Dynamic Workflow Management and Monitoring Using DDS

Pan Pan* Abhishek Dubey* Luciano Piccoli†

*Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN
†Fermi National Accelerator Laboratory, Batavia, IL, USA 60510

Abstract

Large scientific computing data-centers require a distributed dependability subsystem that can provide fault isolation and recovery and is capable of learning and predicting failures to improve the reliability of scientific workflows. This paper extends our previous work on the autonomic scientific workflow management systems by presenting a hierarchical dynamic workflow management system that tracks the state of job execution using timed state machines. Workflow monitoring is achieved using a reliable distributed monitoring framework, which employs publish-subscribe middleware built upon OMG Data Distribution Service standard[1]. Failure recovery is achieved by stopping and restarting the failed portions of workflow directed acyclic graph.

1. Introduction

Advances in super computers and storage technology coupled with the advent of commodity cluster computing and grid computing has enabled scientific research in areas such as biology, disaster simulation, and physics among others to become more accessible to researchers. While use of multiple computation nodes enables more complex problems to be solved, it also increases the administrative burden. This paper details our continuing efforts to build a distributed, yet autonomic workflow management system that can provide reliability, availability and performability. Reliability is the property that measures the statistical ratio of number of jobs that finished correctly. Availability is the statistical measure that informs if the resources required to meet the current workload were present or not. Performability measures the number of jobs that finished within the specified Quality of Service parameters (QoS) such as response time.

In a typical use case, workflow management systems are required to convert the abstract specification provided by the scientists (it is abstract because it does not include details of the resource allocation) to an executable specification. It then executes the parallel sub-steps, also referred to as participants [of the workflow] that are part of the workflow. During the execution, it must monitor the progress of participants and monitor the hardware health. It is also required to provide data-provenance to enable retrieval of intermediate results and failure-recovery (an important property for a self-managing and self-protecting system). Many existing scientific workflow management frameworks either do not provide workflow monitoring, or they provide workflow monitoring as an adjunct service. It is hard to maintain quality of service without considering monitoring as an integrated component of the workflow system.

The challenge in building such an integrated framework is to balance performance with fault-tolerance. We can achieve fault-tolerance with large-scale intensive monitoring, however, that affects the performance of the participants. Moreover, we need to avoid single point of failures, which are present in a centrally managed system. Such bottlenecks are also present in a hierarchical system with a single predefined root. To mitigate these challenges we use Data-Distribution Service for Real-Time Systems(DDS) [16]. Locally, all components of our framework are created and scheduled using a new hard real-time component framework built over a Linux implementation of ARINC 653 [7]. Description of the component framework is out of scope of this paper. But interested readers are encouraged to read [3].

The outline of this paper is as follows: We start with a discussion of our previous work and specific contributions of this paper. Then, we discuss related work. Section 3 describes the workflow management system in detail. In section 4, we describe reliable distributed monitoring, which involves DDS. In section 4.1, we show how workflow specific monitoring is performed. We then discuss the integration of managing framework and monitoring framework with the use of an example. In conclusion, we present an online web monitoring application as an extension to the monitoring framework.

1.1. Previous Work

In [13], an early design and implementation of a scientific workflow execution framework that integrates run-time verification and monitoring was proposed. It included provisions for data provenance, fault tolerance and pre-configured hierarchical online monitoring. The previous version of our monitoring framework was discussed in [4]. It was a statically configured (i.e. hard wired by design based on the computing cluster racks) hierarchical framework that provided periodic monitoring of vital health parameters and autonomic fault

[1] http://www.omg.org/technology/documents/dds_spec_catalog.htm

978-0-7695-4004-7/10 $26.00 © 2010 IEEE
DOI 10.1109/EASe.2010.12

mitigation.

1.2. Contributions of this Paper

In this paper, we extend our hierarchical framework for managing scientific workflows, and integrate it with a reliable and reconfigurable monitoring framework based on DDS. The advantages of this choice will be discussed later in the paper. We also formalize and extend the workflow management system as a dynamic collaboration of several actors, workflow manager, workflow instance manager and participant managers that tracks the state of job execution using timed state machines. These observers are instances of reflex engines, which map incoming events to pre-defined responses [13].

We will later describe that all three components share similar abstract behavior and differ only in the way they implement the running state. We believe that the usage of formal semantics will allow us to formally verify the system design in future.

2. Related Work

Autonomic Computing: The last decade since 2001 has seen a great deal of research activity in the field of autonomic computing. Autonomic computing [10, 15] is inspired from the idea of self-management in autonomous nervous system of biological systems that governs heart rate and body temperature involuntarily, freeing the brain to make other high-level decisions. According to the autonomic computing vision, computing systems should manage themselves based on high-level objectives set by administrators. An autonomic computing system usually consists of several autonomic elements. Each of them possess the capability to monitor their own managed components using sensors, and then use the historical knowledge to analyze the current and future course of actions, which can involve analysis and planning stages.

Understandably, autonomic computing, by the virtue of automating the decision procedure, requires learning and depends on historical information. For this purpose, it involves research in prediction algorithms, machine learning and feedback control. Our current work is built upon the foundation of autonomic computing research.

Workflow Management Systems: Early workflow management comes from business enterprises for partially or totally replacing human work in manufacturing and the office. In [5], original work in business workflow management is described, including workflow related concepts, modeling, implementation and automation.

Scientific workflows are specially designed for complex scientific computation, which has requirements such as process reproducibility, provenance tracking, workflow description and system level workflow management [6]. Pegasus [1] and Kepler [11] are two well-known workflow systems. Kepler provides graphical user interface (GUI) which enables scientists to visually design and execute scientific workflows. It uses an actor-oriented model as the workflow engine where common tasks can be encapsulated as reusable components.

