers on a continuous workflow implementation of the Linear Road benchmark [3]. The Linear Road has been established as the standard benchmark for stream processing systems and endorsed by the developers of the two first DSMSs, namely, Aurora [7] (a collaboration among Brandeis University, Brown University and MIT) and STREAM [17] (from Stanford University). A detailed description of our continuous workflow implementation of the linear road benchmark can be found in Appendix A.

We used the workload generator provided on the Linear Road website[1] to generate car position reports for 0.5 expressways (Figure 5). All the experiments were ran three times each (results

[1]http://www.cs.brandeis.edu/∼linearroad/

Workload L-rating	0.5 highways
Experiment duration	600 sec
QBS Source scheduling interval	5 internal actor iterations
Basic Quantum (QBS) (μs)	500, 1000, 5000, 10000, 20000
Basic Quantum (RR) (μs)	5000, 10000, 20000, 40000
Priorities used (QBS)	5, 10

Table 3: Experimental setup

show the average of the three runs) on the same machine configuration, always one at a time with the system being exclusively used for our experiments. The system used was a dual Pentium Intel Xeon E5345 at 2.33GHz with a total of 8 cores of 4MB cache each and 16GB of main memory. Since CONFLuEnCE is implemented in Java, the virtual machine was allocated 8GB of heap space.

The schedulers used in our evaluation are the ones implemented within the STAFiLOS framework and described in the previous section, namely, the Round-Robin (RR), Quantum Based Source (QBS) and the Rate Based (RB) schedulers. As a baseline for our comparison we use the *Thread Based* (PNCWF) scheduler which is implemented in the PNCWF Director described earlier in Section 2.2. The PNCWF scheduler uses the director developed in [21] and everything needed to support it (such as an extension of the Windowed Receiver adjusted to run in a threaded environment). During initialization of a workflow each actor is associated with a thread based controller to transition it through the iteration phases (initialize, pre-fire, fire, post-fire). The actor thread blocks, when trying to read from its input ports which have no events available, until a window or event is produced. The timeout of timed windows (described in [21]) is handled by the actor thread that is waiting to read from an input port, by waiting only for the amount of time defined by the timeout. Once the timeout is reached, without the Windowed Receiver producing a window, the thread raises the timeout flag on the receiver and forces it to produce a window.

The different parameters we used for configuring the experiments are listed in Table 3. The source scheduling interval listed for QBS means that for every five internal actor firings one source actor firing is scheduled. This ensures that the input data are smoothly inserted into the workflow. The basic quantum values listed for QBS and RR correspond to the q value and slice values respectively as described in Sections 3.1.2 and 3.1.1. The priorities correspond to individual priorities given to the actors, that are taken into account when QBS is running. The highest priority of 5 is given to the actors that handle the immediate output of the workflow. Regarding the tolls those are the *TollCalculation* and *TollNotification*, and regarding the accident notifications those are the *AccidentNotification* and *AccidentNotificationOut*. A priority of 10 was given to the actors relevant to statistics maintenance and accident detection.

4.2 Experimental Results

Experiment 1: Sensitivity Analysis of RR. Figure 6 shows how Round Robin behaves when setting different quantum values. Generally, the scheduler behaves almost the same for the various time slots with the best being 20,000μs which keeps a generally lower response time throughout the experiment until eventually thrashes with the 40,000μs case.

Experiment 2: Sensitivity Analysis of QBS. Figure 7 shows how the Quantum Priority Based Source scheduler behaves with different basic quantum values set. As you may recall, the basic quantum is the value of b in Equation 1.

From the results we see that a basic quantum of 500μs performs the best throughout the experiment compared to the other values. This is due to the fact that having high quantum values given to the

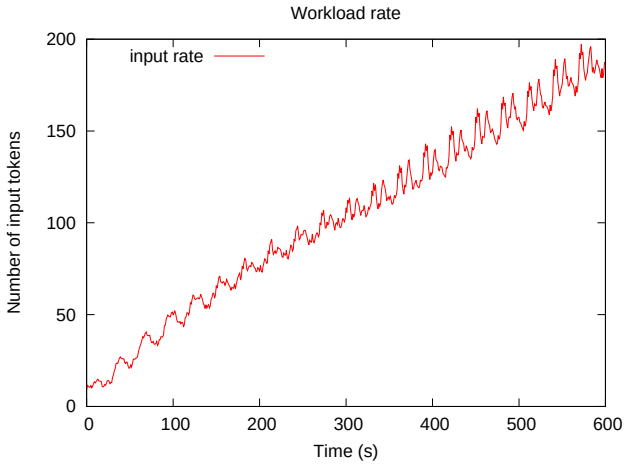

Figure 5: Workload of 0.5 highways.

