NIÑOS Take Five: The Management Infrastructure for Distributed Event-Driven Workflows

Siddharth Ganesan, Young Yoon, and Hans-Arno Jacobsen
{sid, yoon}@msrg.utoronto.ca, jacobsen@eecg.utoronto.ca
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada

ABSTRACT

Many workflows are inherently distributed over large-scale enterprise networks. These workflows involve many collaborating partners that interact in an autonomous and event-driven manner. Managing these workflows in an efficient and reliable manner is a challenging task. In this paper, we propose and evaluate the design for a distributed workflow management system that runs according to a set of protocols which utilize a content-based publish/subscribe messaging substrate. The novel features of our framework include elastic management of resources and flexible re-location of management entities to ensure cost-effective and low-latency management operations. Our experimental evaluation shows that the distributed and event-driven approach scales and maintains constant response time for varying management workloads. The average response time for management operations was reduced by 25% using our elastic management cluster approach compared to a fixed management cluster. Management requests were processed up to 10 times faster.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithms, Design, Experimentation, Management, Performance

1. INTRODUCTION

Many applications require interactions among collaborating partners that operate in different remote locations. For example, complex inter-departmental interactions (as shown in Figure 1) reported to us by an anonymous global electronics company, are not uncommon in the real world. Also, as part of recent initiatives focused on developing smart grids, companies such as Omicron [4] offer distributed control systems that enable interoperation among transmission towers based on the IEC standard protocol set [3]. Such interactions and interoperations can be defined as a workflow whose decomposed tasks can be deployed among the collaborating partners in a decentralized and event-driven runtime framework [26, 48, 34, 40, 49]. In general, the distributed interaction in a collaborative application is referred to as *distributed workflow* or *service choreography* [36].

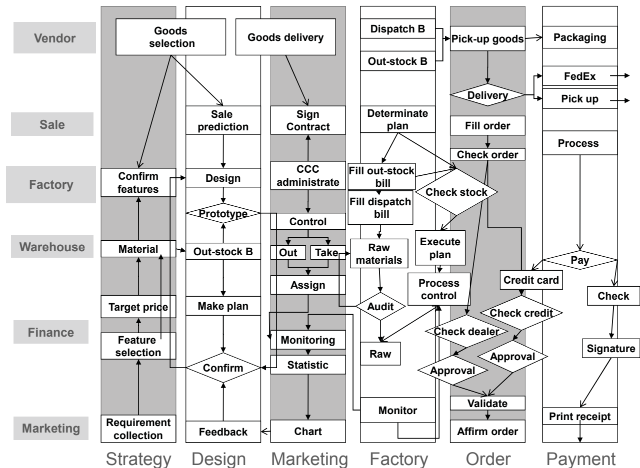

Figure 1: A sketch of a distributed workflow example from a global electronics manufacturing company.

Much research has focused on the development of distributed workflow concepts, including, but not limited to, standardization of the languages for describing interactions such as WS-CDL [43] and for the coordination of interactions such as WS-Coordination [44]; approaches for verification of workflow specifications [14, 18, 21, 29, 39, 11]; entire runtime frameworks [26, 48, 34, 40, 49]; and reliable decomposition techniques [15, 10, 33, 23].

However, related approaches lack insight into the issues involved in the reliable and efficient management of distributed and event-driven workflows, even though the management of these workflows is imperative for a number of reasons. For example, the timely and safe runtime control over workflows such as pause and resume is necessary for immediate handling of any anomalous situations. Discovery of the workflow status is also a critical feature for administrators to check the conformance of the workflow to its specification. While most commercially available centralized workflow processing systems [2, 5, 6, 9] support administrative functions, distributed workflow processing systems often lack equivalent functionality.

The management of distributed workflows is challenging in many respects. First, distributed workflows can typically scale to hundreds of collaborating partners and thousands of concurrent instances, which may generate large management workloads. Hence, management clients may have to issue thousands of status update

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DEBS'11, July 11–15, 2011, New York, New York, USA.
Copyright 2011 ACM 978-1-4503-0423-8/11/07 ...$10.00.

inquiries for the purpose of proactively monitoring for failures. Second, the management workload can dynamically change over time and space. For example, in a smart grid, a sudden spike in the power usage may trigger events that are sent to the back-up power generators to produce additional energy. These events will result in a temporary increase in the management workload. Third, interference can occur amongst concurrent management operations which may result in costly damages. Suppose, two management operations of updating a variable associated with some running workflow instance are issued from two different sources. Not processing the operations in order, when they are planned to be executed in sequence, is a violation of the management logic. Lastly, management operations on distributed workflows are sensitive to network delays, as workflow tasks are dispersed across geographically remote locations. The network delays, for example, can hamper the timely discovery or the halting of an anomalous task.

A centralized management system is not sufficient to effectively meet the aforementioned challenges. Centralized management systems have scaling limits when handling large-scale distributed workflows (like the ones shown in Figure 1). They are vulnerable due to the single-point of failure they represent. Also, there can be cases where a single centralized management entity is not desired at all. For example, in a B2B scenario, delegating management functions to one of the collaborating partners or an external provider, may not be desirable for all interacting partners; rather a fully decentralized management approach, where each partner manages its part of the workflow would be a desirable scenario. Even for the workflows within a single business domain, multiple distributed management entities are necessary, if autonomous and localized control over the tasks at different remote locations is required.

In this paper, we introduce a novel *distributed* management system that conforms to a set of administrative operations, known as *Interface-5*, defined as part of the reference model suggested by the Workflow Management Coalition [45]. We further extend the semantics of Interface-5 to account for the concurrent nature of distributed workflows. That is, concurrent management operations can *interfere* with each other and cause semantically conflicting behavior [27]. For example, suppose there are two concurrent management operations to be executed in order. One is to skip a halted task and trigger the task following the halted task. The other succeeding management operation is to resume the halted task. If the operation of resuming the halted task takes place before the skip operation, then the two operations violate the management semantics. In order to avoid this violation, the skip operation should not be interfered with. We articulate this notion of interference in the management context and devise techniques to prevent interference. The techniques include basic in-order processing of management requests and isolating a batch of management operations based on scheduling constraints.

In addition, our distributed management system is built as part of a management infrastructure consisting of agents that form an elastic cluster, such that the agents can adaptively join or leave the cluster to relieve or consolidate the management workload. Also, the agents can flexibly relocate close to the remote task that requires frequent management operations, so that overall response time of management operations is kept minimal. These features are implemented on top of a content-based publish/subscribe broker overlay [20] as an event-driven messaging substrate. The brokers in this model decouple agents as publishers and subscribers in space and time, thus showing that the publish/subscribe paradigm is well-suited as a large-scale distributed workflow management system.

