

Event-Driven Grid Computing

By: Brian Husted



410.781.6655
410.781.6625 fax
6700 Purple Martin Court
Eldersburg, MD 21784

Summary

The success of Service-Oriented Architecture (SOA) has created the foundation for information and service sharing across application and organizational boundaries. Through the use of SOA, organizations are demanding solutions that provide vast scalability, increased reusability of business services, and greater efficiency of computing resources. More importantly, organizations need agile architectures that can adapt to rapidly changing business requirements without the long development cycles that are typically associated with these efforts. Event-Driven Architecture (EDA) has emerged to provide more sophisticated capabilities that address these dynamic environments. EDA enables business agility by empowering software engineers with complex processing techniques to develop substantial functionality in days or weeks rather than months or years. As a result, EDA is positioned to enhance the business value of SOA.

The purpose of this white paper is to describe the approach employed to overcome the significant technical challenges required to design a dynamic grid computing architecture for a US government program. The program required optimization of the overall business process while maximizing scalability to support dramatic increases in throughput. To realize this goal, an architecture was developed to support the dynamic placement and removal of business services across the enterprise.

Problem Statement

The reference architecture described within this white paper was developed to prototype dynamic grid computing architecture for a US government program. Pyxis Engineering architects were presented with a unique set of constraints:

- Numerous services
- Resource intensive services
- OS and license-dependent services
- Time-consuming startup and initialization of services
- Dynamic demand for services
- High volumes
- Large datasets
- Architectural scalability
- Dynamic workflows

Due to these constraints, the services could not be concurrently deployed in-memory (service persistence) on every node within the compute grid. The resulting heterogeneous distribution of resource intensive services in a high-volume environment imposed a significant architectural challenge for the routing and load balancing of service requests. Legacy systems approached this problem by starting a new service instance for every service request and subsequently terminating the service upon completion. However, this brute force approach is highly inefficient since the startup and initialization of services may take considerably longer than the time to execute the business logic. To increase throughput in the system, service persistence was required.

A simple solution to enabling service persistence was to segregate services into separate hardware clusters. For example, server cluster X runs services A, B, and C and server cluster Y runs services E, F, and G. However, due to the random demand for services, traditional capacity planning techniques were insufficient to ensure the separate clusters of hardware would be efficiently utilized. To achieve scalability in such an unpredictable environment, an architecture that facilitated service persistence while optimizing the distribution of services based on the near real-time demand was required.



Research and Prototype

Pyxis Engineering architects led an extensive effort to research and prototype a proposed architecture. Pyxis Engineering prototyped several architectures using industry leading products based on Java Enterprise Edition (JavaEE), Enterprise Service Bus (ESB), Grid Computing, and batch file processing systems. Based on this research, Pyxis Engineering concluded that basic publish-and-subscribe architectures were inadequate to provide a comprehensive solution without significant custom development and a high risk of failure. Furthermore, Pyxis Engineering found that traditional grid computing solutions addressed very long-running and low volume transactions and thus could not satisfy the requirements.

Although the detail behind the limitations of each individual product is beyond the scope of this document, the following bullets outline the features that were lacking from the evaluated products:

- On-demand placement/removal of services
- Asynchronous high-volume communication to a heterogeneous distribution of services
- Pull-based load balancing to a heterogeneous distribution of services
- Correlation of events to enable complex near real-time processing decisions
- High-throughput asynchronous and dynamic workflows
- Complex service subscriptions