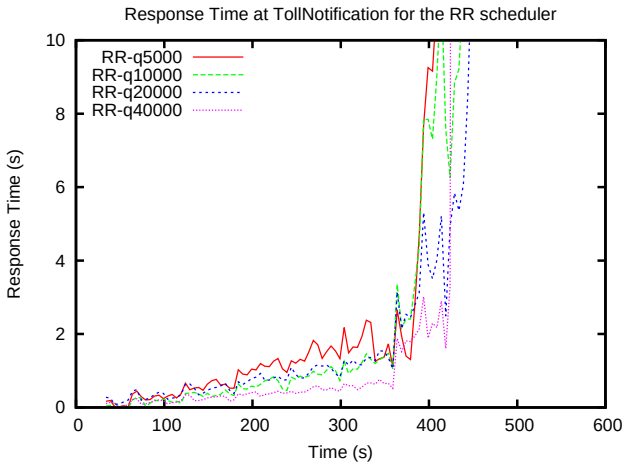

Figure 6: Response Times of the RR scheduler using varying basic quantum values.

actors results in having just a priority based FIFO queue, where each priority class is a FIFO queue with each actor exiting the queue only when it is done processing all of its current events. So a small enough value in this case is adequate.

What is interesting here is that a basic quantum of 5000μs performs worse than one with 10000μs. We attribute this to the fact that in the case of 5000μs the re-quantification of all actors happens more often, resulting in low priority actors accumulating quantum, thus when it is their turn to run, and having also accumulated many events, they will end up starving higher priority actors, such as the output actors which we use to measure the average response time.

Experiment 3: STAFiLOS-based schedulers Vs. OS thread-based scheduler. Figure 8 shows the QBS and RR with the best performing parameters of 500μs and 40,000μs, respectively, from the previous two experiments, along with RB and PNCWF.

The figure shows that QBS and RR exhibit the best response times (under 2sec) until they thrash. The thread-based PNCWF has much lower capacity in terms of input rates, since it thrashed at 320sec when the input rate is about 120 updates/sec, as opposed to the rest of the schedulers which thrash at about 440sec where the input rate is 160 updates/sec. RB exhibits worst average response

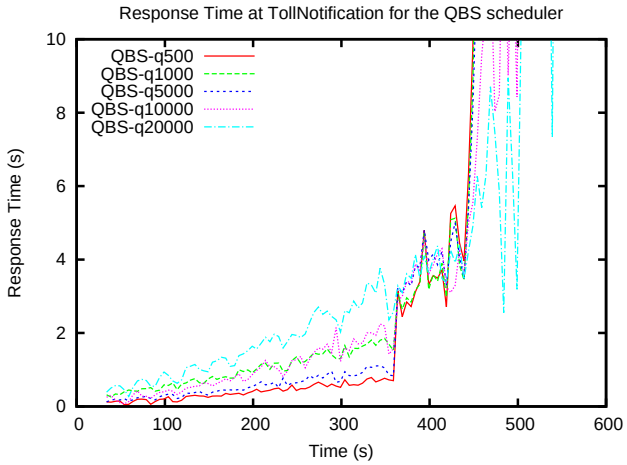

Figure 7: Response Times of the QBS scheduler using varying basic quantum values.

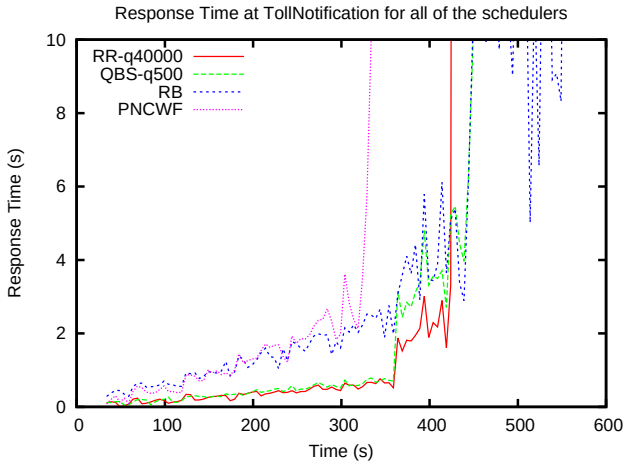

Figure 8: Response Times of all the main schedulers.

times because of the fact that it does not distinguish the source actors as high priority and neither independently schedules them in regular intervals, like the other schedulers. Thus tokens suffer from waiting for a longer period of time to enter the workflow.