Pegasus is another scientific workflow system with the ability of mapping resource-independent abstract workflows into distributed executable workflows. Both Kelper and Pegasus use the Grid as the main source for job processing. The Grid consists of a set of heterogeneous and geographically distributed hosts, where jobs are replicated on extra hosts to achieve reliability.

Unlike Kelper and Pegasus, Lattice QCD computations[2] (LQCD) make use of dedicated clusters at Fermi lab in Batavia, IL, USA. To better exploit capabilities of high-speed, low-latency networks, LQCD computations employ tightly-coupled parallel processing and increase productivity by enabling reliable operation without use of redundancy. Though some light weight monitoring tools for scientific workflows have been developed [14], close workflow monitoring for clusters has been lacking. This work allows LQCD computations to get the runtime performance analysis and workflow status using a reliable, distributed monitoring framework.

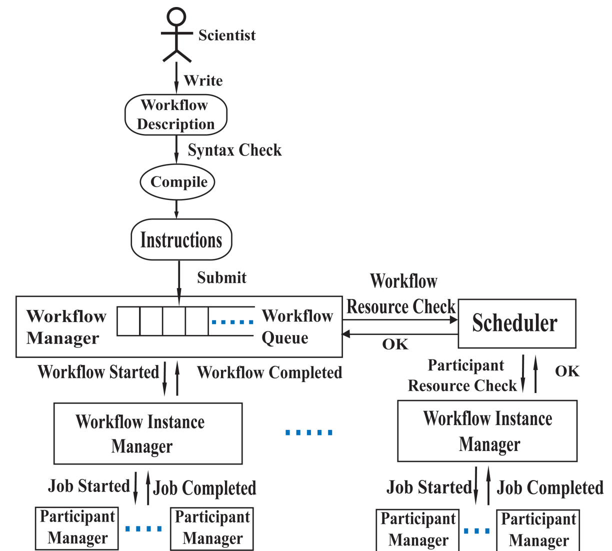

Figure 1. Workflow Management Framework

3. Workflow Management Framework

Instead of using a flat network topology, we use a hierarchical framework for managing scientific workflows. A hierarchical framework benefits from the low resource cost and high reliability. In a hierarchical network topology, each node focuses on its own responsibility, which reduces potential resource redundancy. A manager in such a design is responsible for a limited zone and hence can provide better trade-off between performance and fault-tolerance as it

[2] http://www.usqcd.org/fnal/

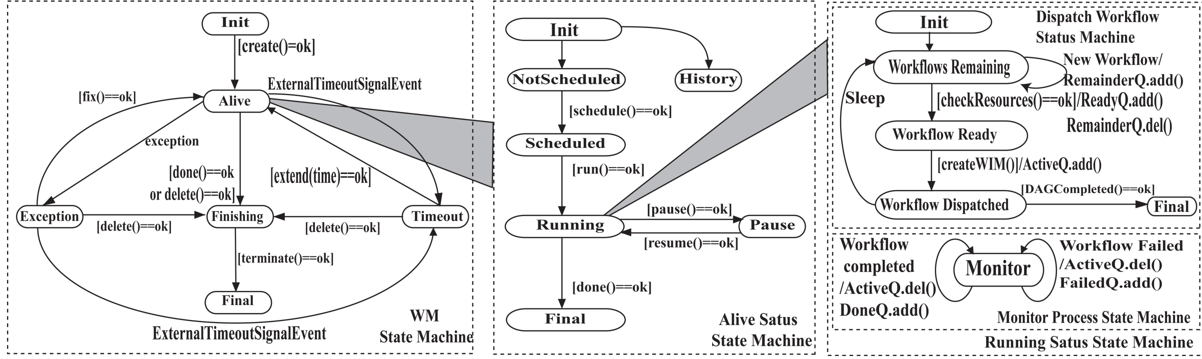

Figure 2. Workflow Manager State Machine

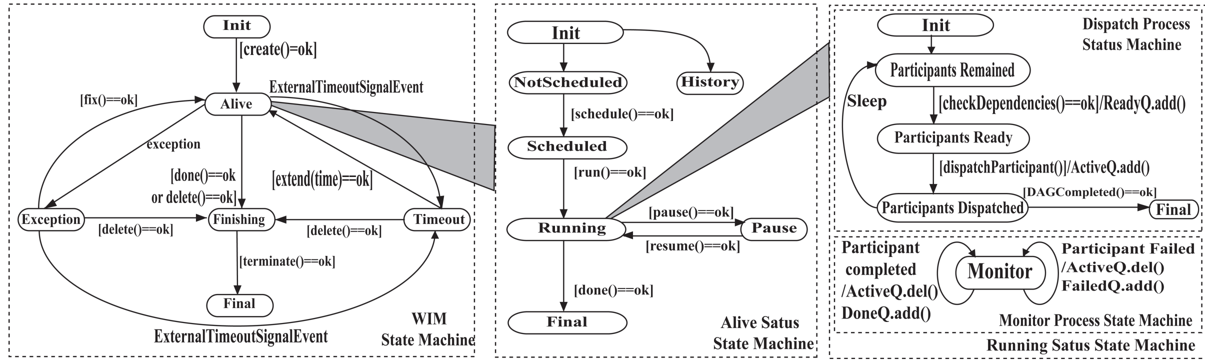

Figure 3. Workflow Instance Manager State Machine

is relatively easier to identify a fault in a limited area. Figure 1 shows our workflow management framework. The next sub-sections details all the actors in this sequence diagram.