The rest of the paper is organized as follows: Section 2 reviews the key management operations and explains the challenges behind executing management operations; Section 3 presents the design of the distributed architecture of the management systems and covers implementation details; Section 4 evaluates the feasibility of our approach based on key empirical results derived from our experiment; Section 5 puts our work in the context of existing workflow management systems and Section 6 concludes.

2. MANAGEMENT ACTIONS

This section reviews the primitive workflow management operations proposed by the Workflow Management Coalition specified in Interface-5 [45]. Based on this specification, we introduce *composite operations* that enable the isolated execution of concurrently running primitive operations. We also review the use of the publish/subscribe paradigm for the execution of workflows. Finally, we discuss various corner cases in the executing of management operations specific to distributed workflows.

2.1 Primitive operations

The typical primitive management operations in Interface-5 [45] are summarized in Table 2.1. The operations are applicable to particular runtime instances of an entire workflow, or to individual tasks the workflow is comprised of.

Type	Example
Workflow supervision	Pause, resume, jump, skip an instance or individual tasks. Terminate a workflow or instance. Assign or update attributes in a workflow.
User management	Establish, delete, suspend or amend privileges or roles of users or workgroups.
Audit management	Query, print, start new or delete an audit trail or event log for monitoring purposes.
Resource control	Set, unset or modify concurrency levels of an entire workflow instance or its individual tasks. Interrogate resource control data such as counts, thresholds, usage parameters.
Workflow status function	Open & close a workflow or task query with optionally set filter criteria.

Table 1: List of primitive management operations.

These primitive operations are executed based on the administrative needs. In order to understand how the primitive operations should be executed in a distributed environment, we first review how distributed workflows are processed in a publish/subscribe-based execution systems.

2.2 Distributed Workflow Execution

Our primary focus is on NIÑOS, a publish/subscribe-based distributed workflow execution system that offers concurrent and autonomous execution of tasks deployed across geographically distributed locations [26].

Figure 2 shows the distributed workflow execution architecture where *task agents* (TA) are deployed to the PADRES [20] broker network as both publishers and subscribers. A workflow is first *deployed* by having TAs exchange advertisement and subscription messages that are in the form of conjunctions over Boolean predicates or composite subscriptions. Unlike subscriptions, also referred to as *atomic subscriptions*, a composite subscription is used to correlate publications over time[1]. A composite subscription is

[1]A composite subscription is not related to a composite management operation introduced above.

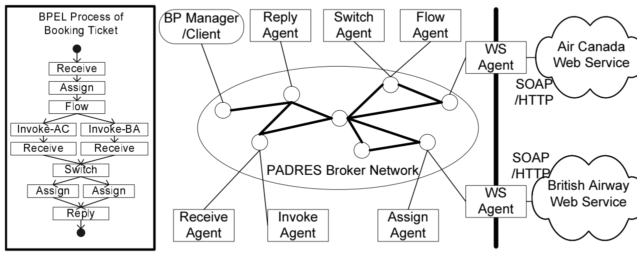

Figure 2: Distributed workflow execution architecture [26].

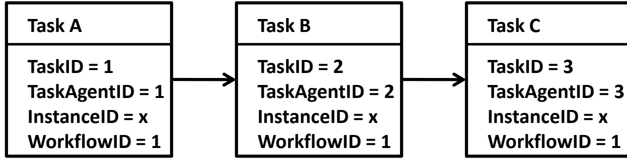

Figure 3: A sample distributed workflow.

needed to express the dependency of a successor task on two or more predecessor tasks, for example.

For example, suppose a sample sequential workflow with an identifier (ID) of "1" is given as shown in Figure 3. During the deployment phase, the TA for Task A advertises the following message.

[class, eq, TASK_STATUS],[workflow, eq, "1"], [taskName, eq, "Task A"],[IID, isPresent],[status, isPresent] [2]

The TA for Task B subscribes to the completion event (SUCCESS) of Task A by issuing the following subscription message.

[class, eq, TASK_STATUS],[workflow, eq, "1"], [taskName, eq, "Task A"],[IID, isPresent], [status, eq, "SUCCESS"].

Once the deployment is completed, the workflow is ready for execution. Workflows can be triggered from anywhere by publishing messages subscribed to by the task in the workflow. For example, the TA for Task A publishes the following message upon successful completion of Task A.

[class, TASK_STATUS], [workflow, "1"], [taskName, "Task A"], [IID, "1"], [status, "SUCCESS"].

Then, TaskAgent2, receives the publication and triggers the execution of Task B.

2.3 Management mechanisms

Assuming the coordination of workflows according to the aforementioned publish/subscribe-based coordination paradigm, we now look into the mechanisms required for executing the primitive management operations listed above. The primitive operations can be classified into three main categories: (1) Workflow modification, (2) discovery and (3) variable update. The discovery operations focus on monitoring the status of workflows, while the modification operations manipulate the workflow and its attributes at runtime. Variable update operations assist users in modifying the data associated with workflows. We select representative operations from each category and explain how they are realized in the publish/subscribe system and how they work around corner cases that arise in an event-driven and distributed environment.

[2]eq is the equal operator on string attributes. The complete description of the PADRES publish/subscribe syntax is available in `http://www.msrg.utoronto.ca/projects/padres/docs/dev_guide`.

2.3.1 Modification operation: Pause/resume

To pause a particular running instance of a workflow, issuing a message to unadvertise/unsubscribe the completion event for the instance comes to mind. But, note that, during the deployment phase, the TAs do not advertise/subscribe to the value of an instance identifier (IID) of a workflow. The TAs, instead, advertise/subscribe to all IIDs of a workflow by specifying the following predicate, [IID,isPresent]. This is to avoid the re-deployment of the tasks when a new instance is triggered.

Therefore, a two-phase execution of the pause operation can be considered. First TAs unadvertise/unsubscribe to the events produced by the workflow, W. Then, TAs re-advertise/re-subscribe to exclude the instance that should be paused by using the not equal operator (neq) in the predicate such as, [IID,neq,<instance ID>]. However, during the first phase, instances other than <instance ID> are going to be disrupted as unadvertisement/unsubscription cause the completion events to be dropped, unless the TAs are synchronized amongst themselves through some coordination mechanism. Even given that coordination would be enforced, the approach of removing and reissuing advertisement and subscription messages is still not desirable, since the unadvertisement and unsubscriptions messages have to traverse through the network, and it can be expensive to delete and insert advertisements and subscriptions at the brokers on which TAs run.