4.3 Discussion

The above experiments clearly show that the schedulers implemented within the STAFiLOS framework have a higher rate tolerance and generally lower response times than Kepler's own Thread-Based director which relies on the underlying OS.

We generally based our CWf implementation of the Linear Road Benchmark on off-the-shelf actors that come with Kepler, which as can be seen from Figures 10-15 adds a great deal of complexity. Furthermore, the off-the-shelf actors lack any performance optimizations found in the CQ operators. As a result the RB scheduler did not perform as well as expected. Adding asynchronous I/O calls as well as implementing schedulers which are able to combine priorities with flow information would greatly improve performance. Moreover, providing a set of stream optimized atomic as well as composite actors, which can accumulate and compensate tokens which are added and expired from a sliding window, would help

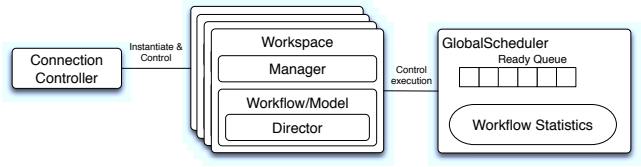

Figure 9: Multi-workflow execution framework

in avoiding redundant multiple aggregate computations and would greatly improve the performance of window-based actors.

Our experiments also reveal that STAFiLOS offers the scheduling flexibility required by monitoring applications within a Continuous Workflow Management System (CWMS) without compromising performance, since the STAFiLOS-based schedulers performed better than the Thread-Based one. The use of the Linear Road Benchmark further show that a CWMS by offering more functionality and flexibility compared to a DSMS might not exhibit the same scalability. However, scalability can be achieved by strategically integrating multiple DSMSs (as in Figure 1), that can be viewed as specialized source actors, to build more complex monitoring solutions, while being able to satisfy any application SLAs. The integrated DSMSs can potentially be tuned to also support load shedding under overloading situations [27, 26].

5. FUTURE WORK: SCALABILITY AND MULTIPLE CWF SCHEDULING

The current implementation of SCWF Director focused on optimizing a single workflow on a single node. To achieve higher levels of scalability we are considering two directions. First, the SCWF Director is made aware of the CPU cores topology in modern machines to balance the distribution of the ready actors queue to each core while considering data dependencies. The second one is a distributed version of SCWF. This version distributes the processing of a workflow among multiple computing nodes in a cluster or the Cloud by placing specific actors to specific nodes.

Beyond scaling a single workflow, we are also working on a multiple CWF processing model. Our current design for a single node is based on the idea of two-level scheduling (Figure 9). At the low-level, each individual workflow director implements its own local scheduler. At the top-level, the global scheduler manages the different workflow instances based on an appropriate CPU capacity distribution policy. It achieves this by allocating CPU resources to the different instances of the Manager class, a PtolemyII/Kepler module which manages the execution of a sinle workflow. It switches between the workflows using the Manager methods `initialize()`, `pause()`, `resume()`, `stop()`.

The ConnectionController is a new module we propose for controlling the execution of multiple workflows externally. When Kepler/Confluence is started in multi-workflow mode from the command line then the ConnectionController is instantiated and is listening for commands to manage running workflows as well as add and remove them from the running list.

Our plan is to first confirm our hypothesis that our proposed scheme is able to handle workflows with different priorities and different optimization metrics. Once we achieve scaling up, we will consider scaling out.

6. RELATED WORK

In current workflow management systems the scheduler defines a static schedule, which based on historical data and activity con-

sumption rates, should optimize the resource utilization of the system. Static approaches work well with the traditional workflow model since workflows are considered one-time interactions. In the more complex case of pipelined execution in data-flow based workflows, the activities run on their own threads and are managed by the operating system (usually by employing a Round Robin policy), which is oblivious to any special characteristics of each activity (e.g., token productivity, time to execute etc.)