Workflow Manager(WM) and Scheduler: To submit a workflow, a scientist writes a workflow description, which are then compiled and submitted to the workflow manager as a workflow. We require that all workflows to be directed acyclic graphs. The workflow manager(WM) is a singleton and resides on the central host in our workflow management framework. We are currently considering the idea of deploying the workflow manager over a virtual machine. That way, we can use the live migration capabilities in order to ensure that it does not become a single point of failure. We rely on existing cluster schedulers such as PBS/Maui[3]. The scheduler is responsible for scheduling workflow with required resources.

When a workflow request from the workflow manager comes, the scheduler checks the resource availability. If required resources for the workflow are available, the scheduler notifies the workflow manager that the workflow can be run. The scheduler also tells the workflow manager the potential resources for execution (the resources for actual execution may change). Thereafter, the workflow manager creates a workflow instance manager to run the workflow. Notice that a workflow instance manager is responsible for a single instance of a workflow. Figure 2 describes the state machine of a workflow manager. The WM state machine is the top level state machine. It shows that the workflow manager is considered to be alive when it is created. Once created, it stays in the 'Alive state' until a timeout, an exception or the 'done event' occurs.

Alive state is a composition of seven sub-states. If the manager is running for the first time, it will enter the 'NotScheduled' state from init state directly. If it is scheduled successfully, the state machine enters scheduled state. Then the workflow manager starts running and enters the 'Running' state. The 'Pause' state and 'History' junction are used for pausing workflow instance manager, and resuming it later. The main activity happens in the Running state, which is a sub state of Alive state. In this state, a workflow manager maintains five queues: (a) **Remainder Queue**: workflows waiting for resources, (b)**Ready Queue**: Workflows that have resources (c) **Active Queue**: currently running workflows, (d) **Done Queue**: Workflows that finished correctly. (e) **Failed Queue**: Workflows that failed with an error.

Workflow Instance Manager(WIM): A workflow instance manager(WIM) controls an execution instance of a workflow[4]. It is responsible for the following tasks (a) Cre-

[3]http://www.clusterresources.com/pages/products/torque-resourcemanager.php

[4]The same workflow can be executed multiple times.

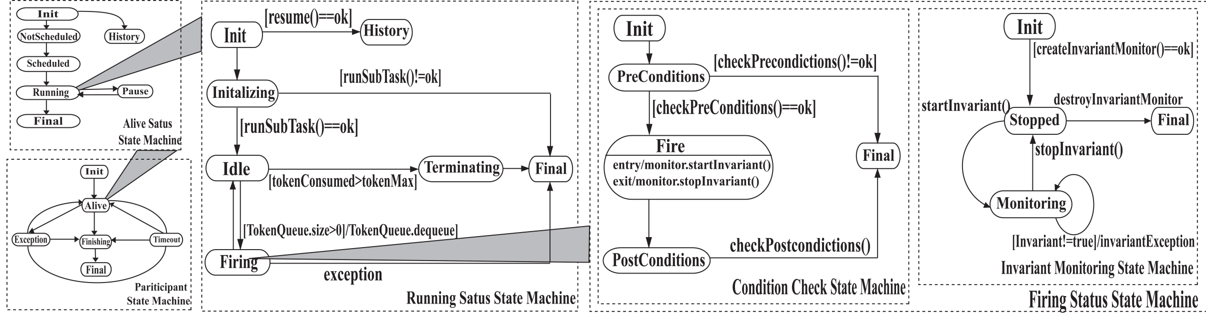

Figure 4. Participant Manager State Machine

ation and starting of participant managers (described later): It checks specified pre-conditions before the launch of a participant and checks the post-condition when participant manager finishes; (b) Monitor participant manager for progress; (c) Report any failure to workflow manager; (d) Record data provenance and return the results to workflow manager.

Figure 3 depicts the state machine of a workflow instance manager. When a workflow instance manager is created, WIM enters the 'Alive' state. Notice that the top-level status machine and the 'Alive' state machine are similar to the workflow manager. The difference lies only in the implementation of the running state. The 'Running' state is a composition of two parallel threads: Dispatch process state machine and Monitor process state machine in WIM.

A workflow is a Direct Acyclic Graph(DAG), in which vertices represent participants and directed edges represent the data and control dependencies between participants [5]. Five queues are used for facilitating participant management:

1. **Remainder Queue**: participants waiting for dependencies to complete,
2. **Ready Queue**: participants that have resources,
3. **Active Queue**: currently running participants,
4. **Done Queue**: participants that finished correctly,
5. **Failed Queue**: participants that failed with an error. This is similar to the queues used by the workflow manager.

At the beginning, all participants in the workflow are in *Remainder Queue*. WIM checks participants' dependencies, and inserts it into the *Ready Queue* if the dependency requirements are met. When resources are available, participants are dispatched for execution and inserted into the *Active Queue*. WIM sleeps for a while and then reenters 'participants remained' state. WIM has a separate thread for monitoring progress. This thread listens to events from participant manager. Two types of events are monitored: participant completed event and participant failed event. If participants finish successfully, they are inserted into *Done Queue*. If participants fail while being executing, they are added to *Failed Queue*. Upon mitigation of the failure condition, they can be reinserted into Remainder Queue again. A workflow instance is considered to be finished when all participants are in *Done Queue*.

[5]A participant is a computation job specified in the workflow.

If the workflow is completed successfully, or if the workflow is deleted by workflow manager, WIM enters the finishing state, in which the workflow instance manager releases all resources. Timeout is an external signal coming from the workflow manager. Timeouts are asynchronous to the state machine.

