12-10-2008

Capturing Workflow Event Data for Monitoring, Performance Analysis, and Management of Scientific Workflows

Matthew Valerio

Satya S. Sahoo
Wright State University - Main Campus

Roger Barga

Jared Jackson

Capturing Workflow Event Data for Monitoring, Performance Analysis, and Management of Scientific Workflows

Matthew D. Valerio[1,2], Satya S. Sahoo[1,3], Roger S. Barga[1], Jared J. Jackson[1],
[1]*Microsoft Research,* [2]*EASE Lab, The Ohio State University,*
[3]*Kno.e.sis Lab, Wright State University*
valerio.8@osu.edu, sahoo.2@wright.edu, {barga, jaredj}@microsoft.com

Abstract

To effectively support real-time monitoring and performance analysis of scientific workflow execution, varying levels of event data must be captured and made available to interested parties. This paper discusses the creation of an ontology-aware workflow monitoring system for use in the Trident system which utilizes a distributed publish/subscribe event model. The implementation of the publish/subscribe system is discussed and performance results are presented.

1. Introduction

Recently there has been a significant amount of interest in capturing and recording various aspects of the execution of scientific workflows. Storing provenance data for workflows, monitoring the resource usage of workflows, and providing monitoring support for efficient workflow management are just a few applications that consume information about executing workflows across distributed machines. This paper explores the need for effective workflow monitoring tools and describes the implementation of a general-purpose high-performance publish/subscribe eventing system for monitoring all aspects of workflow execution. This publish/subscribe system has been implemented to provide monitoring support for the Trident workflow system[1].

2. Motivation

In any workflow execution system, it becomes necessary to provide answers to a number of questions from a user's and management's perspective, such as:

- What is the status of my workflow?
- How long will it take to complete?
- How many resources does it consume?
- Why did it fail at a certain point?
- What was the execution context of the workflow directly before it failed?

- How has each activity in my workflow performed in the past?
- Did the user make changes to the workflow before executing it?

It can be seen from these questions that a number of features are needed.

Workflow monitoring can be divided into three main categories – execution status monitoring, resource monitoring[2], and workflow evolution monitoring[3].

Execution status monitoring refers to the ability to track events related to both activity execution and workflow execution. Knowing when individual activities start and stop allows histograms of execution time for each activity to be generated in an effort to provide accurate estimations of total workflow completion based on past executions of workflows containing similar activities. In addition, tracking overall workflow events such as starting, stopping, going idle, fault condition, and being serialized and deserialized from durable storage allows a comprehensive picture of the workflow health to be presented to the. If the workflow has failed at any point, the user can take corrective action knowing the complete history of events leading up to the workflow failure.

Resource monitoring allows a user to track the resource usage of workflows at a number of granularities. For example, resource usage (e.g. disk space, CPU usage, memory footprint, etc) can be monitored before and after each activity (individual unit of work within a workflow) to determine the resource usage of that activity. Resource usage of an entire workflow can then be inferred based on the resources used by all of the activities comprising the workflow. In addition, an activity may provide certain user-defined tracking events during the execution of the activity as a hint to the workflow runtime that its resource usage should be profiled. Monitoring resource usage of individual activities and workflows allows for cost models to be constructed for each activity, enabling many useful possibilities. For example, if cost models are available for a number of sequentially-

programmed activities and the data bindings between activities indicate no direct dependence, a "smart" workflow monitoring system could suggest execution of these activities in parallel, possibly on different machines, if the cost models show that this is feasible.

Workflow evolution monitoring attempts to capture design-time (as opposed to runtime) changes to a workflow as it is being edited. Such changes would normally involve the addition or subtraction of activities, changes in workflow execution logic, changes to input parameters, and changes to data bindings between activities. These changes are being captured to allow for the possibility of rich provenance services to track the entire history and pedigree of a workflow as it evolves over time.

Given that there exist a number of potential event sources (workflow execution, resource usage, workflow designer, etc) and there exist a number of potential event sinks (logging, monitoring, provenance, etc) of monitoring data potentially executing on different machines within a distributed system, a good candidate for the overall system architecture is a publish/subscribe system[4]. In the next section we present the high-level architecture of an extensible end-to-end workflow monitoring system based on the publish/subscribe model.