Many WfMSs consider the task of running a workflow as combining a set of external services, choreographed using the workflow patterns. In order to do that the system has to find appropriate services that carry out the task of each activity, e.g., [14]. The external services, in addition to a description of their task, also carry a Quality of Service (QoS) characterization. In the context of WfMS and Operating Systems in general, the QoS metric measured is response time. Using this profile, the WfMS can compose the workflow instance in a way which satisfies the overall workflow request's QoS requirements (e.g., finish the whole workflow within the time limit). Similarly, [32] breaks the workflow into subsections, by categorizing the activities into branch or synchronization activities. It then distributes the remaining deadline to the subsections making sure that the workflow will finish before the deadline give a minimum execution time for each task.

The challenges here include the dynamic nature of the external resources and the unpredictable nature of the execution of the various patterns which the workflow is composed of. Considering a network of tasks composed as a workflow, each task is in a ready state once it has all the necessary data in its input ports that will make it complete one iteration. This decision of weather to run a task or not is made by processing a set of task preconditions. Then according to a priority function the scheduler will decide which task to execute next.

7. CONCLUSIONS

In this paper, we presented a generic framework, called STAFiLOS, that supports the implementation of different scheduling policies to meet the QoS requirements of continuous workflows in CONFLuEnCE (a CONtinuous workFLow ExeCution Engine). We evaluated STAFiLOS by compared the native OS thread-based execution with three STAFiLOS-based schedulers from the realm of OS and Data Stream processing on a continuous workflow implementation of the Linear Road Benchmark. The results of these experiments in conjunction with the implementation of the Linear Road Benchmark show that STAFiLOS enhances the expressive power and functionality of CONFLuEnCE by providing the means to meet the QoS requirements of a continuous workflow.

8. REFERENCES

[1] R. Adaikkalavan and S. Chakravarthy. Seamless event and data stream processing: Reconciling windows and consumption modes. In *Database Systems for Advanced Applications*, *LNCS* 6587, pp. 341-356. 2011.

[2] ADMT Laboratory. CMPI/Data Echange Server. http://www.dataxs.org, 2010.

[3] A. Arasu et al. Linear Road: a stream data management benchmark. In *VLDB*, pp. 480-491, 2004.

[4] S. Babu and J. Widom. Continuous queries over data streams. *ACM SIGMOD Record*, 30(3), 2001.

[5] M. Beck. *Linux kernel internals*. Addison-Wesley, 1998.

[6] A. Berfield, P. K. Chrysanthis, I. Tsamardinos, M. E. Pollack, and S. Banerjee. A scheme for integrating e-services in establishing virtual enterprises. In *IEEE RIDE*, pp. 134-142, 2002.

[7] D. Carney et al. Monitoring streams: A new class of data management applications. In *VLDB*, pp. 215-226, 2002.

[8] S. Chandrasekaran et al. TelegraphCQ: continuous dataflow processing. In *ACM SIGMOD*, pp. 668-668, 2003.

[9] P. K. Chrysanthis. AQSIOS - next generation data stream management system. *CONET Newsletter*, 2010.

[10] E. Deelman et al. PEGASUS: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219-237, 2005.

[11] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity-the ptolemy approach. *Proceedings of the IEEE*, 91(1):127-144, 2003.

[12] J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, and J. V. den Bussche. Dfl: A dataflow language based on petri nets and nested relational calculus. *Information Systems*, 33(3):261 - 284, 2008.

[13] IBM System S. http://www-01.ibm.com/software/sw-library/en_US/detail/R924335M43279V91.html, 2008.

[14] M. C. Jaeger, G. Rojec-Goldmann, and G. Muhl. QoS aggregation for web service composition using workflow patterns. In *IEEE EDOC*, pp. 149-159, 2004.

[15] B. Ludäscher et al. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039-1065, 2006.

[16] Microsoft streaminsight, http://www.microsoft.com/sqlserver/2008 /en/us/r2-complex-event.aspx, 2008.

[17] R. Motwani et al. Query processing, resource management, and approximation ina data stream management system. Technical Report 2002-41, Stanford InfoLab, 2002.

[18] P. Neophytou. *Continuous Workflows: From Model to Enactment System*. PhD thesis, University of Pittsburgh, 2013.

[19] P. Neophytou, P. K. Chrysanthis, and A. Labrinidis. Towards continuous workflow enactment systems. In *CollaborateCom*, pp. 162-178, 2008.

[20] P. Neophytou, P. K. Chrysanthis, and A. Labrinidis. CONFLUENCE: Continuous workflow execution engine. In *ACM SIGMOD*, pp. 1311-1314, 2011.