A simpler and more cost-effective approach is to apply the pause mechanism on the broker event queues, as shown in Figure 4. After the pause request is received, each TA holds any events with the ID of the instance that has to be paused on a separate wait queue. A resume operation is symmetric to the pause operation such that TAs flush the wait queue that contains the paused events upon receiving the resume request. An unintended outcome can occur due

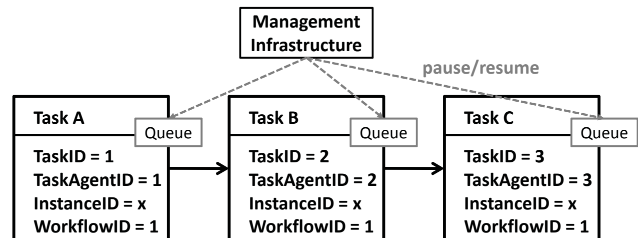

Figure 4: Execution of pause/resume operations.

to propagation delays. For example, given the sample workflow in Figure 3, assume that a pause operation of an instance at Task B. This command can fail, if the instance completes execution before the pause command actually reaches the TA running Task B. A few ways to solve this problem include: (1) Querying the status of the workflow just before pausing; (2) pausing a more distant task which has a lower chance of completing before the command reaches it. For instance, Task C can be paused instead of Task B, when Task A is currently being executed; (3) broadcasting the pause operation to all the TAs responsible for that instance or workflow and performing that operation at the earliest possible opportunity.

2.3.2 Modification operation: Skip

A running instance of a workflow, can skip over a task. For example, in Figure 5, Task B is to be skipped and then Task C is to be triggered instead upon the completion of Task A. The skip oper-

ation is performed as follows: (1) the TA for Task B halts the task of instance I by simply ignoring the completion event of instance I from the TA for Task A; (2) the TA for Task A newly advertises a completion event for instance I; (3) the TA for Task C subscribes to the event the TA for Task A publishes.

Note that the TA for Task B is still subscribed for the completion event coming from the TA for Task A. Thus, the events keep getting delivered unnecessarily to the TA for Task B, which may generate unnecessary messages. In order to prevent this, the TA for Task B, can unsubscribe to the completion events from the TA for Task A, and then re-subscribe to the completion events from the instances of the workflow W except the instance I using the not equal operation in the predicate as introduced in the previous section. This entails the risk of dropping the messages for instances other than I. To prevent the message drops, the TA for Task A must hold the completion events during the preparation of the skip operation. The overhead of the synchronization in this case is acceptable as the synchronization is applicable only to the TAs for the to-be-skipped (Task B) and skipping (Task A) tasks.

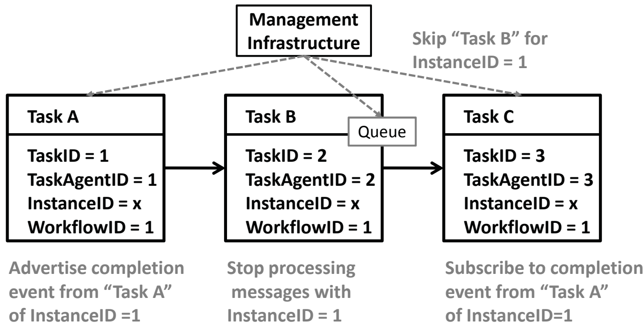

Figure 5: Execution of skip operations

2.3.3 Discovery

Each TA is responsible for the information regarding the instances it is currently executing or pausing. Information about completed instances are handled by the TA responsible for the final task of the workflow. Alternatively, a logically centralized service could be used to store and deliver all this information.

Discovery operations do not create concurrency issues with other management operations. However, there is a possibility that queries relating to concurrently executing workflow instances are ignored in the de-centralized case. Revisit the sample workflow in Figure 3, assume that Task A has completed and execution transitions to Task B for instance x. During this transition time, suppose a query for concurrently executing instances is issued to all TAs. While, execution transitions, both the TA for Task A and the TA for Task B do not include instance x in the query response. This is because, from the perspective of both TAs during the transition time, there is no concurrently executing task for the queried instance (instance x). To handle this concurrency problem, the TAs keep ownership of task instances until the succeeding TA takes ownership upon receipt of the trigger message.

2.3.4 Variable updates

This class of operations can support accessing or modifying variables and attributes associated with workflow instances and entire workflows. For example, user management operations and certain resource control operations such as set, unset, and modify of concurrency levels can be performed through variable updates. This class of operations is discussed in detail by Li *et al.* in the context of the NIÑOS architecture [26].

2.4 Composite Operations

So far, we have looked into the mechanism for the execution of primitive operations. Primitive operations are issued by multiple management clients concurrently. For example, suppose two management clients (Client 1 and Client 2) issue operations on a workflow instance (I=1), as shown in Figure 6(a). The operations can be interleaved depending on the order in which they are requested. However, this can be unsafe as the operations may semantically conflict with each other. Suppose one client issues *"pause instance - amend privilege of a user - resume instance"* operations, while another client issues *"fetch instance status - resume"* that are executed between the *"pause instance"* and *"amend privilege"* operations. In such a case the user whose privilege was supposed to be amended can continue to execute a task of the workflow instance without properly updated privileges. To prevent such interference amongst multiple clients, we introduce *composite operations* in this section.

The management client may want a sequence of primitive operations on a set of instances to be executed as a batch, such that the batch does not interfere with operations issued by other management clients. We refer to a batch of primitive operations as a *composite operation*. The *interference* is more precisely defined in Definition 1.

Definition 1. A primitive or composite management operation that is applied to a set of instances $\mathbb{I}_1$, *interferes* with another management operation that is applied to a set of instance $\mathbb{I}_2$, *if* $\mathbb{I}_1 \cap \mathbb{I}_2 \neq \phi$.

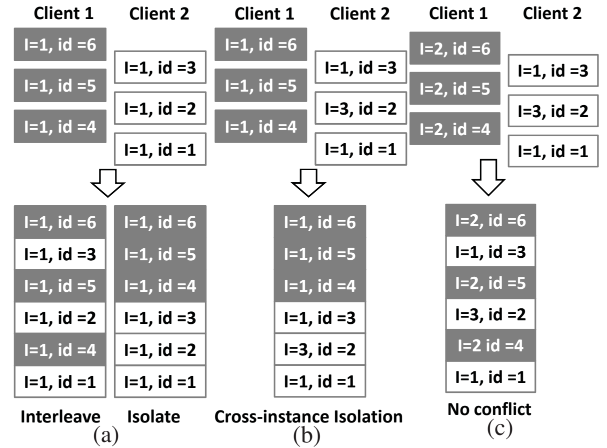

Figure 6: Example of isolation through composite operations.