3. Proposed Solution
3.1 Blackboard

The proposed solution for capturing monitoring data involves the integration of a number of event publishers and a number of event subscribers using a high-performance publish/subscribe architecture that we call the "blackboard". Publishers in varying forms gather monitoring information about workflows and post messages to the blackboard. In turn, these messages get routed to any number of subscribers. A high-level view of this architecture is shown in Figure 1.

In this architecture, the monitoring service is only one of many possible services that will consume published event information about workflow execution. Other subscribers could include a simple logging service that logs messages to a file or a database, or a more sophisticated provenance service for capturing all aspects of an executing workflow, the data on which it operates, the webservice calls that it makes, and many other related types of information.

This architecture allows for a number of extensibility points. For example, any number of message publishers can be added to the system at any time, depending on the need. If only workflow runtime events (as opposed to workflow designer events) are required, then only that publishing service can be utilized. Similarly, any number of subscriber services can be used depending on the needs. If, for example, provenance and logging are not needed, then those services may be ignored and only the monitoring system will register with the Blackboard to receive event notifications.
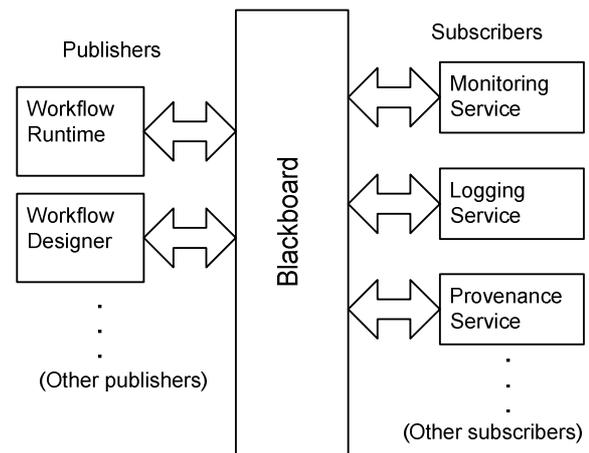

Figure 1 – Overall blackboard architecture

Messages are passed from publishers to the blackboard and from the blackboard to the subscribers by using a "blackboard message". Each blackboard message contains a collection of key-value pairs where both the key and value are strings.

Subscribers indicate their interest in receiving certain messages by sending the blackboard a subscription profile containing a list of blackboard message keys. Only those key-value pairs of a blackboard message for which a subscriber has expressed an interest are published to the subscriber.

From a performance view, if a number of publishers produce a large amount of messages, it is wasteful of network resources to send those messages to the blackboard if no subscribers are listening. Typical workflow monitoring scenarios have the potential to produce very detailed messages, but subscribers may not be interested in all details. To address this problem, it is possible for the blackboard to direct each publisher to only publish messages that are contained in a subscription profile.

The subscription profile that the blackboard sends to the publisher is called the "aggregate subscription profile" and is determined by the blackboard to be the set union of the subscription profile from all subscribers. If at any time the aggregate subscription profile changes, then all publishers are notified. When a publisher receives this aggregate subscription profile,

it ensures that it only publishes messages containing concepts that have been requested by at least one subscriber. This eliminates unnecessary network overhead and stops an event from being published if nobody is listening.

To receive notifications of the aggregate subscription profile, each publisher must also register and unregister itself with the blackboard. The blackboard internally maintains a list of all currently-active publishers and subscribers. If at any point it cannot contact a publisher or subscriber, it is removed from the list. The publisher and subscriber lists are persisted to disk to provide fault-tolerance if the blackboard should fail.