Participant Manager: A Participant Manager runs on the job head node. The job head node is chosen arbitrarily from the list of nodes allocated for a participant job. Note that we do not allow one node to belong to more than one participant simultaneously. The participant manager is responsible for distributing binaries across the nodes or running several binaries(e.g. behaving as an independent Workflow Instance Manager). A participant manager monitors the parallel job and returns the result to the workflow instance manager.

The state machine of participant manager is shown in figure 4. It is the same as that of workflow manager and workflow instance manager. Again, the difference lies in the implementation of running state machine. The running state implements a Kahn Process Network [9]. After initializing, the participant manager runs sub tasks and enters idle state. If there are input tokens to be processed, the participant manager will enter firing state to consume tokens. In this state it executes all the parallel instances of MPI jobs. If any exceptions happen, or a sub task does not run successfully, or enough tokens have been consumed, the participant manager enters the 'final state'. The firing state contains two state machines: 'Condition check state machine' and an 'Invariant monitoring state machine'. In the condition check state machine, preconditions are checked first. If preconditions are satisfied, fire state will be entered, and 'monitor.startInvariant()' function is called. When the task is finished, 'monitor.stopInvariant()' function is called, and the participant manager enters 'PostConditions' state. In this state post conditions are evaluated and final state is entered. These pre-, post- and invariant checks for each firing cycle of the participant Kahn Process are based on the three tuple

program definition specified by Hoare in [8].

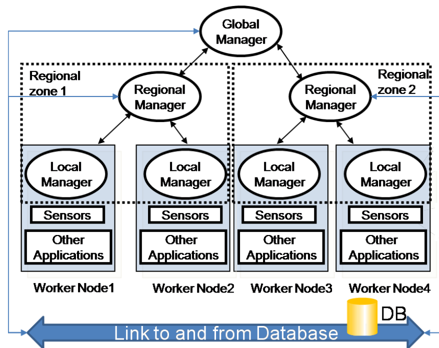

Figure 5. The layout of management entities involving global (G), regional (R) and local (L) managers.

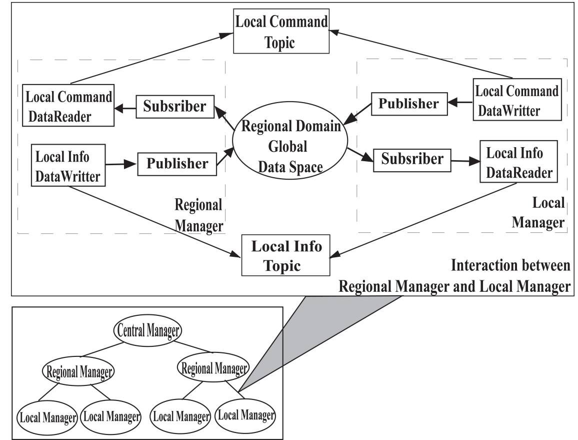

Figure 6. Interaction between Regional Manager and a Local Manager

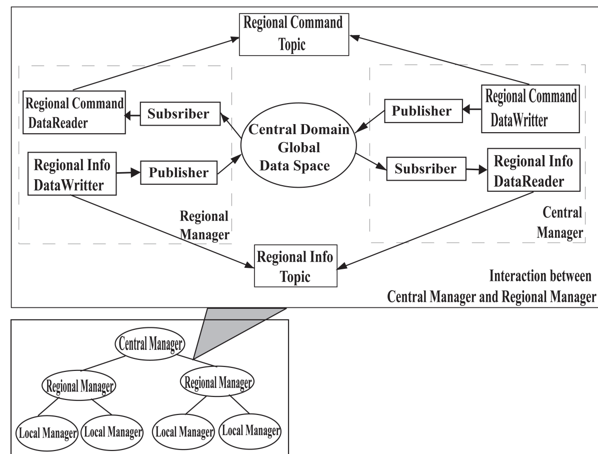

Figure 7. Interactions between Central Manager and a Regional Manager

4. Monitoring and Messaging Framework

The monitoring framework is required to support the autonomic capabilities such as self-management and self-healing. Two sets of values need to be monitored: infrastructure resource, and workflow status. Infrastructure resources need to be checked before a workflow instance or a participant manager can be scheduled. For example, some workflows may output large amount of data, which requires disk space as an essential resource; while other workflows may have high demand for computing capability, thus CPU utilization need to be checked. If a workflow fails, it is a waste of resource and time required to start the workflow from the beginning. Workflow status record helps determine the failure reasons. Moreover, by using recorded intermediate results, a workflow can be resumed from the last failure point. Scientific workflows require that the results can be reproduced. Workflow status and the intermediate results provide data provenance for reproducibility.

The hierarchical network structure of monitoring is described in Figure 5. A local manager runs in each machine to collect and report local status. It gets monitoring control commands from a regional manager. A regional manager is a bridge between local managers and the central manager. It supervises a set of local managers, collects status reports from local managers, filters out necessary information, and transfers the information to the central manager also known as the global manager. A regional manager gets monitoring control commands from the central manager, and distributes the command to local managers it supervises. The central manager also collects all status reports, processes these reports and facilitates administrator to manage the system.

Existing computing nodes may be replaced or upgraded, and new nodes may join the computing network at any time. In both situations, easy extensibility is required. Traditional server-client communication model is not suitable in this case. In addition, a traditional client server model suffers from the single point failure and performance bottleneck. Additional servers may be a solution to single point failure. However, the mechanism of server communication and interaction is complex. Moreover, in our architecture, all managers consume status messages and distribute control commands and consume messages at the same time, acting as server and client simultaneously.