[21] P. Neophytou, P. K. Chrysanthis, and A. Labrinidis. CONFLUENCE: Implementation and application design. In *CollaborateCom*, pp. 181-190, 2011.

[22] P. Neophytou, R. Gheorghiu, R. Hachey, T. Luciani, D. Bao, A. Labrinidis, E. G. Marai, and P. K. Chrysanthis. Astroshelf: understanding the universe through scalable navigation of a galaxy of annotations. In *ACM SIGMOD*, pp. 713-716, 2012.

[23] T. M. Oinn et al. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045-3054, 2004.

[24] C. Paszko and C. Pugsley. Considerations in selecting a laboratory information management system (LIMS). *American Laboratory*, 32(18):38-43, 2000.

[25] O. Perrin and C. Godart. A model to support collaborative work in virtual enterprises. *DKE*, 50(1):63 - 86, 2004.

[26] T. N Pham, P. K. Chrysanthis, and A. Labrinidis. Self-managing load shedding for data stream management systems. In *IEEE SMDB*, 2013.

[27] T. N. Pham, L. A. Moakar, P. K. Chrysanthis, and A. Labrinidis. Dilos: A dynamic integrated load manager and scheduler for continuous queries. In *IEEE SMDB*, pp. 10-15, 2011.

[28] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Algorithms and metrics for processing multiple heterogeneous continuous queries. *ACM Trans. Database Syst.*, 33(1), 2008.

[29] I. StreamBase. Streambase: Real-time, low latency data processing with a stream processing engine. http://www.streambase.com, 2006.

[30] P. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE TKDE*, pp. 555-568, 2003.

[31] J. Wang, D. Crawl, and I. Altintas. Kepler + hadoop: a general architecture facilitating data-intensive applications in scientific workflow systems. In *ACM WORKS*, pp. 1-8, 2009.

[32] J. Yu, R. Buyya, and C.-K. Tham. Cost-based scheduling of scientific workflow application on utility grids. In *e-Science*, pp. 140-147, 2005.

APPENDIX

A. LINEAR ROAD BENCHMARK

Linear Road simulates a toll system for the motor vehicle expressways of a large metropolitan area. The tolling system uses "variable tolling": an increasingly prevalent tolling technique that uses such dynamic factors as traffic congestion and accident proximity to calculate toll charges. Linear Road specifies a variable tolling system for a fictional urban area including such features as accident detection and alerts, traffic congestion measurements, toll calculation and historical queries. For the purpose of our evaluation we only focused on the stream processing aspect of the benchmark and thus we excluded the historical queries.

The application provides a single feed of car position updates. Each car updates its position every 30 seconds. That includes its position (expressway id, direction, lane, segment of the highway) and current speed. While the workflow processes this feed, it is required to provide notifications to the cars about their toll charges every time they switch segment, based on a set of conditions. It also needs to alert them of any accidents which happened down the road in order for them to exit the highway and choose another route.

Our workflow implementation of the Linear Road benchmark consists of two levels of workflow hierarchy. The top level consists of all the major tasks and wiring between them, required by the application, and is governed by a Continuous Workflow director (either a STAFiLOS based one or the thread based PNCWF Director). The second level of the hierarchy consists of sub-workflows of main tasks in the top level and represent tasks like detecting stopped cars, calculating the number of cars in each segments etc. These are all governed either by SDF directors (if the consumption and production rate is constant) or by DDF directors (if the consumption and production rates are more fluid, e.g., if the sub-workflow includes decision points and does not have constant production rates at the internal actors.) Our implementation also requires the support of a relational database to store statistics on the road congestion as well as the recent accidents detected.

The workflow, as shown in Figure 10, is divided into three main areas. One to take care of the accidents, one for calculating the segment statistics, and one for calculating and notifying the cars about their tolling charges.

A.1 Accident Detection and Notification

The accident detection consists of three composite actors. The first one is for detecting stopped cars. If a car reports the same location in 4 consecutive position reports then it is considered stopped.