Typical publish/subscribe systems will include expressions in the subscription profile to be evaluated to determine if a message should be published. This stems from the need for an expressive way to fine-tune the types of messages a publisher receives. Another (non-exclusive) method to address this need can be accomplished by assigning a clearly-defined ontological concept to each key-value pair of the blackboard message. In this system, the ontological concept is the key, and subscribers inform the blackboard of their subscription profiles as a list of ontological concepts that it wishes to receive, essentially forming a Knowledge Based Network (KBN)[5]. Because the key of the key-value pair within a blackboard message is an ontological concept, blackboard messages can be viewed as groups of "concept-value pairs". Expression evaluation by the subscription profile will be added in the future.

It should be noted here that the blackboard system is completely problem-agnostic and is not tied directly to the application of workflow monitoring. Any number of publishers and subscribers for any number of applications could be added to the system later. Now that the publish/subscribe foundation has been described, we can turn our attention to the specific publishers and subscribers necessary to perform workflow monitoring.

3.2 Publishers

The most important aspect of the workflow monitoring system is the ability to extract information regarding workflow and activity execution from the workflow runtime.

Windows Workflow Foundation (WF) provides for a number of extensibility mechanisms, of which one is the ability to provide custom services that are loaded into the workflow runtime.

To this end, a custom tracking service has been created to accept these event notifications from the runtime, filter them according to the aggregate subscription profile, and publish the messages to the blackboard system.

From the vantage point of the tracking service within the workflow runtime, a number of important duties can be performed. Since the tracking service can subscribe to all of the events provided by the runtime, it can perform specific actions, such as message publication and resource monitoring, at critical points in the workflow execution process.

The first duty of the tracking service is to publish event notifications as a workflow executes. Three types of events can be obtained from the workflow runtime – workflow tracking events, activity tracking events, and user tracking events. Workflow tracking events indicate changes occurring to the workflow as a whole, such as whether it has been initialized, started, stopped, persisted, resumed, etc. Activity tracking events provide more fine-grained information about the state changes for individual activities. Even more, the user tracking events allow for the possibility for activity designers to emit customized tracking events from within an activity, yielding even more fine-grained information about the actual execution of the code contained in the activity. All of these events are listed in Figure 2.

```
Workflow Events    Activity Events
Aborted             Canceling
Changed             Closed
Completed           Compensating
Created             Executing
Exception           Faulting
Idle                Initialized
Loaded
Persisted          User Events
Resumed             User-defined
Started
Suspended
Terminated
Unloaded
```

Figure 2 – Events provided by the workflow runtime

Each of the workflow and activity events is mapped directly to a concept in the ontology. The aggregate subscription profile supplied by the blackboard contains a list of concepts for which there are subscribers. When the custom tracking service receives tracking data, each event and property value of the tracking record is mapped directly to a concept in the ontology. These concept-value are filtered based on the concepts listed in the aggregate subscription profile. The remaining concept-value pairs are packaged into a blackboard message and published.

The second duty of the tracking service is to perform resource monitoring capabilities. To accomplish this, a number of "resource monitors" may register with the tracking service and indicate which resources they are capable of monitoring. This is done by specifying a "resource concept" in the ontology (CPU usage, memory usage, disk usage, etc) that it can monitor.

Two main scenarios can trigger the evaluation of resource monitors. First, resource monitoring may be initiated at the beginning of an activity. Second, resource monitoring may be user-initiated from within the code of an activity.

For the first case, when the tracking service is notified of events pertaining to the starting and stopping of activities, it checks its aggregate subscription profile to determine if it contains any resource concepts. If so, the corresponding resource monitor is invoked, producing an output value (a string) that is included in the outgoing blackboard message. However, some resource monitoring scenarios (such as CPU monitoring) cannot be accurately captured by a single event and must run alongside the executing activity. To support this scenario, resource monitors may be constructed to allow periodic execution on a separate thread during the execution of the activity, alerting the resource monitoring service when it has a value that should be published. This implementation allows for the possibility of a resource monitor to either produce frequent messages (useful for real-time monitoring) or long-running general information messages (useful for capturing averages). This implementation allows subscribers to construct cost models associated with the resource usage of individual activities.

The second case involves a user-initiated tracking event from within the activity code to invoke a resource discoverer. In this scenario, the activity code will create a user tracking record with a specific format that indicates the resource concept for which to execute a resource discoverer. This allows fine-grained resource monitoring as an activity is executing to support real-time monitoring.