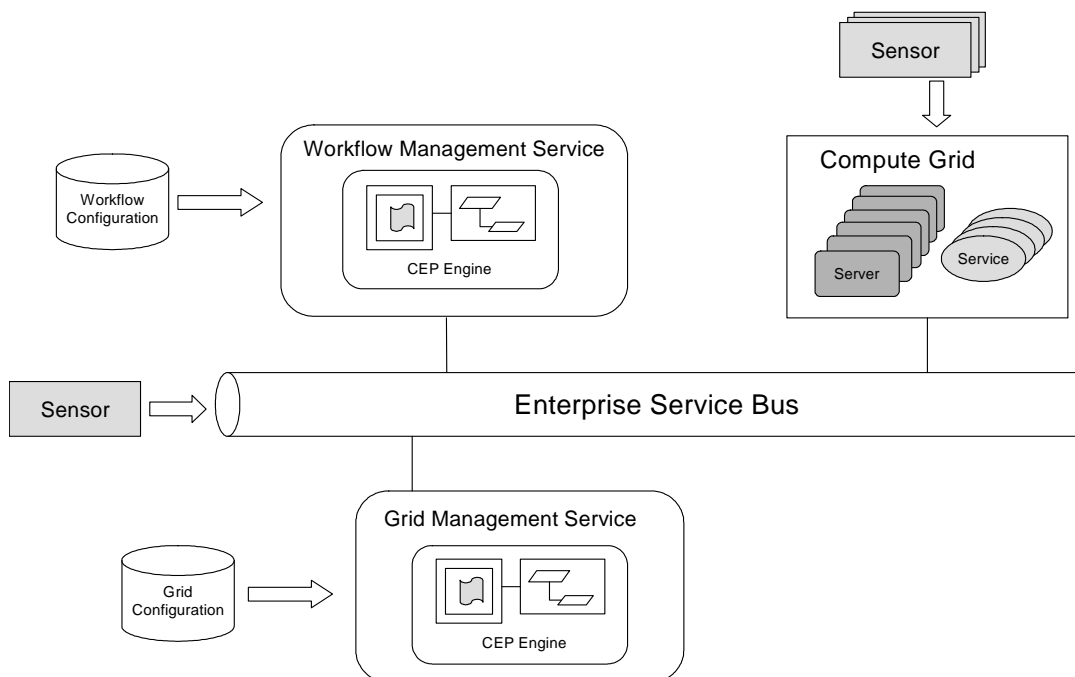
As a result from these findings, Pyxis Engineering turned to an emerging architectural style known as Event-Driven Architecture (EDA). EDA tools and techniques dramatically reduced both the complexity and time-to-delivery of developing the prototype.

Reference Architecture

Pyxis designed a reference architecture to provide a comprehensive solution to dynamically scale services, enable loose coupling and reusability of services, and to optimize the utilization of computing resources. The architectural style selected to best support this design was Event Driven Architecture (EDA). Complex Event Processing (CEP) was the primary EDA technique implemented, which greatly accelerated the development of both the workflow and grid management services.

Overview

The diagram below provides a logical view of the reference architecture. Almost every component listed in the diagram is both a consumer and producer of events. The grid management service continuously receives events from sensors that relay the near real-time state of the compute grid such as CPU, IO, memory, and queue depths. The workflow management service receives events from the clients requesting the execution of business transactions. Business services produce events to signal the completion of a business process. The Enterprise Service Bus (ESB) provides the asynchronous, high speed integration backbone for both the sending and receiving of events among the distributed services. Together these components provide the capability to adapt to unpredictable environments.



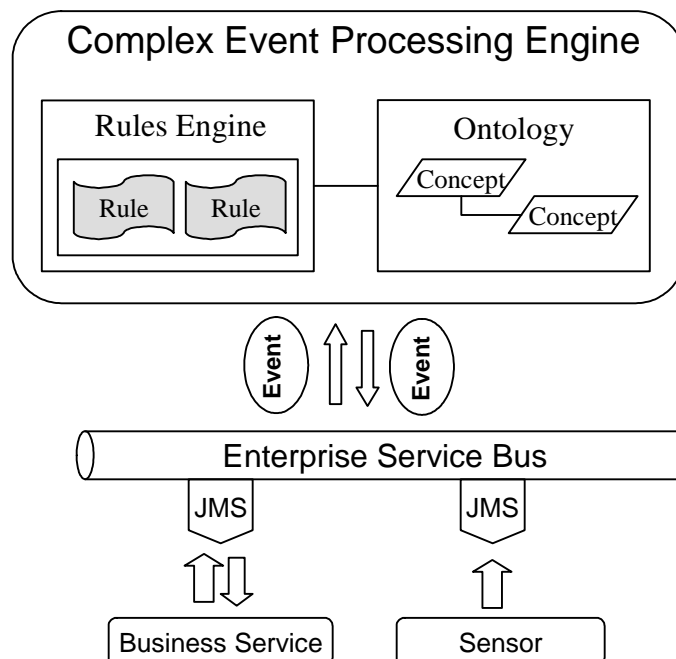
Complex Event Processing

The role of Complex Event Processing (CEP) is to recognize and react to conditions that impact the business process. Both the workflow and grid management services were implemented using CEP engines to facilitate service orchestration and optimal utilization of the compute grid. The CEP engine is an amalgamation of events, a data model (ontology), and a rules engine working in concert to signify a problem or imminent problem, opportunity, threshold, or a deviation within the business process. An ontology is a data model composed of concepts and their relationships to other concepts. A concept is an entity within the business process. A concept instance represents the state of a physical entity such as a server or transaction.