Figure 6 illustrates isolation of concurrent management operations through the enforcement of composite operations. In Figure 6(a), suppose each client issues three management operations on the same workflow instance (I=1). In order to avoid that operations from the two clients interleave, the operations issued by Client 2 are blocked until all operations by Client 1 are completed. The operations by Client 1 are executed in a batch and they are not *interfered* with the operations of Client 2. Note that, as Definition 1 specifies, the composite operations can be applied across many instances as well. In such a case, if the set of instances intersects between two management operations, one of the management operations has to be blocked. For example, as shown in Figure 6(b), Client 2 is blocked, because Client 2 has two management oper-

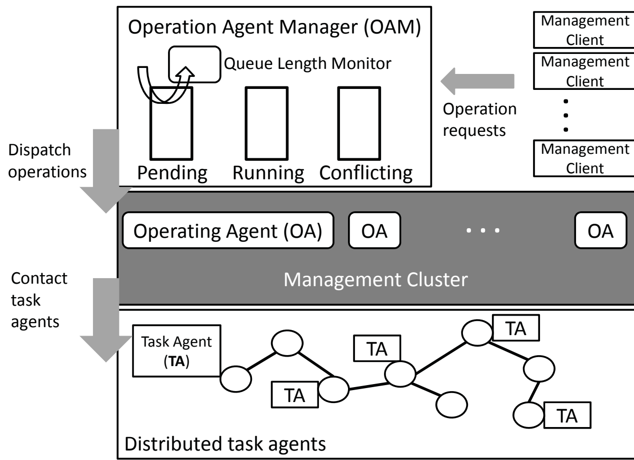

Figure 7: Architecture of the management cluster of OAs managed by an OAM.

ations (id=1 and id=3) on the workflow instance, I=1, which can intersect with the management operations of Client 1 (id=4, id=5 and id=6). However, if there is no set of instances that intersect between two management operations, then the two management operations can run fully concurrently. For instance, as shown in Figure 6(c), Client 1 and Client 2 issue management operations that do not overlap on a set of instances, therefore there is no concern for a conflict.

Composite operations trade off degraded concurrency for a conflict-free execution of management operations through isolation. A composite operation is similar in spirit to a transaction. However, at least for now, our composite operations do not support atomicity, consistency and durability, which is subject to future work.

In the following section, we discuss the distributed architecture of the management infrastructure.

3. MANAGEMENT INFRASTRUCTURE

In the previous section, we presented the mechanism of executing management operations over distributed workflows. A centralized management entity could interact with the TAs to issue management requests. However, a centralized approach may not scale and suffer from a single-point failure. Instead, we develop an event-driven and distributed management infrastructure where the management agents flexibly join, leave and move in an on-demand fashion.

3.1 Architecture

Our management infrastructure is built on top of the publish/subscribe-based workflow execution system described in Section 2.2. The management infrastructure consists of three main entities: Operation agents (OA), operation agent managers (OAM) and management clients (MC). The OA is responsible for dispatching management requests directly to the TAs (task agents). OAs form an elastic cluster managed by the OAM that distributes management requests from the MC amongst the available OAs. By default an OA cluster is provisioned for a single workflow, and the OAM keeps the mapping between the OA cluster and the associated workflows. OAs can also be clustered flexibly by the type of operation they perform. There can be multiple OAM-OA clusters in the entire system. Figure 7 shows the interactions among the different entities in more detail. The underlying communication substrate is the PADRES publish/subscribe system that allows all entities to be loosely-coupled and benefit from location transparency [20]. The detailed specification of the publish/subscribe messages that are exchanged among the entities are available in the online appendix [22].

3.2 Scheduling of management operations

We provide the mechanism of processing the management request made concurrently by the MCs.

Primitive operations: Here, we explain how the primitive operations are scheduled. First, upon receipt of the management request for a particular instance I from some MC, the OAM checks if any previously issued management operation on I is still running by checking the list of running instances (*Running*) and pending operations (*Pending*) that are kept by the OAM. If there is already a management operation running or pending for I, then the current management operation is enqueued in the *Conflicting* queue (Algorithm 1:1-2), otherwise, the management operation is assigned to an OA that is available (Algorithm 1:5).

Algorithm 1: dispatch(op)

```
1 if runningList contains op.instanceID || pendingQueue
  contains op.instanceID then
2     conflictQueue.enqueue(op);
3 else
4     if an OA is available then
5         runningList.add (op.instanceID);
6     else
7         pendingQueue.enueue(op);
```

Algorithm 2: fetchNextOp

```
1 for op ∈ conflictQueue do
2     if ¬ (pendingQueue contains op.instanceID || runningList
      contains op.instanceID) then
3         if op is first for instanceID then
4             conflictQueue.dequeue(op);
5             pendingQueue.enqueue(op);
6 opPending = pendingQueue.dequeue;
7 runningList.add(opPending.instanceID);
```

If there are no OAs available, a management operation will be put in the *Pending* queue as long as the operation does not conflict with the operation currently running or pending (Algorithm 1:7).

When, an OA becomes available the OAM first scans through the *Conflicting* queue to identify the first operation of any instance that does not conflict with the operations pending or running. The operation that does not conflict with any existing operations are enqueued in the *Pending* queue. Then, the OAM dequeues an operation from the *Pending* queue and assigns it to an available OA. The OAM fetches the earliest operation from either the *Pending* or *Conflicting* queue (Algorithm 2). This scheduling mechanism satisfies the following property.

Property 1. ***Cross-client ordering of concurrent operations on an instance***: Suppose a pair of management operations, m_0 and m_1, for a given instance I are issued by two different clients. If the operations are received by an OAM, where m_0 is received before m_1, then the OAM must block m_1 until m_0 completes its execution.

Property 1 ensures that an operation on a particular instance is mutually and exclusively executed by a single OA in order to pre-

vent out-of-order processing of concurrent operations through parallel OAs. An OA has time-outs on each operation to prevent infinite waits and consequently suffer from deadlocks. However, a high degree of concurrency can still be achieved if operations on different instances are issued concurrently.

Note that not all management operations have to be scheduled to be executed by an OA. For example, concurrent queries for workflow state, *i.e.*, discovery requests, do not have to be synchronized. Also, concurrency among variable update requests is already handled through the use of *variable agents* in NIÑOS[26]. For data stored in attached historic data stores (i.e., databases), databases themselves provide concurrency solutions which can be leveraged by a TA. Hence, an MC can contact variable agents, TAs or database service agents directly for these management operations. However, even these operations can be a part of composite operations which have to be scheduled through the OAM as explained in the following section.

If the same operation repeats itself either in the *Pending* queue or *Running* list, then only one of these operations will be executed to eliminate duplicates[3]. For example, if two management clients send commands to turn on the same device simultaneously, then only one of these commands is executed.

Composite operations: In case a batch of operations on multiple instances has to be executed without interruption, a composite operation can be used as explained in Section 2.4. Scheduling of composite operations is as follows. All primitive operations in a composite operation are issued by an MC in sequence. Each operation in the sequence is tagged with a composite operation ID. The operation is executed one at a time based on the availability of the OAs, while the other operations are put in the *Pending* queue. Once the last operation in the composite operation completes, the composite operation ID is removed from the *Running* list. Any other primitive or composite operations issued by other MCs are put in the *Conflicting* queue if they conflict with the previously submitted composite operations. Otherwise, they are put in the *Pending* queue. This scheduling mechanism for the composite operations satisfies the following property.