In addition to execution and resource monitoring, design-time monitoring event data may be published. This data will be collected in the workflow editor application (a graphical interface such as the Trident workflow designer[6]) and published to the blackboard.

3.3 Subscribers

Subscriber services receive messages from the blackboard based on a list of concepts (the subscription profile) supplied to the blackboard.

The simplest example of a useful subscriber is that of a logging service that naively stores messages directly into a file or database. Such a service is useful in some scenarios (e.g. debugging), but doesn't provide an efficient method for analyzing the received messages to extract useful trends.

A monitoring service not only receives workflow execution events, but also processes these events and stores them to allow efficient execution of relevant queries. These queries involve specific information involving resource usage and execution time for every version of every activity and workflow in the system. This rich set of information is made available as a webservice to allow other consumers to make decisions based on this information. This allows for a number of interesting possibilities, such as a user being shown an estimated time to completion as they are designing a workflow, or the activities of a workflow being scheduled in parallel even though the workflow indicates sequential execution.

In addition, it is possible for a provenance service to be constructed which utilizes event data from both workflow execution and design-time publishers. This provenance service allows the complete history of the workflow to be stored in such a way that it can be executed at a later date if needed. Discussion of the provenance service is well outside the scope of this paper[7].

4. Implementation

4.1 Network Communication

To facilitate the blackboard system as a research platform, all network communication is hidden behind an extensible interface. The links between the subscriber and blackboard and between the blackboard and publisher can have any implementation "plugged in".

Currently only an implementation utilizing Windows Communication Foundation (WCF) has been constructed and tested. The low-level transport binding was chosen to be TCP binary for maximum performance, though the WCF implementation can be reconfigured to use any other binding (HTTP, MSMQ, etc) without any problems.

Another implementation using Decentralized Software Services (DSS), a foundational component to Microsoft Robotics Developers Studio (MRDS), is planned. Alternative implementations could also be added through the provided blackboard interfaces.

4.2 High-performance Message Distribution

The blackboard must store a comprehensive list of all subscribers and the concepts for which they are subscribed. The subscription store is illustrated in Figure 3.

http://subscriber1	{Concept1, Concept2, ...}
http://subscriber2	{Concept2, Concept3, ...}
http://subscriber3	{Concept1, Concept3, ...}

⋮

Figure 3 – Subscription store showing the list of concepts for which a particular subscriber has expressed interest

Because many messages will be flowing through the blackboard, the implementation needs to be as performant as possible. To this end, the blackboard utilizes a reverse lookup index to allow fast distribution of incoming messages to subscribers. Since subscription and unsubscription requests are assumed to happen much less frequently than message publications, it makes sense to optimize the message separation and publication algorithm by creating a "subscriber lookup index".

The subscriber lookup index (SLI) structure is shown in Figure 4. The index is constructed from the subscribers and associated concepts in the subscription store.

Concept1	{http://subscriber1, http://subscriber3, … }
Concept2	{http://subscriber1, http://subscriber2, … }
Concept3	{http://subscriber2, http://subscriber3, … }

⋮

Figure 4 – Subscriber lookup index (SLI) showing the list of subscribers for each concept

The subscriber lookup index keeps a list of all subscribers that have subscribed to receive notifications for a specific concept. Since a subscriber can be uniquely identified by its URI, the URI is stored in the collection of current subscribers for the concept. The URI also serves as the return address for the service on the subscriber that is listening for messages published by the blackboard. The idea for this implementation is similar to that of the UNIX inode table which is intended to track the processes accessing an individual file.