Data Distribution Services for Real-Time Systems(DDS): Data-Distribution Service for Real-Time Systems (DDS) [16] is a middleware standard for publish-subscribe communication model that overcomes the typical shortcomings of traditional client-server model. It features extensive Quality of Service (QoS) configurations. In publish-subscribe model, message sender and message receiver are decoupled. Messages are distributed by publisher without knowing who will receive the messages. Each message is associated with a special data type called

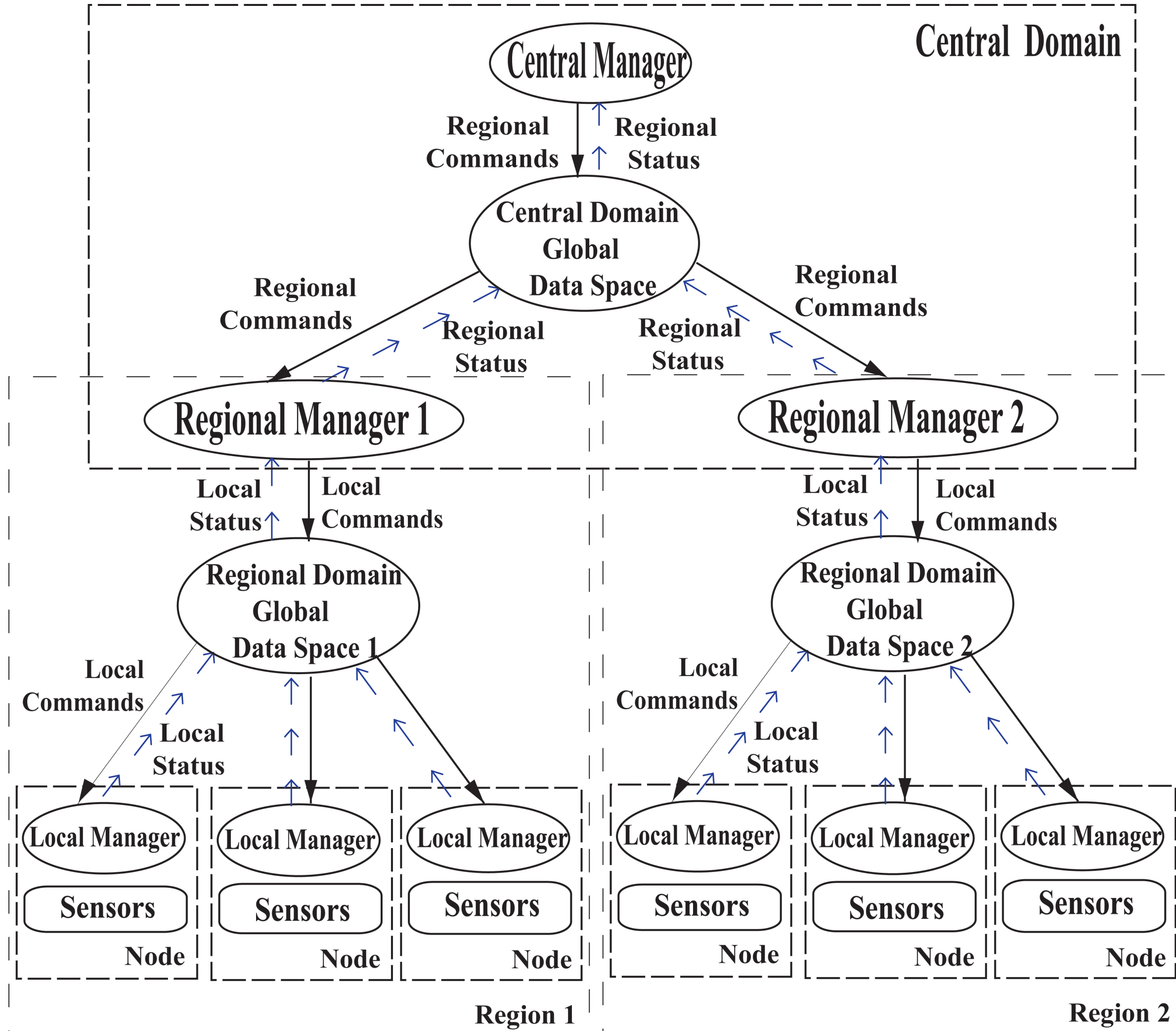

Figure 8. Data Delivery Channels. The host status information flows from bottom to top direction and the commands flow in the opposite direction.

topic. A subscriber registers to one or more topics of its interest. DDS guarantees that the subscriber will receive messages only from the topics that it subscribed to. In DDS, a host is allowed to act as a publisher for some topics and simultaneously act as subscriber for others.

4.1. Implementation Details

Two types of topics are required for workflow monitoring: control commands and host status information. Host status information is for reporting performance status including CPU utilization, memory utilization and so on. The host status information messages distribute the sensor values. A status message includes node name, sensor name and status information. Thus the receiver gets to know the status for a particular sensor at a specific node. As discussed in previous section, each computing machine runs a local manager. A local manager consists of a scheduler and a set of sensors. A scheduler runs each sensor either periodically or sporadically (according to the configured property). For periodic sensors, sensing period depends on the influence of sensor value on node's health, as well as the rate of change of the sensed parameter. For example, CPU utilization is critical for workflow scheduling, and changes rapidly. Thus the sensing time period is set to a short value (10 seconds). This scheduler as described in the introduction section, is based on a new hard real-time component framework [3]. The distributed synchronization of all sensors is achieved by an algorithm explained in [2].