Property 2.*Per client and per instance set isolation:* Let a composite operation, m_0, in the sequence of primitive operations, p_0, p_1, $\ldots$, p_i, executed on a set of instances, $\mathbb{I}_0$, of a workflow, then any primitive or composite operations on $\mathbb{I}_1$ following p_0, such that $\mathbb{I}_0 \cap \mathbb{I}_1 \neq \phi$, is blocked until m_0 completes its execution. Each primitive operation in the composite operation also executes sequentially.

Property 2 ensures isolation and operation ordering of a composite operation by locking the instance set, $\mathbb{I}_0 \cup \mathbb{I}_1$. This degrades the level of concurrency.

A stronger isolation level applies for the case when the composite operation consists of a sequence of primitive operations on the set of instances across different workflows as stated in Property 3.

Property 3.*Per client and cross workflow isolation:* Let a composite operation, m_0, in the sequence of primitive operations, p_0, p_1, $\ldots$, p_i, execute on a set of different workflows, $\mathbb{W}_0$, then any primitive or composite operation on $\mathbb{W}_1$ following p_0, such that $\mathbb{W}_0 \cap \mathbb{W}_1 \neq \phi$, is blocked until m_0 completes its execution. Each primitive operation in the composite operation also executes sequentially.

[3] This only applies to modification and composite operations.

Property 3 can be easily supported by having a single OAM handle multiple workflows. However, provisioning a single OAM for all the workflows in the system limits the scalability of the management infrastructure. Thus, multiple OAMs can be assigned to a subset of the workflows, and the OAMs can coordinate themselves to enforce isolation.

3.3 Deployment and dynamic migration

The communication channels among OAM, OAs and TAs are set at deployment time by using subscriptions and advertisements. Each TA subscribes to management operation command messages from OAs, and advertises management operation command response messages to the OAs.

On the other hand, OAs subscribe to management operation command response messages and advertise management operation command messages. The *status* attribute in the corresponding subscription or advertisement has three possible values: *completed*, *paused* and *in progress*. OAs and OAM exchange subscriptions and advertisements, via the *OpControlLogic* message class to set up the management control logic [22].

One of the challenges in building a distributed management infrastructure mentioned in Section 1 is caused by network delays inherent to the distributed character of the architecture. The response time of management operations can be sensitive to the location of the OAs that interact with the TAs. To address this challenge, we deploy the OAs and the OAM in a location on the underlying publish/subscribe-based messaging systems where the average distance between the OAM and the TAs is minimal. Given n TAs, the average distance between OAM and TAs is formulated as follows:

$$\frac{\sum_i^n (d(OAM, TA_i) + d(OA_j, OAM))}{n}, \tag{3.1}$$

where d(x,y) denotes the distance between x, y measured in message substrate overlay hop counts. Alternatively, other metrics could be used. OAs can be initially co-located with the OAM or placed in adjacent locations. This is a static approach that cannot account for the dynamic management workloads that can be skewed towards particular TAs over some period. To address varying workloads over time, OAM and OAs can be dynamically migrated towards the TAs that experience heavy traffic, so that overall latency in executing the management operations can be further reduced. Dynamic migration is based on the metric in Equation 3.2 which adapts Equation 3.1 to include the workload information:

$$\frac{\sum_{i,j}^{n,m} (w \times d(TA_i, OA_j) + d(OA_j, OAM))}{n}, \tag{3.2}$$

where w is the workload of TA_i that is measured as the number of management operations that are executed. The OAM maintains this information which is made available at deployment time. At deployment time, not only OAMs but also OAs are migrated to the new location through the movement protocol developed by Hu *et al.* [24]. The static approach in Equation 3.1 assumes that the workload is uniform, thus w is 1. The transactional concerns of the message delivery during the movement are atomicity, consistency, and isolation properties which are supported through the movement protocol. Hu *et al.* empirically proved the scalability of the protocol as it yielded constant average movement latency under 5 seconds in terms of topology size and the number of clients to move. Therefore, the movement protocol can be used to enforce service level agreement (SLA) [32]. In Section 4, we show the benefit of the dynamic migration in the management infrastructure.

3.4 Elastic management cluster

We assume that a pool of resources in form of a cloud, for example, is available throughout the enterprise network, which allows our management infrastructure to flexibly allocate OAs to handle high loads, or consolidate load during periods of low utilization of the existing OAs. A feedback loop is built to the OAM that monitors the queue length and determines the need for changing the management capacity through allocation or consolidation. We reassert the need of the publish/subscribe substrate, since dynamically joining OAs can seamlessly integrate themselves to the management cluster by simply issuing the set of advertisements and subscriptions to talk to the OAM whose location is transparent to the OAs.

4. EVALUATION

This section presents the experimental evaluation of our management infrastructure and the management mechanisms itself. The management infrastructure has been implemented on top of the open source PADRES[4] content-based publish/subscribe system. The management infrastructure was evaluated on a Dell PowerEdge 2900 with two quad-core 3.00 GHz processors and 16GB of RAM.

An acyclic and connected publish/subscribe broker overlay was constructed to enable asynchronous communication among the management entities. Five TAs were deployed to the overlay to process sequential workflow instances. Up to 12 OAs were utilized in a dynamically growing and shrinking management cluster managed by a single OAM. Management operations were issued at varying frequencies as high as 10 messages per second.

Given this evaluation setting, we conducted the following experiments: we profiled the performance of modification and discovery operations to assess the overhead of the management cluster; we analyzed the effect of an elastic management cluster; we observed the performance trade-off between isolation and concurrency by varying the number of conflicts in the management workload; and finally we demonstrated the effects of dynamic migration and an elastic management cluster.

4.1 Execution latency of primitive operations

This experiment evaluates the latency of executing two types of primitive operations: Workflow modification and discovery. The average, minimum and maximum execution latencies were measured by an MC that first initiates the management operations. The number of OAs was fixed at 5; all of which were managed by a single OAM. As shown in Figure 8, the management cluster was deployed to a overlay of PADRES brokers (n1, n2 and n3). The management cluster was placed on the location where the distance (Formula 3.1) between the OAM and each of the 5 TAs was constant, so that the effect of the network delay is controlled. Management operations were issued at a rate of at most 2 messages per second to emulate the actions of a human administrator. As shown in Figure 9, on average, a workflow modification operation such as pause, resume and skip took approximately 60 ms, while a discovery operation took approximately 10 ms. The 50 ms execution latency gap between the two different types of operations represents the overhead of the management cluster, since modification operations are processed through a management cluster of an OAM and OAs, while discovery operations directly communicate with the TAs as explained in Section 2. The execution latency for both categories of management operations remained constant during the experiment. This is because the management cluster was not saturated due to sufficient capacity (5 OAs). The elasticity of our management cluster to handle overloads is demonstrated in the following experiment.