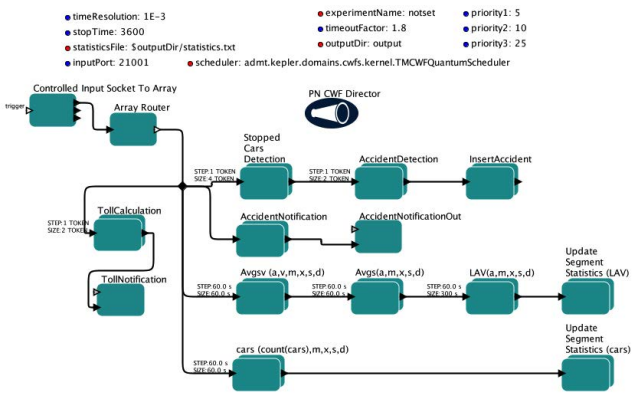

Figure 10: The Linear Road Benchmark top level workflow.

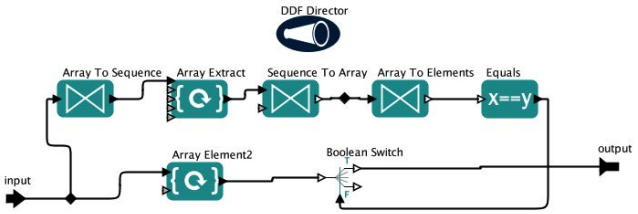

Figure 11: The Linear Road Benchmark accident detection sub-workflow.

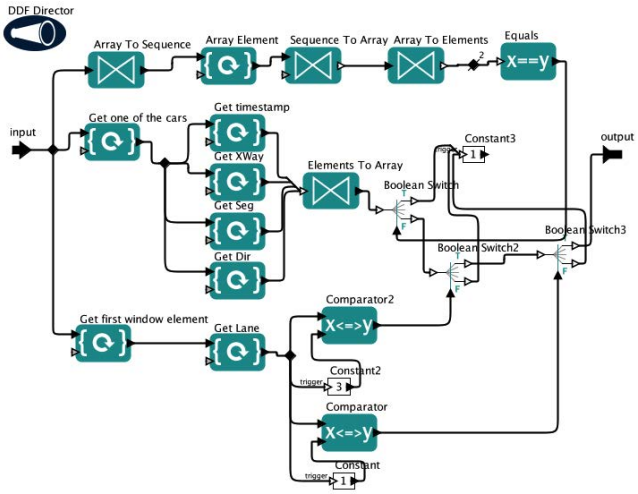

Figure 12: The Linear Road Benchmark accident detection sub-workflow.

The sub-workflow defining this functionality is depicted in Figure 11. The input port of this actor has the following window semantics: {Size: 4 token, Step: 1 token, Group-by: car ID}. When fired, this actor processes a window of the last four position reports of each car and compares the positions. If it is stopped then the actor outputs the first of those position reports and sends it to the Accident Detection actor.

The Accident Detection actor is the second one in this pipeline, and the implementation is shown in Figure 12. This actor takes windows of two position reports, which represent the same position, and compares the car IDs. If the car ids are different, and they are not in an exit lane, that means a car accident is in progress. The input port of this actor defines the following window semantics: {Size: 2 tokens, Step: 1 token, Group-by: position}. If an accident is detected then this is propagated to the Insert Accident actor which records the incident into the relational database. We omit the description of the third actor, because it just consists of constructing an INSERT statement and submitting it to the database.

The application also requires that any car entering a range of segments upwards an accident, be notified within 5 seconds of the position report. The notification is generated by the Accident Notification composite actor which, for each position report of a car, checks in the database to see if there is a car accident registered within four segments downstream of each car. The actor is shown in Figure 13.

A.2 Segment Statistics

The Toll Calculation formula relies on the system keeping some statistics regarding each segment of the expressway. Specifically,