Monitoring: Currently, we are using sixteen periodic sensors on all nodes (see table 1). The heartbeat sensor is critical for monitoring. A heartbeat sensor sends out liveness mes-

Sensor Name	Period	Description
CPU Utilization	10 seconds	Aggregate utilization of all CPU cores on the machines.
Swap Utilization	10 seconds	Swap space usage on the machines.
Ram Utilization	10 seconds	Memory usage on the machines.
Hard Disk Utilization	60 seconds	Disk usage on the machine.
CPU Fan Speed	10 seconds	Speed of CPU fan that helps keep the processor cool.
Motherboard Fan Speed	10 seconds	Speed of motherboard fan that helps keep the motherboard cool.
CPU Temperature	10 seconds	Temperature of the processor on the machines.
Motherboard Temperature	10 seconds	Temperature of the motherboard on the machines.
CPU Utilization Per Process	10 seconds	CPU usage of each process on the machines.
Swap Utilization Per Process	10 seconds	Swap space usage of each process on the machines.
Ram Utilization Per Process	10 seconds	Memory usage of each process on the machines.
CPU Voltage	10 seconds	Voltage of the processor on the machines.
Motherboard Voltage	10 seconds	Voltage of the motherboard on the machines.
Network Utilization	10 seconds	Bandwidth utilization of each network card.
Network Connection	10 seconds	Number of TCP connections on the machines.
Heartbeat	300 seconds	Periodic liveness messages.

Table 1. Sensors

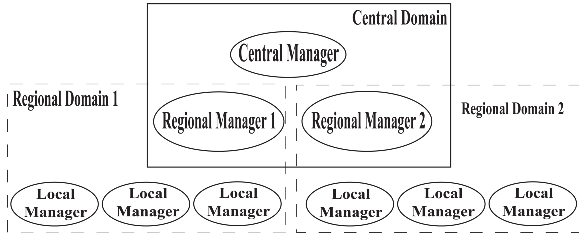

Figure 9. Domain Deployment

sages periodically (see table 1), which indicates the host's availability. If two or more contiguous liveness messages are missing, the server pings the host to check whether the host is alive or not. Meanwhile, jobs on the failed node are rescheduled on other nodes, which avoid potential delay of workflow completion. Each sensor collects a type of performance status values, and stores these values locally. In normal mode, the sensor's value is allowed to fluctuate within a certain range. Sensor reports the value only if the difference between current value and recorded value becomes greater that the specified dead band.

Control Commands: Control commands are responsible for controlling sensors. Currently, we support five different commands: STOP (stop the sensor), START (start the sensor), RELOAD (reload the library consisting sensor implementation), CONFIG (change the sensor configuration parameters, including period, deadband and threshold), QUERY(specifically ask for the value of a particular sensor). There are two levels of control command topics: regional commands, which are distributed from the central manager to a selected regional manager, and local commands, which are issued by regional managers to respective local managers. Central managers can issue local commands by using a regional command for the corresponding region and wrapping the local command in the header. A regional command consists of command type, sensor name, and command content. The central manager distributes regional commands with the respective regional ID embedded in the message header. When a regional manager receives a regional command, it extracts the local command, and distributes the command to all local mangers in that region .

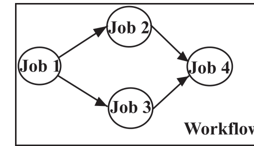

Figure 10. Workflow Example

Manager Interactions: The interactions between the central manager, regional managers and local managers are shown in figure 6 and figure 7. In our monitoring framework, the central manager publishes regional commands, and is a subscriber for regional status messages. Regional managers subscribe to local status messages and regional commands, process these messages (and filter them) and commands, and publish regional status messages and local commands. Local managers publish local status messages, and subscribe to local commands. Figure 9 shows the domain deployment for the monitoring framework. Since the central manager only interacts with regional managers, and a local manager only interacts with its regional manager that supervises the region, there is no need for local managers to communicate directly with the central manager. We deploy a central domain, which is responsible for communications between the central manager and regional managers. A regional domain contains a regional manager and several local managers. This design eliminates the possibility that network overload occurs at central manager. Every region is distinguished using a unique ID. The typical flow of data in this framework is shown in figure 8.

