Monitoring Federated Services in CE Grids

Sekhar Soorianarayanan and Michael Sobolewski
sobol@cs.ttu.edu
SORCER Lab, Texas Tech University,
Lubbock, TX 79409

ABSTRACT: The goal of the Service-ORiented Computing EnviRonment (SORCER) is to form grids of distributed services that provide engineering data, applications and tools on a network. Such environment requires a mechanism for monitoring and debugging of service-oriented programs: tasks (distributed atomic activities) and jobs (aggregations of jobs and tasks) in the service grid and passing of results from the grid environment back to the requestor. The design issues of such a system are described along with its implementation in the SORCER environment.

KEY WORDS: Service-oriented computing, grid computing, concurrent engineering, Jini network technology.

1 INTRODUCTION

SORCER, which stands for Service ORiented Computing EnviRonment, is a service-based concurrent engineering project which is based on evolution of the concepts and lessons learned in the FIPER project [1-8], a $21.5 million program founded by NIST (National Institute of Standards and Technology) [1]. The goal of the SORCER project is to develop an infrastructure for global communication of product information, data, methods, and tools while satisfying stringent product performance requirements. The architecture of the SORCER system is designed to be flexible enough to handle the needs of almost any product from aircraft engines to manufactured goods such as plastics.

The architecture of the SORCER system is service-based, network-centric, and web-centric. This architecture houses the large pool of distributed services that execute business logic and integrate tools and applications in the underlying engineering domain. The web-centric architecture enables HTTP communication between a web-based client and the SORCER system, as well as transparent access to the globally distributed data and the pool of federated services. The individual services requested by the SORCER system act on behalf of the web client, both in the role of providing services (provider mode) and requesting services (requestor mode). When requesting services, the SORCER system also brokers the requests, delegating them to the appropriate registered service providers.

One of the key problems that arise with a distributed system of services and clients is the need for the clients to monitor the execution of exertions [3, 5] in the distributed SORCER environment. An *exertion* is a distributed activity which can be either atomic or compound. In SORCER there are two types of basic exertions defined: tasks and jobs. A *task* is the atomic exertion that is defined by its data (a context model), and by its method. A *job* is the compound exertion that is comprised of tasks and other jobs, so the job is a recursive structure expressed in terms of exertions. While a classic computer program is organized list of statements in a programming language; a job's exertions can be treated as distributed statements that tell the SORCER environment what to do with task context models [3-5]. Execution of a job might require many service providers. Once the job starts executing, the exertions in a job get bound to relevant providers that are determined in runtime. This dynamic collection of bound providers is called a federation. Also, it might take many hours or days to execute a job. Due to these reasons, it becomes very necessary to monitor, control and debug the execution of the jobs. It's also necessary to have a recovery mechanism through which, a user can correct and resume from a failure point of a misbehaved exertion. This results in higher tolerance level of any error in service-oriented programming.

2 SYSTEM ARCHITECTURE OVERVIEW

Building on the object-oriented paradigm is the *service-oriented* paradigm, in which the objects are distributed, or more precisely they are *network objects* and play some predefined roles. A *service* provider is an object that accepts messages from *service requestors* to execute an item of work – a *task*. The task object is a service request – a kind of elementary grid operation executed by a service provider. A service *jobber* is a specialized service provider (broker) that executes a job – a compound request in terms of tasks and other jobs. The job object is a *service-oriented program* that is dynamically bound to all relevant and currently available service providers on the grid. This collection of grid providers dynamically identified by a jobber is called a *job federation*. This runtime network or *grid federation* is the jobs' execution environment and the *job object* is a service-oriented program. In other words, we apply the object-oriented concepts directly to the grid in the service-oriented paradigm. Tasks and jobs as grid programs are called *exertions*. A task is the *atomic exertion* that is defined by its *context model* (data), and by its *method* (a pair: interface and selector). A *service context* is the basic data structure based on the percept-calculus knowledge representation scheme [9]. A *context model* is a tree like structure of data being processed. Each path of a tree names a leave node where the data resides. An exertion method defines a service provider (grid object) to be bound to in runtime. This network object provides the business logic to be applied to the exertion context model. The computing framework based on concepts: context model, method and exertion is called for short the CME *framework* (see Figure1 for details).