As events arrive, concept instances are updated to reflect the real-time state. The CEP engine continuously evaluates changes in the state against patterns of interest that impact the business. Once a pattern is identified, rules perform actions such as generating events to invoke a business service or to alert operational staff of an impending problem. For example, if the CPU and memory utilization on a particular server in the compute grid has been trending upwards for the last 10 minutes, the grid management service would react by temporarily marking the server as unavailable. In turn, this would prevent new services from being added to an over-utilized resource. Likewise, once the server returns to an acceptable state, the server would be marked as available for service deployment.

Enterprise Service Bus

The Enterprise Service Bus (ESB) is an abstraction layer between the CEP services, business services, and sensors. ESB provides the communication backbone to federate event-driven services in a loosely coupled fashion. The role of the ESB is to route and transform messages irrespective of the communication protocol used by the service. As a result, service endpoints, protocols, and transforms can be altered without impacting source code or business logic. This transparent approach provides a great degree of flexibility within the services allowing them to focus purely on the business logic.



Business Service Architecture

Overview

The business service architecture was designed to provide the maximum flexibility for reuse of the business process. The business logic is completely decoupled from transactional state, the messaging protocol, and operational capabilities. This design allows the protocols, products, and underlying enterprise architecture to change without impacting the business logic performed by the service.

Service Queue

Each business service is associated with a single Java Messaging Service (JMS) queue. A pull-model load balancing approach was implemented to distribute requests to service instances distributed across the compute grid. This approach optimizes a heterogeneous computing environment by allowing services to retrieve requests at a rate determined by the underlying processing capacity of each individual server.

Operational Capabilities

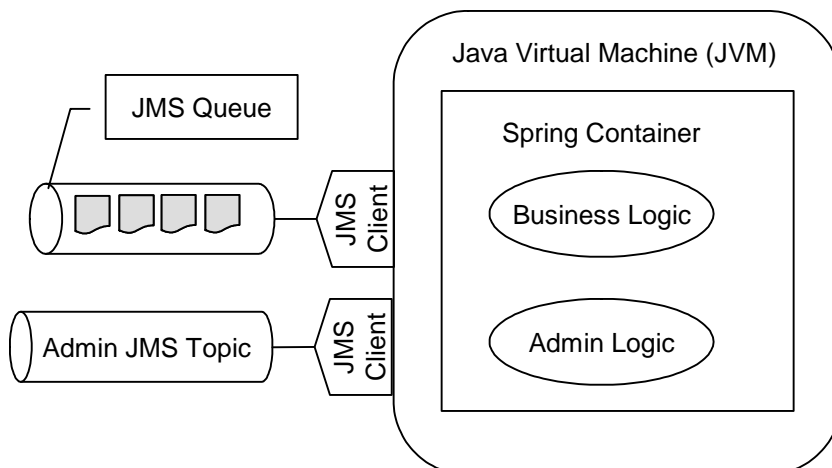
Services are equipped with operational capabilities for runtime management and provisioning. All services listen for administrative requests on a topic. Administrative capabilities include:

- Graceful or abrupt termination of the service
- Abrupt termination of the current service request without termination of the service
- Ping / discovery to determine the health or existence of the service
- Pausing to temporarily stop new requests
- Providing service metrics

These operational capabilities exposed by the services can be used at runtime by the grid management service or by web clients for human monitoring and troubleshooting.

Architecture

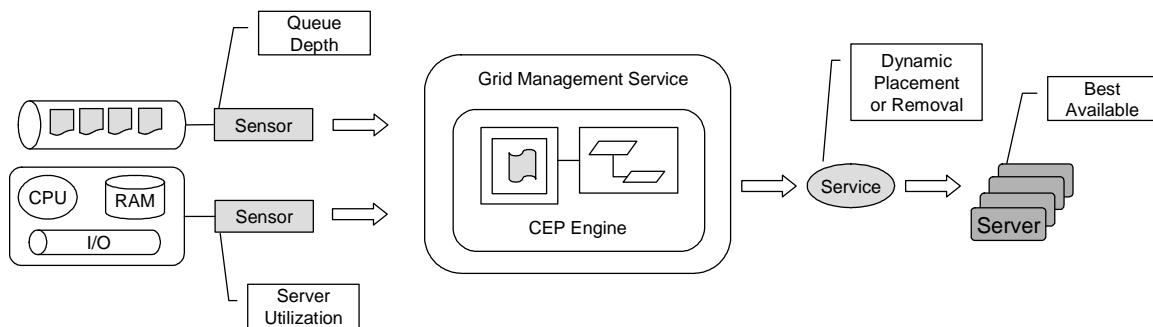
As depicted by the diagram below, each business service exists within a separate Java Virtual Machine (JVM). The Spring Container provides a rich set of container facilities similar to Java Enterprise Edition (JavaEE) application servers.