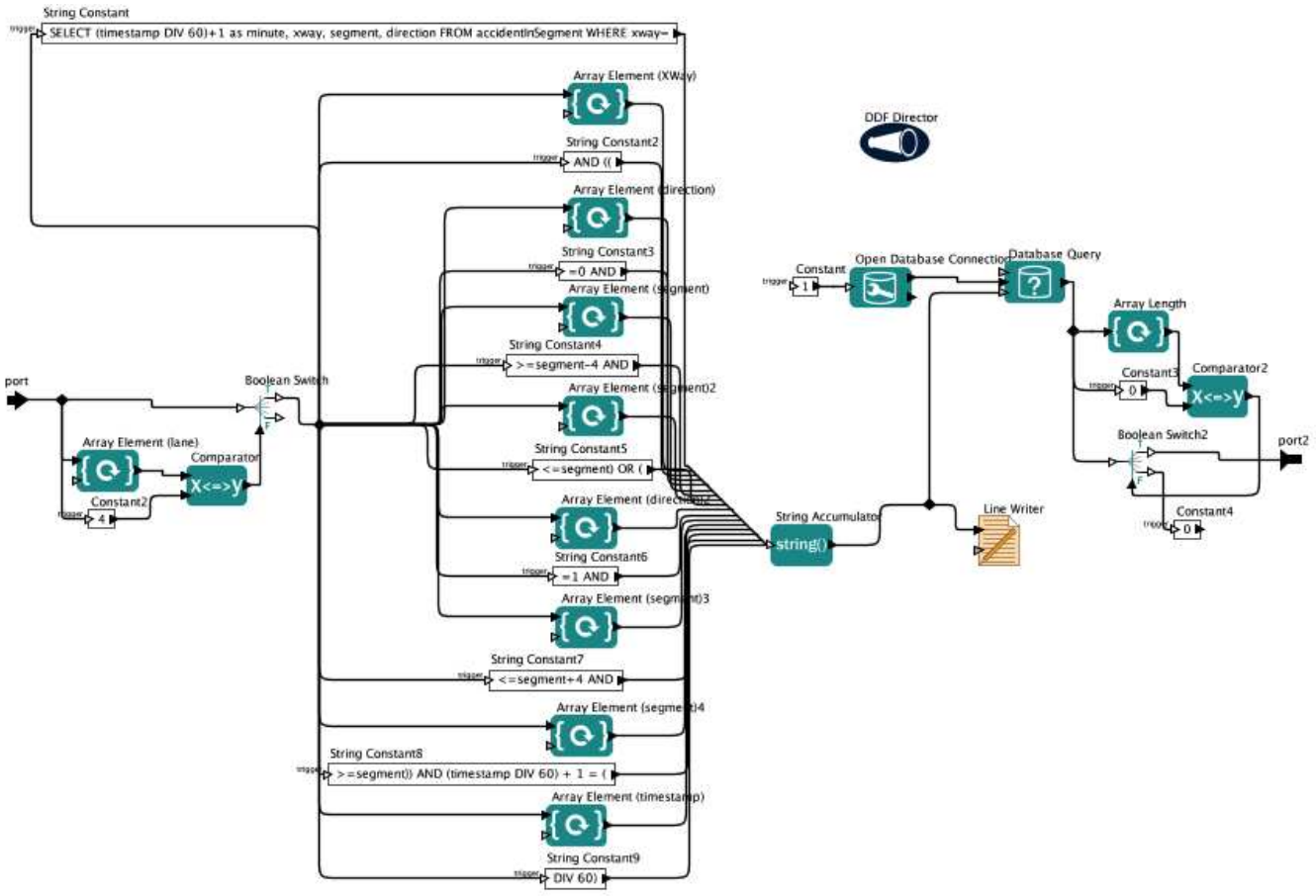

Figure 13: The Linear Road Benchmark accident notification sub-workflow.

the tolls depend on the number of cars present in a segment in the previous minute and the "Latest Average Velocity" (*LAV*) value for the segment. LAV is the average of the average speed of all the cars that passed through that segment every minute, for the past five minutes. In order to calculate the LAV value, we first calculate the average speed per car, for each segment, (*Avgsv* composite actor in Figure 14), and then the average speed of all the cars in the segment (*Avgs* composite actor).

The actor that calculates the average speed of a car has the following window semantic definition: {Size: 1 minute, Step: 1 minute, Group-by: Car ID, Expressway, Direction, Segment number}. The output from this actor is propagated to the *Avgs* actor which calculates the overall average speed per segment, per minute, and has the following semantics: {Size: 1 minute, Step: 1 minute, Group-by: Expressway, Direction, Segment number}.

The actor that calculates the number of cars per segment (*cars*), per minute, has the following window semantic definition: {Size: 1 minute, Step: 1 minute, Group-by: Expressway, Direction, Segment number}.