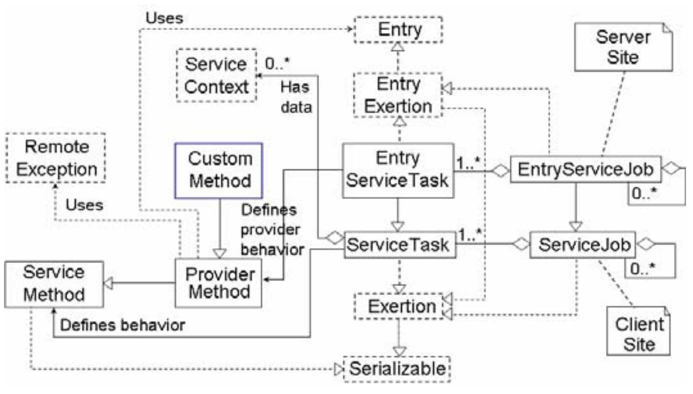


Figure 1 Context-Method-Exertion Paradigm

The method is primarily defined by a *provider type* (interface) and *selector* (method name) in the provider's interface. The service method may also refer to a piece of code to be downloaded and executed by a provider (mobile code). Optionally, additional attributes might be associated with the method, for example a provider's name or provider's identifier. The information included in the exertion method allows the SORCER program to bind the exertion to the network object and process the exertion's context by one of its peer's operations, which is defined by its published interface. This type of service provider is called a *method provider*. Another type of service provider is a *context provider* that provides shared data to the grid via the observer-observable design pattern [12]. Thus, both context and method providers represent grid data and operations to be used in the grid-oriented programs. A context model in jobs is called a control context since it defines the control strategy for the execution of a job's exertions. All the component task contexts of a job constitute a combined context called a *job context*.

A job usually is created in the GUI interactively as a service-oriented program. Each "instruction" in that program is represented by the exertion in it. All dependencies between the data shared between exertions are captured with metaattributes in exertion context models.

2.1 *Exertion Execution*

All services in the SORCER environment are peers. They implement the global interface called *Servicer*. A service is an act of invoking a method of following signature on a service provider:

```
Exertion service(Exertion)
```

All service providers in SORCER extend the `ServiceProvider`, class which is a system level abstract class, implementing `Servicer`. The default implementation of the service method in `ServiceProvider` looks into the service method of the exertion and checks if the service provider itself implements the interface defined in exertion. If it does, then via reflection the method defined by the selector is invoked with the exertion's service context as the argument. Hence all providers can implement any number of custom interfaces with methods accepting `ServiceContext` as argument. If the interface defined by the service method is not implemented by the service provider, the called provider dynamically finds the right provider with the aid of object registry and forwards its exertion to the right peer in the network to be bound for execution.

The clear separation of method, data and control strategy, and having the global `Servicer` interface helps us to build dynamic federation of services in the service grid. Also, no service requires any prior knowledge of another service and there's no hassle of finding on-the-fly a provider for a particular interface in the grid. All service providers in SORCER act as true service peers.

2.2 *Job Execution*

Specialized service providers that are called *Jobbers* perform Job execution. A `Jobber` coordinates execution of a job using the job context model called a *control context*. The control context defines a job's execution strategy. A strategy implements a master/slave-computing model [13] with sequential or parallel execution of slave exertions with the master exertion executed as the last one.