When the blackboard is initialized, it loads the list of subscription profiles if it exists and constructs an initial subscriber lookup index. The pseudocode for constructing the initial subscription lookup index is shown in Figure 5.

```
map<uri,set<concept>> subStore = load()
map<concept,set<uri>> sli = Ø

proc initialize_sli() {
  foreach sub in subStore {
    foreach concept in sub.concepts {
      sli[concept].add(sub.uri) }}}
```

Figure 5 – Pseudocode for initializing the subscriber lookup index

When a new subscriber sends a new subscription profile to the blackboard, the subscription profile store as well as the subscriber lookup index must be updated. The algorithm for the subscribe operation is shown in Figure 6.

```
proc subscribe(uri address, set<concept> profile)
{
  subStore[address] := profile
  foreach concept in sli.concepts {
    if concept ∈ profile {
      sli[concept].add(address)
      profile.remove(concept) }
    else {
      if address ∈ sli[concept] {
        sli[concept].remove(address)
        if sli[concept] == Ø
          sli.concepts.remove(concept)
    }}
  foreach concept in profile {
    sli.add(concept,address)
  }}
```

Figure 6 – Pseudocode for the subscription operation

Each concept in the existing subscription lookup index is checked to determine if it is contained in the incoming subscription profile. If it exists, then the subscriber's address is added to the list of subscribers for that particular concept in the lookup index. If not, then it is removed from the list of subscribers. If the last item from the list of subscribers was removed, then the entry for that particular concept is completely removed from the subscription lookup index.

When a subscriber wishes to unsubscribe from the blackboard, it sends its unique return address. This return address is removed from the subscription store, and the subscriber lookup index is adjusted accordingly. For each concept in the subscription profile of the removed address, the list of subscriber addresses in the subscriber lookup index is retrieved, and the address is removed. If the last address was removed, then the entire concept entry is also removed from the subscriber lookup index. The peudocode for this algorithm is shown in Figure 7.

```
proc unsubscribe(uri address) {
  if address ∈ subStore.uris {
    profile := subStore[address]
    subStore.remove(address)
    foreach concept in profile {
      if concept ∈ sli.concepts {
        if address ∈ sli[concept] {
          sli[concept].remove(address)
          if sli[concept] == ∅ {
            sli.remove(concept)
          }}}}}
```

Figure 7 – Pseudocode for the unsubscription operation

Because of the initial effort invested in maintaining the subscriber lookup index, the message publication process is straightforward. The algorithm is shown in Figure 8.

```
proc send_messages(message m) {
  map<uri,message> outgoing = ∅
  foreach concept,value in m {
    foreach uri in sli[concept] {
      outgoing[uri].add(concept,value) }}
  foreach uri,message in outgoing {
    send(uri,message) }}
```

Figure 8 – Pseudocode for message publication

This process is optimized to require minimum computational cost for every message that is published. The simplicity of the concept-value messages leads to higher performance than other ontology-aware publish/subscribe systems[8].

While not shown in the above pseudocode, logic is also necessary to determine specifically when the aggregate subscription profile changes. It is conceivable that a subscriber may send multiple identical subscription profiles to the blackboard. In this case, the aggregate subscription profile will not change, and there is no need to notify all publishers of the new aggregate profile. This case is handled in the implementation by detecting the condition where the subscription profile is modified, and only notifying the publishers when the aggregate profile actually changes.

It should also be noted that the blackboard maintains a list of all registered publishers as well. However, since only the address of the publisher needs to be stored, the implementation is trivial.

4.3 Failure recovery

The blackboard provides for the possibility that it may fail, requiring that it return to its previous state. To this end, both the subscriber and publisher store are persisted to disk when the blackboard has detected that these data structures have changed.

Many other scenarios can be envisioned regarding the loss of messages over the network and failure of a subscriber. At this point the blackboard implementation only provides a best-effort guarantee and makes no reliability guarantees regarding message delivery. For the initial case where a few dozen machines on the same local area network are communicating with the blackboard system, this does not present a problem. However as the system scales further, reliable messaging will be added in the future to address these scenarios.
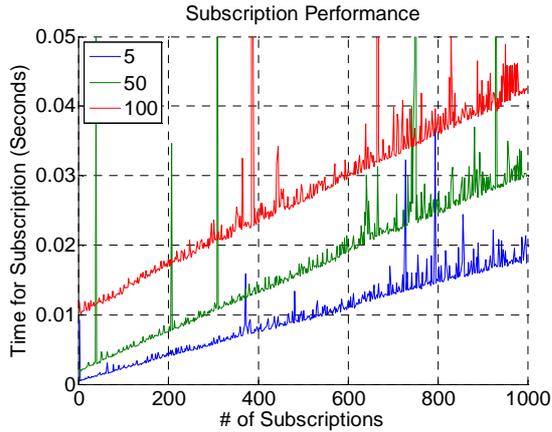
5. Performance

Three aspects of the blackboard were evaluated in isolation – subscription time, unsubscription time, and publishing latency. In addition, an end-to-end benchmark was also conducted that included the communication infrastructure (but no network latency).