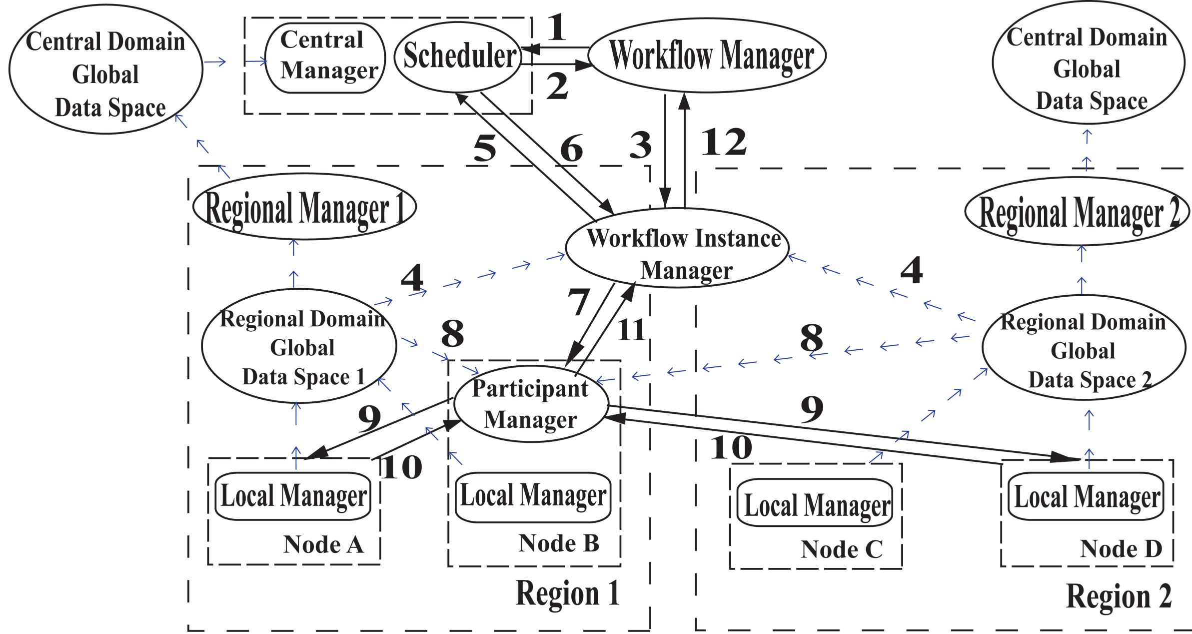

Figure 11. Combined Infrastructure

5. Integration With the Workflow System

In previous sections, we described the workflow managing framework and their timed state machine semantics. We also discussed the monitoring framework. In this section we will discuss their integration by using a simple representative example. Consider a workflow shown in figure10, which consists of four participants: job 1, job 2, job 3, and job 4 that has been submitted to workflow manager. Participant 1 is required to execute first. Then participant 2 and 3 are executed before participant 4 works. The order of participant executing order can be represented as $\{1\} \rightarrow \{2,3\} \rightarrow \{4\}$.

The workflow instance manager (WIM) checks the resource requirement for participant with the scheduler, and return sets of nodes(N) and regions(R). The requirements for a participant are represented as $N = \{n_1, n_2 \ \ldots n_i\}$, and $R = \{r_1, r_2 \ \ldots r_j\}$, where $Children(r_k) = \{n_p \ \ldots n_q\}$ and $N = \bigcup(Children(R))$.

Assume that the requirements for participant Job1 are $N = \{n_A, n_B, n_D\}$, $R = \{r_1, r_2\}$, where $Children(r_1) = \{n_A, n_B\}$ and $Children(r_2) = \{n_D\}$. The integrated infrastructure for this example is shown in figure 11. Each region contains two nodes with local managers, a regional manager, and a regional domain data space. Local managers send their status information data to the regional domain data space. The scheduler and the central manager run on the same host. The central manager and regional managers are in the central domain, and share status information through central domain. The workflow manager runs on a separate host. During the workflow execution, the workflow instance manager subscribes to different domains at different times. Following steps illustrate the sequence of activities:

1. When a scientist submits a workflow, the workflow manager sends a request to the scheduler and asks whether it can run the workflow or not.

2. The scheduler checks resource availability, and returns a union of possible regions (region 1 and region 2) with resources. Notice that the union of possible resource regions may differ from the final regions that will execute the workflow.

3. The workflow manager creates a workflow instance manager in a separate host, and tells the workflow instance the union of regions(region 1 and region 2) that may have the required resources.

4. The workflow instance manager joins all these regions (region 1 and region 2) by subscribing to their domain, and starts listening to node status information in these regions.

5. Workflow instance manager sends a request to the scheduler when it is ready to run a participant.

6. The scheduler sends back the exact set of nodes and their regions (node A and node B in region 1, node D in region 2) for running the participant.

7. The workflow instance manager starts participant manager in any of the given nodes. This node is also known as the job head node. Data for running the participant (e.g. binaries, the set of nodes and their regions for executing the participant) is sent to participant manager when it is created.

8. The participant manager joins the given regions (regional 1 and region 2) and starts listening to the status of specific nodes that it has been allocated to (node A in region 1 and node B in region 2).

9. The participant manager finishes initialization and starts firing when the data tokens are present (node A in region 1 and node B in region 2).

10. When a node finishes its work, it returns the results to participant manager.

11. When the participant manager gets results back from all given nodes (node A in region 1 and node B in region 2), and gets the final result of the participant, it sends the final result to the workflow instance manager. At that time, the participant manager will stop running.

12. The workflow instance manager checks DAG dependencies, and starts scheduling job 2 and job 3. When all jobs are finished and the whole workflow instance is completed, the workflow instance manager returns results to the workflow manager. The workflow instance manager then leaves all regions, and is destroyed.

Notice that the workflow instance manager and participant managers dynamically subscribe to the sensor topics that they are interested in. Reliable QoS provided by DDS framework ensures that the participant manager or the workflow instance manager will receive the data even if any one of the nodes in the region fails. In such a case, regional managers are automatically restarted on a different node in the region. The framework is reliable even in case of failure of a worker node, participant manager failure, or workflow instance manager failure. **Node Failure**: If a non participant manager node, A, fails, its heartbeat will be lost. When the participant manager (in node B) realizes that the node A is lost it will schedule the work to other node.

Participant Failure: If a participant manager (in node B) fails, its heartbeat will be lost. The workflow instance manager then restarts the participant manager for executing the participant on another set of nodes. The workflow manager can selectively restart the failed participant from the failed queue. Thus, it does not need to start the workflow from beginning. In case a participant cannot recover (a consecutive failure), the workflow instance manager uses the algorithm outlined in [12] to execute the maximal workflow that can finish and informs the administrator.

Workflow Instance Manager Failure: If a workflow instance manager fails, its heart beat will be lost. the workflow manager then creates a new workflow instance manager for executing the workflow instance again. However, it recovers the information about the participants that have already been completed successfully from the database. Thus, it restarts the session and reruns only the participants that have not executed yet.