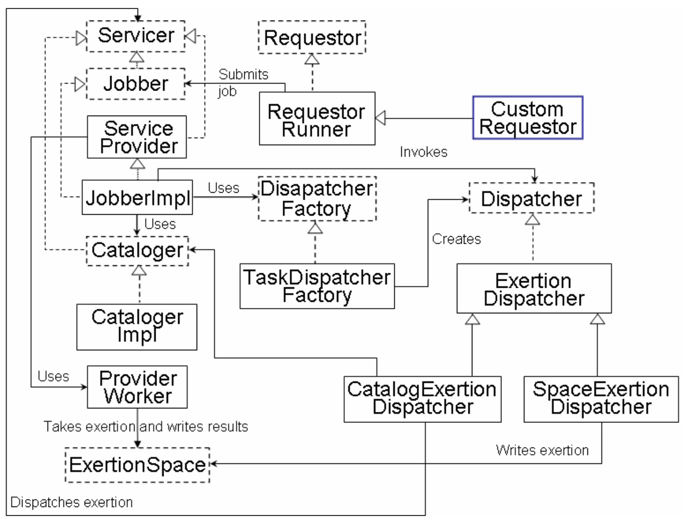


Figure 2 Job execution in sorcer

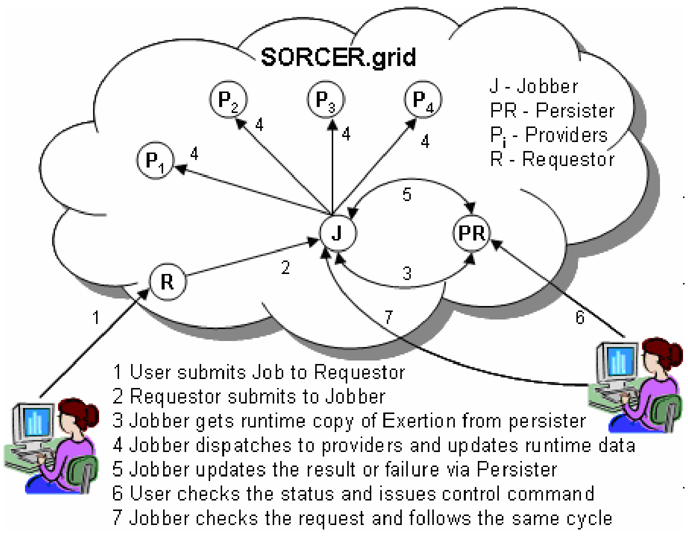
Jobbers use dispatchers to dispatch component exertions to the right providers in the grid. In SORCER there are four types of dispatchers that implement different type for control strategies. These include sequential and parallel dispatchers for Catalog and Space. A relevant dispatcher is assigned to a jobber by the dispatcher factory based on the job's control context. The UML diagram for job execution is presented in Figure-2.

3 CONCEPTUAL ARCHITECTURE

During the bootstrap stage of a service provider, the provider receives a unique identifier called *providerID*, from the object registry that keeps a catalog of all services in the local grid. This *providerID* is globally unique and independent to location or platform of the service.

When the jobber receives a request to execute a job, it creates a runtime copy of the job, and then it persists in a data store. The jobber now starts dispatching the runtime exertions to the right providers in the grid. Before dispatching to the service providers, the jobber sets the providerID in the dispatched runtime exertions of the runtime job and updates the runtime database accordingly.

Each runtime exertion has a process state associated with it. All exertion process states are maintained by the Jobber, which updates them in the persistent store. This helps us to determine the state of distributed activities from one central location. Also, this scheme allows the requestor to send remote commands to control the execution of its running job.



1 User submits Job to Requestor
2 Requestor submits to Jobber
3 Jobber gets runtime copy of Exertion from persister
4 Jobber dispatches to providers and updates runtime data
5 Jobber updates the result or failure via Persister
6 User checks the status and issues control command
7 Jobber checks the request and follows the same cycle

Figure 3 Conceptual architecture

Any requestor can monitor its exertions using the persisted information regarding the corresponding runtime exertions. Any request to suspend/stop/resume can now go to the right provider with the aid of the providerID set inside each runtime exertion. When the provider receives a request to alter the state of the running exertion, it checks a set of rules and alters the state of the runtime exertion and updates the runtime database. When a monitoring call reaches the jobber, it propagates the request to the right providers, as it keeps track of which provider is executing what part of the distributed activity of the job.

Once an exertion is suspended or failed, the user would get to know the cause of the failure/error from the persistent store of monitoring system and then the user can make adequate corrections to the exertion by debugging the underlying problem. Once the corrections are made, the user can resubmit the exertion by resuming the exertion from the failure point and continue monitoring the distributed execution again. Thus any failure can be inspected, debugged, and recovered. Figure-3 illustrates the behavior of the monitoring system.

4 IMPLEMENTATION OF THE MONITORING SYSTEM

SORCER treats monitoring also as a service, which is provided by all the peers in the federation. A `Monitor` interface is presented in Figure-4. All providers in the system extend the `ServiceProvider` class and `ServiceProvider` implements both `Servicer` and `Monitorable` interfaces.