[4]Available for download at `http://padres.msrg.org`.

We also observe that the execution latency for any type of management operation can increase, if the TAs are overloaded. For example, if a TA is involved in a workflow that goes through frequent iterations, then, the TA may not promptly respond to the request from the management cluster or MCs. As optimization, management operations can be processed at a higher priority at the publish/subscribe brokers, so that more critical management operations can be delivered to the TAs more quickly.

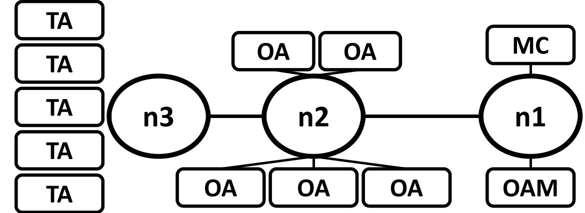

Figure 8: The topology for evaluating execution latency of primitive operations.

4.2 Effect of elastic management cluster

The effectiveness of a management cluster is measured by two parameters: The execution latency and the overhead of running the cluster. A more effective approach has both lower latency and lower overhead when faced with varying workloads. Execution latency is measured for different frequencies of incoming requests by using the management client (MC). The amount of resources used is measured by the size of the cluster, more specifically, by the number of OAs present in the management cluster. To evaluate both parameters, we compared two different clusters: one with an elastic cluster and the other with a fixed number (4) of OAs. The other settings were the same as in Section 4.1. As shown in Figure 10, the MC constantly tried to saturate the OAM with modification requests at increasing frequencies. This in turn increased the queue lengths within the OAM. So, the OAM grew the management cluster (by adding an OA) and kept the response time nearly constant. For any given management request frequency, the elastic management cluster had a better average execution latency of 45 ms when compared to 60 ms of the static management cluster (as observed in Figure 9). The MC sent a batch of 20 requests for every frequency. The duration represents the time interval in which all the requests of a given batch were completed.

4.3 Trade-off between isolation and concurrency

To evaluate the trade-off between isolation and concurrency, a key scheduling constraint, the number of conflicts to resolve, was modified while keeping the number of management requests per second constant. Given the same setting as the experiment in Section 4.1, the MC increased concurrency by issuing modification requests to more workflow instances so that the chance of interference, as defined in Definition 1, was reduced.

Figure 11 shows a sharp drop in execution latency at a concurrency level of 2. After which there is a gradual decrease from 58 ms at concurrency level 2 to 50 ms at concurrency level 10. The benefit of increased concurrency is reduced because the parallelism is bounded by the fixed number of OAs in the cluster, and each OA can process only one management request at a time. To let the management infrastructure scale and exhibit higher speedup, we have

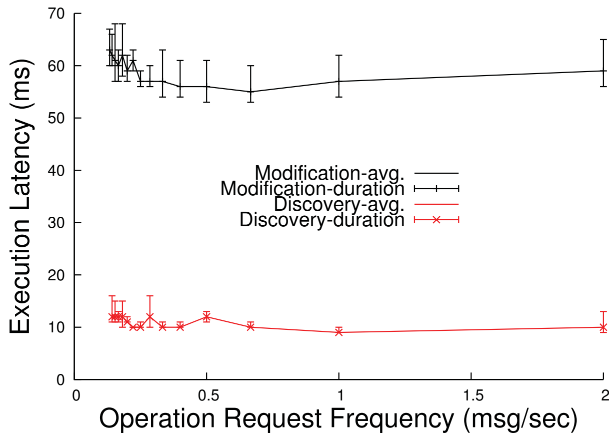

Figure 9: Execution latency trend of different primitive operations.

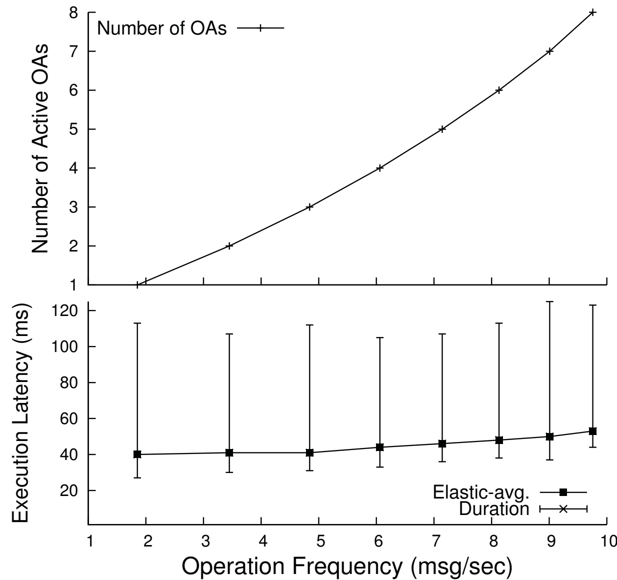

Figure 10: The effect of an elastic management cluster.

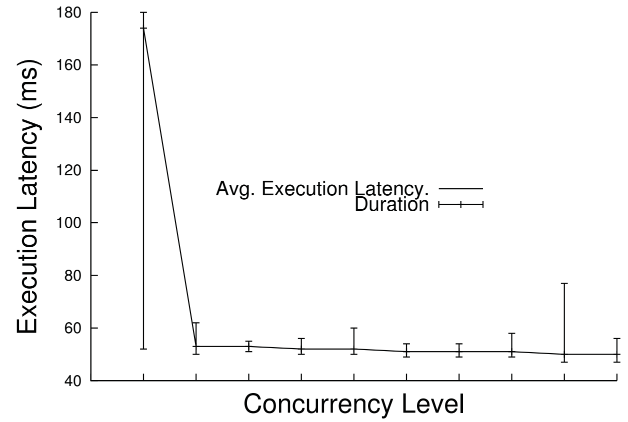

Figure 11: The effect of isolation on performance.

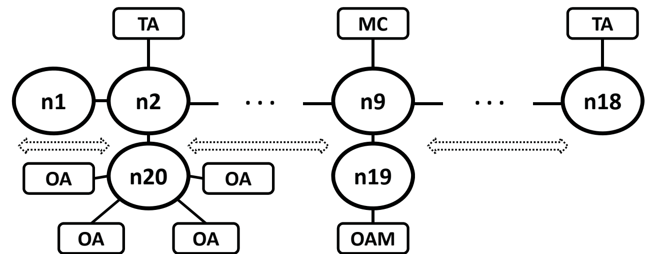

Figure 12: The dynamic migration experiment setting.

already demonstrated the growing and shrinking of the management cluster in Section 4.2.