Workflow Manager Failure: The central manager recreates the workflow manager if its heartbeat is lost.

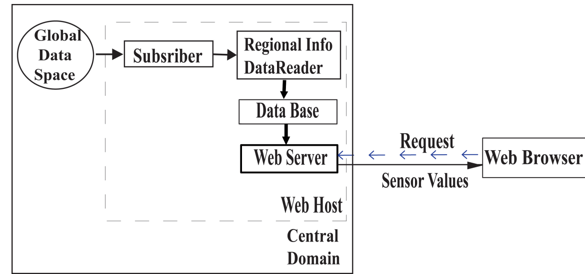

Figure 12. Web Extension Architecture

Central Manager Failure: Due to the reliable Quality of service provided by DDS, Central manager data topic is not lost even if the node hosting central manager fails. We can just re-instantiate the central manager on another node and it can retrieve all the previous data without any loss. We can see that this architecture does not suffer from a single point of failure. This is made possible by the use of reliable data distribution services framework.

6. Monitoring Web Services

Due to the flexibility of our monitoring framework, extra modules can be plugged in at runtime. A subscriber or publisher can join or leave a domain at any time, without affecting other participants existing in that domain, or causing any changes to current architecture. For example, we have developed a web services extension that plots the monitoring values. With this extension, a scientist is able to learn the current performance of any host at run time from a web browser remotely, without installing any special client application. The structure of this service is shown in figure 12. Since only status information is needed, we omit command delivery in this figure. It is possible to extend this service so that the user can directly issue control commands from the browser. The web host contains a DDS subscriber, and a database service. The DDS subscriber joins central domain and registers to status information topic. When status information is published by workflow instance manager, the web host receives it and stores it into a database. On the client side, a scientist views status information through a web browser. A request for a particular sensor values is sent from the web browser to the web host. When the web server gets the request, it queries sensor status data from the database, and sends it back to the client.

7. Conclusions

In this paper, we described the design and implementation of a new workflow management system using the OMG Data Distribution Service (DDS) standard. It is an extension of our previous work on the autonomic scientific workflow management systems by presenting a hierarchical dynamic workflow management system that tracks the state of job execution using timed state machines. We described the

two specific channels of information flow: monitoring status message and control commands. The advantages of using DDS were also discussed. Firstly, it allows us to be dynamic in that publishers and subscribers can be created at runtime. Secondly, we eliminate any single point of failure in the network. We also presented timed state machine based formal semantics for all the major actors in the framework. Future work in this area involves further study of workflow-resource mapping, and formal verification of the state machine semantics.

8. Acknowledgment

This work was supported in part by Fermi National Accelerator Laboratory, operated by Fermi Research Alliance, LLC under contract No. DE-AC02-07CH11359 with the United States Department of Energy (DoE), and by DoE SciDAC program under the contract No. DOE DE-FC02-06 ER41442. We are grateful to the help and guidance provided by Ted Bapty, Sandeep Neema, Jim Kowalkowski, Jim Simone, Don Holmgren, Amitoj Singh, Nirmal Seenu and Randolph Herber.

References

[1] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237, 2005.

[2] A. Dubey, G. Karsai, and S. Abdelwahed. Compensating for timing jitter in computing systems with general-purpose operating systems. *ISORC*, 2009.

[3] A. Dubey, G. Karsai, R. Kereskenyi, and N. Mahadevan. Towards a real-time component framework for software health management. Technical Report ISIS-09-111, Institute for Software Integrated Systems, Vanderbilt University, 11 2009.

[4] A. Dubey, S. Nordstrom, T. Keskinpala, S. Neema, T. Bapty, and G. Karsai. Towards a model-based autonomic reliability framework for computing clusters. *EASE'08*, pages 75–85, 2008.

[5] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: from process modeling to workflow automation infrastructure. *Distrib. Parallel Databases*, 3(2):119–153, 1995.

[6] A. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, C. Goble, M. Livny, L. Moreau, and J. Myers. Examining the challenges of scientific workflows. *IEEE Computer vol*, 40:24–32, 2006.

[7] A. Goldberg and G. Horvath. Software fault protection with ARINC 653. In *Proc. IEEE Aerospace Conference*, pages 1–11, March 2007.

[8] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[9] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[10] J. Kephart and D. Chess. The vision of autonomic computing. *IEEE Computer*, 2003.

[11] B. Ludscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system. In *Concurr. Comput. : Pract. Exper*, page 2006, 2005.

[12] S. Nordstrom, A. Dubey, T. Keskinpala, R. Datta, S. Neema, and T. Bapty. Model predictive analysis for autonomic workflow management in large-scale scientific computing environments. In *EASE '07*, pages 37–42, 2007.

[13] L. Piccoli, A. Dubey, J. Kowalkowski, and J. Simone. Lqcd workflow execution framework: Models, provenance, and fault-tolerance. *Journal of Physics: Conference Series*, 2009. Accepted for Publication.

[14] S. M. Serra da Cruz, F. N. da Silva, L. M. R. Gadelha, M. C. Reis Cavalcanti, M. L. M. Campos, and M. Mattoso. A lightweight middleware monitor for distributed scientific workflows. In *Proc. 8th IEEE International Symposium on Cluster Computing and the Grid CCGRID '08*, pages 693–698, May 19–22, 2008.

[15] R. Sterritt. Towards autonomic computing: effective event management. In *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*, pages 40–47, 5-6 Dec. 2002.

[16] OMG. Data distribution service for real-time systems, v1.2. Technical report, Object Management Group, 2007.