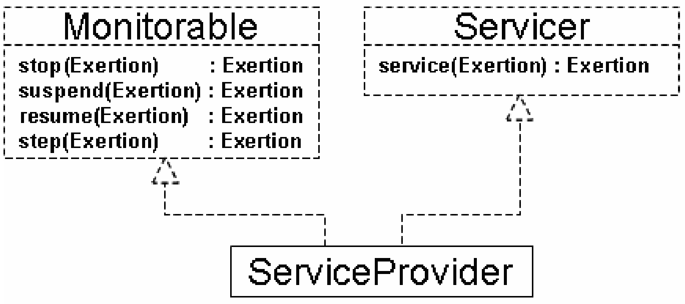

Monitorable
stop(Exertion) : Exertion
suspend(Exertion) : Exertion
resume(Exertion) : Exertion
step(Exertion) : Exertion
Servicer
service(Exertion) : Exertion
ServiceProvider

Figure 4 All providers are Monitorable

As mentioned above, each provider has its own unique identifier. This unique identifier is the *serviceID* provided by the *Jini lookup service* (object registry) [10, 11] when a provider joins the SORCER grid. Peers forward a service-oriented program (exertion) submitted by a requestor to the right provider. Each peer would check if the service requested matches the service provided by it. If not, it forwards to the right provider defined by the exertion's method.

A runtime data store is maintained for all exertions being executed in the grid. This way the user can monitor all the runtime jobs from the data store and can keep track of the distributed activities. The provider while executing a task of job may update the results intermittently in the runtime data sore. The user may also check the results periodically. All runtime interactions with the data store are maintained by a specialized persisting service called the *Persister*.

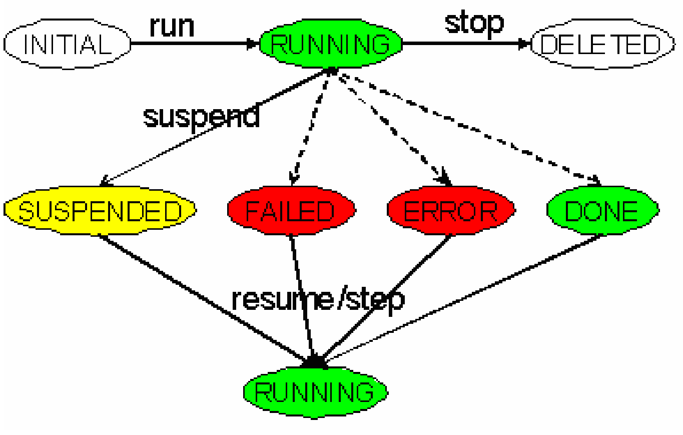


Figure 5 Exertion States

When a provider receives an exertion for execution, it initially does the following
1) Sets its providerID in the exertion.
2) Then it checks if the exertion is already a runtime exertion. If so it updates the exertion. Otherwise, via the persister service, it creates a new runtime exertion for the exertion it's about to execute.

All providers are remote objects and they store information about the current running exertions and their statuses in different formats. All service providers maintain a static map of the currently executing exertionIDs and the statuses of the corresponding exertions. This is called the *Exertion Status Map* (ESM). A jobber is a specialized provider that specializes in execution and coordination of jobs. Jobbers spawn a separate dispatcher thread for the execution and management of a runtime job. Each Jobber maintains a static map of the currently executing runtime exertionIDs and the corresponding dispatcher's threads that it executes. This map is called the *Dispatcher Thread Map* (DTM). Every dispatcher before executing the individual exertion of the job sets the providerID of the provider to which it dispatches the individual exertion of the job to and updates accordingly the runtime data store. Once the exertion is dispatched to the individual providers, it is the responsibility of the providers to maintain the ESM and intermittently update the current state of the data of the running exertion in the data store. This way, the user can gather useful information about the running job and its distributed components.