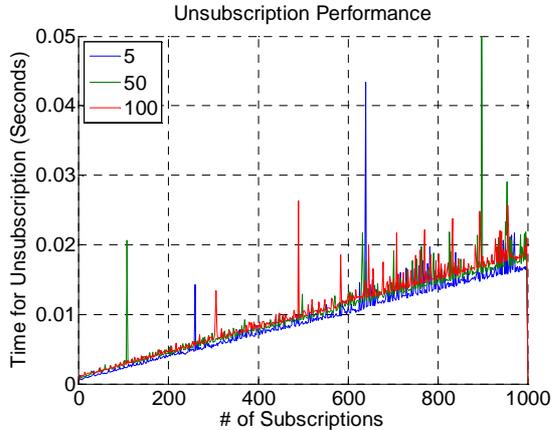### 5.1 Subscription and Unsubscription Time

To evaluate the subscription and unsubscription time of the blackboard in isolation, a series of "shims" were created to mock the interaction that the blackboard has with the underlying WCF services. These shims allowed the benchmarks to test only the core blackboard algorithms and exclude all functionality related to transmitting the messages over the network. All tests were performed on a machine running Windows Vista Ultimate having a 3.0 GHz Intel Core 2 Duo processor and 4 GB of RAM.

The subscription time (in seconds) was measured as a function of the number and size of subscriptions. A plot of this is shown in Figure 9. For this test, a subscription profile containing 5, 50, or 100 randomly-generated subscription items (concepts) was created for each iteration. Subscription items between two profiles had a 33% probability of overlapping between profiles. For each of the 1000 subscriptions, the time of performing the subscription operation was measured.

Subscription Performance

Figure 9 – Subscription time (in seconds) versus the number of subscriptions, for subscription profiles containing 5, 50, and 100 subscription items.

It can be seen from the plot that the performance of the blackboard scales linearly with the number of subscriptions. Also, the time required for the subscription operation increases with the number of subscription items, as expected. A number of large spikes in subscription time can be seen on the plot as a result of a number of noise sources, including other currently-executing processes and the non-deterministic nature of the .NET garbage collector.

Unsubscription Performance

Figure 10 – Unsubscription time (in seconds) versus the number of subscriptions, for subscription profiles containing 5, 50, and 100 subscription items.

Once the 1000 subscription operations were completed, each subscriber address was then unsubscribed from the blackboard. The time to complete the unsubscription operation was measured as a function of the number of subscriptions in the blackboard and the size of the subscription profile. The results are plotted in Figure 10.

The unsubscription times show a linear trend and an almost negligible dependence on the number of subscription items in the removed profile.

The linear trend in the performance of the blackboard subscription and unsubscription operations indicates that the blackboard scales very well under a large number of subscriptions.

5.2 Publishing Latency

Another good performance indicator is the time required to separate an incoming message into outgoing messages to each subscriber. For this benchmark, a fixed collection of 10 subscribers was created. To illustrate the worst-case scenario, the subscription profiles from each subscriber do not overlap. Each subscription profile contains 1000 concepts. A number of messages are constructed with sizes up to 1MB (10,000 concept-value pairs where the value is 100 bytes in length) and published to the blackboard. The publish operation was completed 10 times for each message and an average was recorded. The result is the plot shown in Figure 11.

It can be seen that the publish time increases linearly with the size of the message. In typical scenarios, most messages will contain less than 10kB of data, though this plot indicates that the blackboard can handle much larger messages. The blackboard publishing algorithm scales very well with increases in message size.