Grid Management Service

Overview

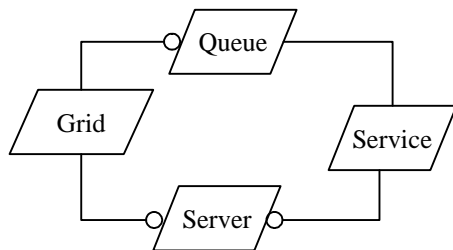
The Grid Management Service (GMS) enables the near real-time optimization of the compute grid. Sensors continuously feed events to the GMS providing both an inventory and the state of every server and queue throughout the enterprise. Current and historical information about both servers and queues are used to perform mathematical calculations, and the results are compared to complex conditions to trigger the dynamic placement and removal of services on the best available computing resource. The diagram below depicts this process in detail.



Ontology

The ontology for the GMS is centered on the concept of a compute grid. The grid ontology was modeled after a series of servers, queues, and services. Services within the grid are associated with many servers but only a single queue. The GMS determines the demand for each service by monitoring the queue depth of each queue. The queue concept maintains a history for the queue depth. The server concept maintains history for CPU, IO and memory utilization.

The history maintained by these concepts is used to calculate statistical averages and rates of change. The service concept maintains static information about the OS, licensing, resource constraints, the queue associated with the service, and the minimum and maximum thresholds for the queue depth.





Complex Processing

The GMS utilizes complex processing to optimize the throughput of the system while minimizing the thrashing that can occur when starting and stopping services on-demand. Thresholds are defined per queue and per server and are stored in a database so that they be manipulated at runtime to tune the system. Complex conditions are rules that compare thresholds against the current and historical state of concepts within the grid. The power of the CEP engine is leveraged when all of the current, historical, statistical and metadata collected from the concepts and numerous events must be correlated and compared against complex conditions to determine the appropriate and best available server to start or stop a service.

Rapid Scalability

The GMS was designed for maximum scalability and provides the capabilities to rapidly respond to increases in processing demand. Servers can be added or removed without a need to shutdown or restart any component. As new servers are added, sensors relay information to the GMS to include the host name of the server. The GMS processes these incoming events by initially verifying that a server concept exists with the host name contained within the event. If the server concept does not exist, the GMS will create an instance, load the configuration, and add it to the grid concept for immediate use in service distribution. Similarly, if a server is shutdown or removed from the grid, the GMS will remove the server concept.

Monitoring and Failover

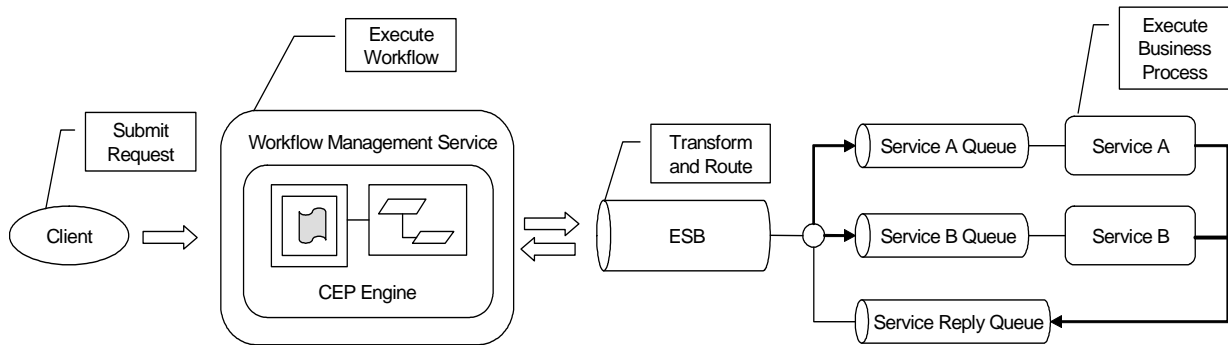
The GMS monitors the health of every service and server within the grid. This is accomplished by periodically publishing a ping event to the administrative topic. Services respond back to the GMS ping event by sending an event with a unique identifier that correlates them to their associated concept instances. Once a reasonable amount of time has expired, the GMS will traverse the list of services and servers to identify any instances that did not respond and remove them from working memory.