4.4 Effect of dynamic migration

To highlight the benefit of dynamic migration, we moved the management entities along a flat publish/subscribe broker overlay to systematically vary the average distance from the OAM to TAs, as shown in Figure 12. OAs and OAM are connected to an edge broker (e.g., n19, n20). The locations of MC and TAs are fixed. When the load is uniformly random, (i.e., $w = 1$ in Equation 3.2), n9 would be the optimal location for all OAs and the OAM.

Two comparison runs for randomized requests with varying frequency were performed and their execution latencies were measured. One of the two runs had a random placement of the management cluster. This was emulated by using uniformly randomized delays (between 20 ms to 70 ms) in each OA and OAM. This was to emulate their offset from the optimal deployment position. In the other run, the cluster had dynamic migration and the entire cluster was attached to broker n9, so there were no additional emulated delays.

In Figure 13, when dynamic migration feature was enabled the execution latency was approximately 10 times lower than the execution latency with random deployment.

5. RELATED WORK

In this section, we put our work in the context of workflow management systems, distributed workflow processing systems, and management system from other domains that adopt publish/subscribe-style processing.

Workflow management systems: Notable commercial workflow management systems are IBM WebSphere MQ Workflow [2], Oracle Business Process Management [5], Sage ERP X3 [6], Symantec Workflow [9] and SAP Netweaver [7]. These products offer process change functions and real-time monitoring/analysis of process instances. The management is done in a centralized fashion and does not support distributed management and execution of workflows.

Event-driven monitoring approaches for the distributed Web services are found in [47, 28, 25, 38]. These approaches express policies and requirements in the event calculus, so that inconsistent events can be detected. For assessing performance interference, Taiani *et al.* provide a black-box approach for profiling concurrent and composite applications on grid [41]. These systems are focused mostly on discovery and monitoring features. Our approach goes beyond supporting discovery and monitoring operations; we aim at supporting the full pallet of workflow execution management, especially targeting distributed workflows.

Several existing management systems focus on ensuring isolation and consistency for transactional Web services in a workflow.

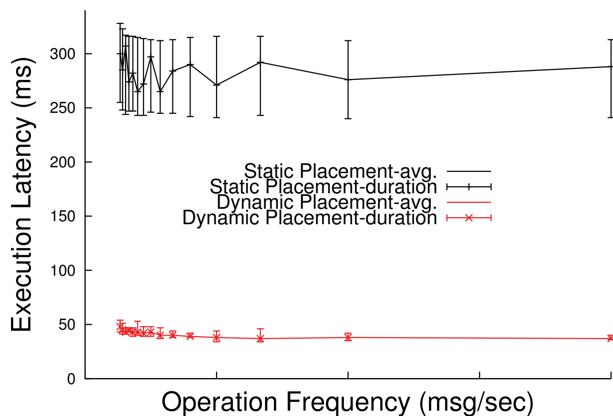

Figure 13: The effect of dynamic migration.

Choi *et al.* developed a management system that detects and fixes inconsistent states in loosely coupled Web services [17]. Paul *et al.* [35] and Puustjarvi *et al.* [37] focus on ensuring isolation properties in Web service transactions while maintaining acceptable levels of service and a high degree of concurrency. Alrifai *et al.* [13] introduce an extension to the Web service transaction protocol to ensure consistency of data when independent business transactions access data concurrently under relaxed transactional properties. These approaches focus on the reliable execution of transactional Web services, which is an orthogonal concern to the problem we address. In this paper, we focus on enabling different levels of isolation for executing concurrent management operations.

Distributed workflow processing systems: Distributed workflow processing has been studied to address scalability, fault resilience, and enterprise-wide workflow management [12, 46, 31]. Alonso *et al.* present a detailed design of a distributed workflow management system [12]. Muth *et al.* [31] describe a behavior preserving transformation of a centralized activity chart, representing a workflow, into an equivalent partitioned one. The transformation is realized in the MENTOR system [46]. The objective of this work is to enable the parallel execution of the partitioned flow, while minimizing synchronization messages, and to analytically prove certain properties of the partitioned flow [31]. Another parallelization approach is based on control flow and data flow analysis that are used to parallelize the business process so that the highest possible concurrency can be achieved [33]. Casati *et al.* present an approach to integrate an existing business processes into a larger workflow [16]. They defined event points in business processes where events can be received or sent using a centralized publish/subscribe model. The interaction of existing business processes is synchronized by event communication. These approaches focus on the processing a workflow according to a specification in a distributed environment; they often runtime re-configuration and control capabilities.

Li *et al.* [26] propose a workflow processing approach operating in a distributed environment, which partially addresses runtime management. In [26], activities in a business process are decoupled and are executed by activity agents, which are publish/subscribe clients, and the communication between agents is performed in a content-based publish/subscribe broker overlay network. Basic workflow supervisory functions such as pause and resume are supported, but at a coarse-grained level, *i.e.*, pausing or resuming entire activity agents. In this work, we extended the management capability in [26] to support a much more finer-grained level of control over individual tasks and workflow instances, also operating across different workflows.

Publish/Subscribe-based management systems: There is a proven track record for the successful adoption of publish/subscribe systems in supporting distributed applications with monitoring and control capabilities. Mukherjee *et al.*, for example, used a publish/subscribe system to monitor and detect intrusions [30]. Tock *et al*. built a stock-market monitoring system [42]. Fawcett *et al*. employed publish/subscribe to detect changes to a business activity [19]. For the distributed orchestration of interoperable electric devices, IEC 61850 specifies protocols such as the Generic Substation Event (GSE), which are used for real-time transfer of event data [3]. GSE is a publish/subscribe-based communication platform adopted by companies such as ABB and Siemens [1, 8]. The event-driven monitoring and control by means of a publish/subscribe system in the above listed approaches enables the asynchronous and scalable management of applications, properties directly founded in the publish/subscribe-based realizations.

6. CONCLUSIONS

We presented and evaluated an approach for the fine-grained management of event-driven and distributed workflows. Our approach employs a publish/subscribe messaging substrate that supports location transparency and asynchronous communication among all management entities and task agents. The management entities can flexibly join or leave the management infrastructure depending on the management workload. In addition, management entities can migrate to locations to reduce overall latency of management operations.

The experimental evaluation shows that our distributed and event-driven approach scales and maintains constant execution latency for varying management workloads. Also, the average execution latency was reduced by 25% using our elastic management cluster approach. The execution latency with dynamic migration enabled was substantially lower than without migration.

7. ACKNOWLEDGEMENT

This research was supported by IBM's Center for Advanced Studies and Bell Canada's Bell University Laboratories R&D program, and the Natural Sciences and Engineering Research Council of Canada. We would also like to thank our colleagues including Vinod Muthusamy and Chunyang Ye for providing valuable feedback on this manuscript.

8. REFERENCES

[1] ABB control systems. http://www.abb.com/product/us/9AAC910002.aspx.

[2] IBM WebSphere MQ workflow. http://www-01.ibm.com/software/integration/wmqwf/features/.