Publishing Latency
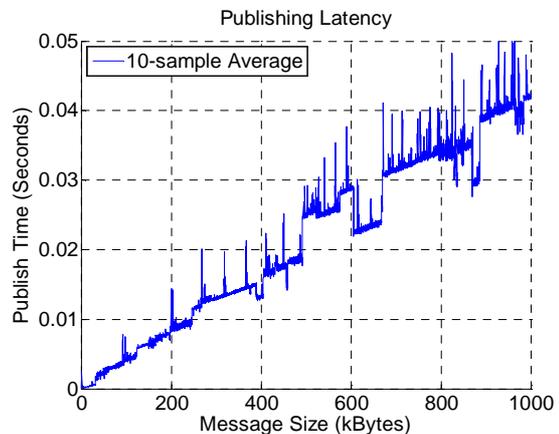
Figure 11 – Publishing latency (in seconds) versus message size. Each message of a certain size was sampled 10 times and the average is plotted.

5.3 End-to-end Benchmark

To quantify the number of messages per second the blackboard can handle in a realistic scenario, a full integration test was constructed involving three locally-

hosted publisher services, the blackboard, and three locally-hosted subscribers.

Each service was hosted in its own application domain (AppDomain) to separate it from the other services (while still being executed from the same process).

The WCF communication implementations were used to demonstrate a complete end-to-end system, while at the same time eliminating network latency since all traffic traveled over the loopback network adapter. In addition, the three publishers were executed using different threads to exercise the thread synchronization implemented within the blackboard.

During this test, 3000 messages (1000 from each publisher) carrying a 100-byte payload were published. It was found that the blackboard could handle around 1600 messages per second at a maximum throughput of 1.22 Mbps.

6. Demo

For this workshop, a demo of the monitoring system will be presented. This demo will showcase a full end-to-end scenario illustrating the execution of multiple workflows within the Trident system, event data being published to the blackboard, and event data in turn being consumed by both a logging and monitoring service. The monitoring service will provide a user-interface for interacting with both stored and real-time workflow events.

7. Conclusion

In conclusion, a method has been presented which allows multiple distributed workflow execution environments to publish event information to multiple listeners using a publish/subscribe system. Events may come from workflow execution, resource monitoring, or workflow design and be routed to logging, monitoring, and provenance systems. The design of the underlying publish/subscribe system has been presented, and benchmarks detailing its performance in real-world scenarios have been shown. A demo utilizing the Trident workflow system will be presented at the conference.

8. References

[1] Trident: Scientific Workflow Workbench for Oceanography, http://www.microsoft.com/mscorp/tc/trident.mspx.

[2] B. Balis, M. Bubak, M. Pelczar, "From monitoring data to experiment information – monitoring of grid scientific workflows", *IEEE International Conference on e-Science and Grid Computing*, Bangalore, India, December 2007.

[3] S. B. Davidson, J. Freire, "Provenance and scientific workflows: challenges and opportunities", *SIGMOD '08*, Vancouver, Canada, June 2008.

[4] P. T. Eugster, P. A. Felber, R. Guerraoui, A.-M. Kermarrec, "The Many Faces of Publish/Subscribe", *ACM Computing Surveys*, Vol. 35, No. 2, June 2003.

[5] D. Jones, J. Keeney, D. Lewis, D. O'Sullivan, "Knowledge-based networking", *International Conference on Distributed Event-Based Systems*, Rome, Italy, July 2008.

[6] R.S. Barga, D. Fay, D. Guo, S. Newhouse, Y. Simmhan, and A. Szalay, "Efficient scheduling of scientific workflows in a high performance computing cluster", *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments,* Boston, MA, June 2008.

[7] R. S. Barga and L. A. Digiampietri. "Automatic capture and efficient storage of e-Science experiment provenance", *Concurrency and Computation: Practice and Experience*, 20(5):419–429, 2008.

[8] J. Wang, B. Jin, and J Li, "An ontology-based publish/subscribe system", *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, Toronto, Canada, November 2004.