A.3 Toll Calculation and Notification

The toll calculation is initiated for each car whenever it switches from one segment to the next. To achieve this it has the following window semantics: {Size: 2 tokens, Step: 1 token, Group-by: Car ID}. Each time it is fired it processes a window containing the last two position reports of a car. If those reports have a different segment id then a new toll has to be calculated for that car. The calculation is done by querying the relational database table which keeps the segment statistics. The SQL query used to calculate the toll, for a specific car is the following:

```
SELECT
   CASE
      WHEN LAV < 40 AND numOfCars > 50 AND (
         SELECT COUNT(*)
         FROM accidentInSegment AS ais
         WHERE ais.xway = xway
            AND ais.direction=dir
            AND ((dir=1 AND seg <= ais.segment+4
            AND seg >= ais.segment) OR
               (dir=0 AND seg >= ais.segment-4
                AND seg <= ais.segment))
            AND ais.timestamp>=330-60
      ) = 0
      THEN 2*POWER((numOfCars - 50),2)
      ELSE
         0
   END as "Toll"
FROM `segmentStatistics`
WHERE xway=$xway
   AND seg=$segment AND dir=$direction
```

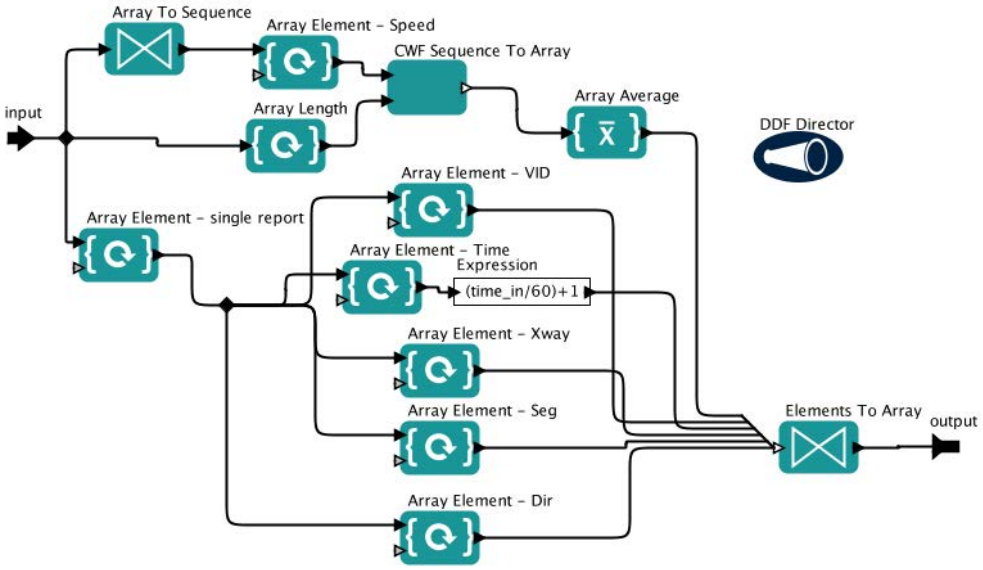

Figure 14: The Linear Road Benchmark car average speed sub-workflow (*Avgsv*).

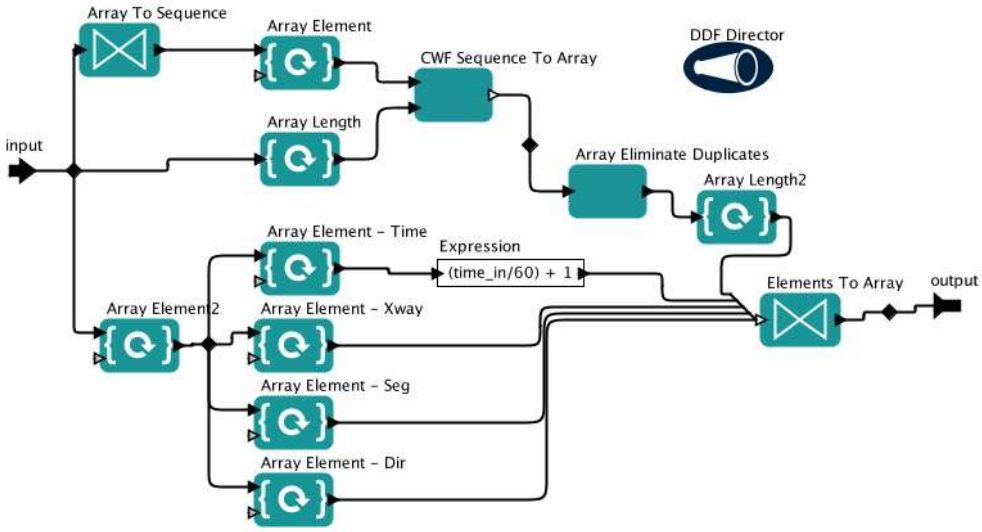

Figure 15: The Linear Road Benchmark car count sub-workflow (*cars*).