Moreover, the GMS will detect services that respond with identifiers that can not be correlated to an existing service concept and add them to working memory. This allows the GMS to discover services that may have been manually started and to recover gracefully from a software or hardware failure. Thus in the event the GMS is shutdown, services can remain deployed and operational, and when the GMS is restarted, the services will automatically be discovered and accounted for.

Workflow Management Service

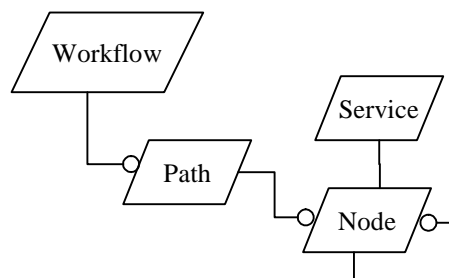
Overview

The Workflow Management Service (WMS) enables a highly configurable customer-driven approach to workflow execution and service orchestration. The WMS was designed to provide the maximum flexibility when modeling workflows regardless of their complexity. A client submits a request to the WMS to initiate a transaction. Once the identity of the customer is determined, the resulting workflow is loaded and executed. The WMS manages the workflow and routes requests to the ESB for delivery to the business services. The services then respond back to the service reply queue where the events received by the WMS are correlated back to the transaction to trigger the next step in the workflow. This high level overview is depicted in the diagram below.



Ontology

A unit of work is represented by the concept of a workflow. A workflow is composed of a series of paths, nodes, and services. The relationships between these concepts are designed to model workflows that support an acyclic digraph data structure. In other words, the model supports parent and child dependencies among services, asynchronous paths of execution, and convergence of paths at prescribed nodes. This ontology allows services to execute in a prescribed order while optimizing the throughput of the transaction. The node concept contains attributes that enable customers to modify parameters and complex subscriptions related to the business services. In addition, the node concept contains a relationship to itself to model the parent child relationships among services. The service concept contains the destination used by the WMS to route the requests.





Complex subscriptions

Services may contain conditional logic that must be evaluated before a request is executed. These conditions are typically based on data produced by the output of other services. For example, a data archive service may only execute if the data to be archived contains a byte length greater than zero. However, in the case of the reference architecture, more complex logic is needed by business services such as conditional (if-then-else) or Boolean (and-or-not). The WMS implements this conditional logic using an EDA pattern referred to as complex subscriptions. The WMS can evaluate complex conditions before a decision is made to send a request to the service.

The decoupling of subscription logic from the service has multiple benefits including greater efficiency during workflow processing and improved utilization of network bandwidth. Furthermore, this approach, coupled with complex processing, allows for dynamic subscriptions based on customer-driven requirements and subsequently increases the reusability of the service.

Complex Processing

The primary responsibility of the WMS is to process workflows and manage transactional state. As transactions arrive within the WMS, workflow processing is immediately started asynchronously to other transactions that may already be running. As services respond to signal completion, the WMS leverages complex processing to correlate the response to both the associated workflow and path concept instances, determine which services should be concurrently executed, evaluate any complex subscriptions, and route the resulting requests to the services. In addition, the WMS must evaluate these conditions across multiple paths to determine if the transaction has completed.

Scalability and Overload Management

Since transactional processing occurs asynchronously, a single WMS instance is capable of managing tens of thousands concurrent transactions. In fact, the reference architecture was designed to support multiple concurrent WMS instances such that scalability would only be limited by external factors such as the network. However, without proper overload management, the same asynchronous behavior can flood the WMS with transactions resulting in a memory overflow.

To prevent an overload scenario from occurring, Complex Event Processing was employed to monitor the number of transaction concepts in working memory. Once a threshold maximum is reached, an imminent overflow condition is detected and the incoming queue is suspended from accepting new transactions. Once the transaction volume diminishes to a minimum threshold, the queue is resumed.

Conclusion

The reference architecture developed by Pyxis Engineering provides a unique solution to enabling scalability in response to a dynamic environment. The loosely coupled design allows the protocols, products, and enterprise architecture to change without impacting the business services. This solution provides a general purpose and reusable framework to solve common computing and workflow requirements found across many organizations.