[3] IEC 61850. http://www.iec.ch/.

[4] Omicron GOOSE configuration. http://www.omicron.at/en/products/pro/communication-protocols/iec-61850/goose-configuration/.

[5] Oracle business process management. http://www.oracle.com/us/technologies/bpm/index.html.

[6] Sage ERP X3. http://www.sageerpx3.com/.

[7] SAP Netweaver business process management. http://www.sap.com/platform/netweaver/index.epx.

[8] SPPA-E3000 operational improvement and integration. http://www.energy.siemens.com/fi/en/automation/power-generation/operational-improvement-integration.htm.

[9] Symantec Workflow. http://www.symantec.com/connect/workflow.

[10] W. M. P. d. Aalst and T. Basten. Inheritance of workflows: an approach to tackling problems related to change. *Theoretical Computer Science*, 270(1-2):125–203, 2002.

[11] W. M. P. v. d. Aalst, M. Dumas, C. Ouyang, A. Rozinat, and E. Verbeek. Conformance checking of service behavior. *ACM ToIT*, 8(3):1–30, 2008.

[12] G. Alonso, D. Agrawal, A. E. Abbadi, C. Mohan, R. Gunthor, and M. Kamath. Exotica/FMQM: A persistent message-based architecture for distributed workflow management. In *IFIP*, 1995.

[13] M. Alrifai, P. Dolog, and W. Nejdl. Transactions concurrency control in web service environment. In *ECWS*, 2006.

[14] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of msc graphs. *Theoretical Computer Science*, 331(1):97–114, 2005.

[15] M. Broy, I. H. Krüger, and M. Meisinger. A formal model of services. *ACM ToSM*, 16(1):5, 2007.

[16] F. Casati and A. Discenza. Modeling and managing interactions among business processes. *Journal of Systems Integration*, 10(2):145–168, Apr. 2001.

[17] S. Choi, H. Kim, H. Jang, J. Kim, S. M. Kim, J. Song, and Y.-J. Lee. A framework for ensuring consistency of web services transactions. *Information and Software Technology*, 50:684–696, 2008.

[18] G. Decker, A. Barros, F. M. Kraft, and N. Lohmann. Non-desynchronizable service choreographies. In *ICSOC*, 2008.

[19] T. Fawcett et al. Activity monitoring: Noticing interesting changes in behavior. In *SIGKDD*, 1999.

[20] E. Fidler et al. Distributed publish/subscribe for workflow management. In *ICFI*, 2005.

[21] X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE ToSE*, 31(12):1042–1055, 2005.

[22] S. Ganesan, Y. Yoon, and H.-A. Jacobsen. Pub/Sub implementation of the management infrastructure protocols. Technical report, March 2011.

[23] H. Giese, S. Burmester, W. Schäfer, and O. Oberschelp. Modular design and verification of component-based mechatronic systems with online-reconfiguration. In *FSE*, 2004.

[24] S. Hu, V. Muthusamy, G. Li, and H.-A. Jacobsen. Transactional mobility in distributed content-based publish/subscribe systems. In *ICDCS*, 2009.

[25] S. Kulvatunyou, J. Durand, J. Woo, and H. BenMalek. Governing web services using event-driven monitoring. In *IEEE EDOC*, 2007.

[26] G. Li, V. Muthusamy, and H.-A. Jacobsen. A distributed service-oriented architecture for business process execution. *ACM TWeb*, 4(1):1–33, 2010.

[27] L. T. Ly, S. Rinderle, and P. Dadam. Integration and verification of semantic constraints in adaptive process management systems. *Data Knowl. Eng.*, 64(1):3–23, 2008.

[28] K. Mahbub and G. Spanoudakis. Run-time monitoring of requirements for systems composed of web-services: Initial implementation and evaluation experience. In *ICWS*, 2005.

[29] M. Montali, M. Pesic, W. M. P. v. d. Aalst, F. Chesani, P. Mello, and S. Storari. Declarative specification and verification of service choreographiess. *ACM TWeb*, 4(1):1–62, 2010.

[30] B. Mukherjee et al. Network intrusion detection. *IEEE Network*, 1994.

[31] P. Muth, D. Wodtke, J. Weisenfels, A. K. Dittrich, and G. Weikum. From centralized workflow specification to distributed workflow execution. *Journal of Intelligent Information Systems*, 10(2):159–184, 1998.

[32] V. Muthusamy, H.-A. Jacobsen, T. Chau, A. Chan, and P. Coulthard. SLA-driven business process management in soa. In *CASCON*, 2009.

[33] M. G. Nanda, S. Chandra, and V. Sarkar. Decentralizing execution of composite web services. In *OOPSLA*, 2004.

[34] S. Overbeek, B. Klievink, and M. Janssen. A flexible, event-driven, service-oriented architecture for orchestrating service delivery. *IEEE Intelligent Systems*, 24(5):31–41, 2009.

[35] D. Paul, F. Henskens, and M. Hannaford. Isolation and web services transactions. *ICPDCAT*, 2007.

[36] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[37] J. Puustjarvi. Concurrency control in web service orchestration. In *CIT*, 2008.

[38] M. Rouached, O. Perrin, and C. Godart. A contract-based approach for monitoring collaborative web services using commitments in the event calculus. In *WISE*. 2005.

[39] G. Salaün and T. Bultan. Realizability of choreographies using process algebra encodings. In *IFM*, 2009.

[40] S. Tai, T. A. Mikalsen, and I. Rouvellou. Using message-oriented middleware for reliable web services messaging. In *WES*, 2003.

[41] F. Taiani, M. Hiltunen, and R. Schlichting. The impact of web service integration on grid performance. In *HPDC*, 2005.

[42] Y. Tock et al. Hierarchical clustering of message flows in a multicast data dissemination system. In *PDCS*, 2005.

[43] W3C. Web services choreography description language version 1.0, 2005. http://www.w3.org/TR/ws-cdl-10.

[44] W3C. Web services coordination, 2007. http://docs.oasis-open.org/ws-tx/wscoor/.

[45] WFMC. The workflow reference model, 1995. http://www.wfmc.org/reference-model.html.

[46] D. Wodtke, J. Weißenfels, G. Weikum, and A. K. Dittrich. The mentor project: Steps toward enterprise-wide workflow management. In *ICDE*, 1996.

[47] Z. Wu, Y. Liu, and L. Wang. Dynamic policy conflict analysis in operational intensive trust services for cross-domain federations. *ICIAS*, 2009.

[48] Y. Yoon, C. Ye, and H.-A. Jacobsen. A distributed framework for reliable and efficient service choreographies. In *WWW*, 2011.

[49] S. Zaplata, K. Hamann, K. Kottke, and W. Lamersdorf. Flexible execution of distributed business processes based on process instance migration. *Journal of Systems Integration (JSI)*, 1(3):3–16, 7 2010.