Any exertion that starts executing on the individual provider side may complete successfully or fail. An exertion might also fail due to some input error, in which case, its state is marked as ERROR. Once the exertion is completed successfully or fails, the providers return to the Jobber, which marks the running exertion as DONE/FAILED/ERROR and saves it in the data store. In the case of any failure or error, the jobber marks the exertion as FAILED and waits for the other dispatched exertions to finish. Once finished, the dispatcher would mark the runtime job as FAILED/ERROR and persist it in the data store. The state diagram of exertion execution is depicted in Figure 5.

All providers capture not only the runtime results in the exertion context model, but also they can capture any exceptions or errors that are encountered during the execution. In case of any FAILURE/ERROR of the executing exertion, the message associated and the exception is added to the context model of the exertion. This way the causes of failure or exceptions are carried back to the user via the context model. Once the exertion encounters any exception, the providers remove the entry from DTM and ESM and return the exertion to the calling method. The client receives notification through the Notification Manager [4]. The user can see the runtime job from the Job Monitor (web-based user agent) and check the status of job, component exertions, results of still running exertions, any exceptions encountered while running etc. Using the Job Editor (user agent), the user can make necessary modifications/corrections on a FAILED/SUSPENDED runtime job and rerun the job from any point. This is the central idea of the debugging capability of the job monitoring framework.

One of the main features of the Job Monitor is that users can issue commands to suspend/stop a running job and resume a suspended job. When the client issues a command to suspend/stop a job, with the help of the providerID saved in the job, the request is forwarded to the right Jobber. Once the request reaches the jobber, the jobber pulls out the right DTM entry and issues command to suspend/stop to the dispatcher thread. The Dispatcher thread now issues command to all the running exertions inside the dispatcher to stop/suspend, again with the help of the providerID which is set inside the running exertions. Once the command reaches the respective providers, each provider now has a chance to cleanup and marks the exertion as SUPENDED/STOPED and returns the exertion with the current state of data which is preserved inside the context. If the provider is not smart enough to maintain the state of the atomic activities like task, then

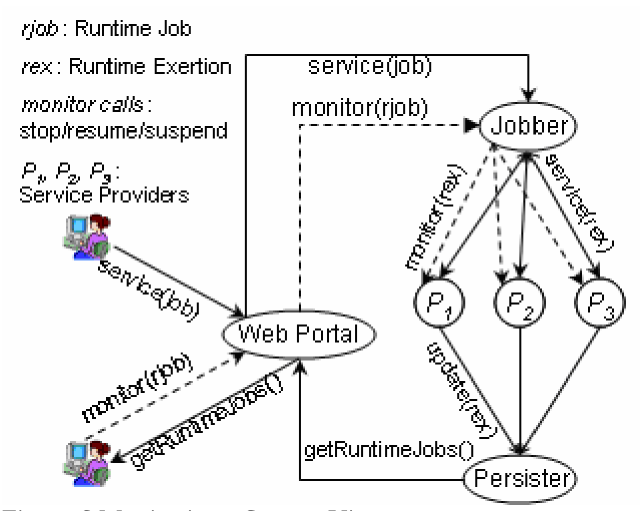


Figure 6 Monitoring - System View

the provider can choose to return back the initial state of the context model. Once the jobber receives all the exertions from all federated providers, it marks the job as SUSPENDED/STOPPED and persists in the data store. If the user has requested to stop the running job instead of suspend, then the corresponding runtime job is removed from the data store. (see Figure 6)

A FAILED/SUSPENDED job can be resumed or stepped by the user. Once requested to resume, it's the same as the execution of a job from INITIAL state, except for the following differences: 1) A new runtime copy is not made, as the job is already a runtime job 2) The job can be resumed from any previous point the user specifies. If the user does not specify anything, the job is resumed from the suspended or failed point. If the user chooses to step a job instead of resuming/running, then the exertions inside the job are run step by step. After each step, the job gets suspended automatically and the notification is sent to the user to review and resume the job.

The user has the option of setting a review flag for a component exertion at any point in the job before the job is submitted for execution. The jobber automatically would suspend the execution of the job at that point and wait for the review from the user. The user can review the execution and then resume the job from that point. Other than setting review on exertions, the user can also mention the point from which any exertion is to rerun and mention if any exertion needs to be skipped. All these functionalities would be handled by the Jobber service.

5 IMPLEMENTATION OF THE USER AGENT

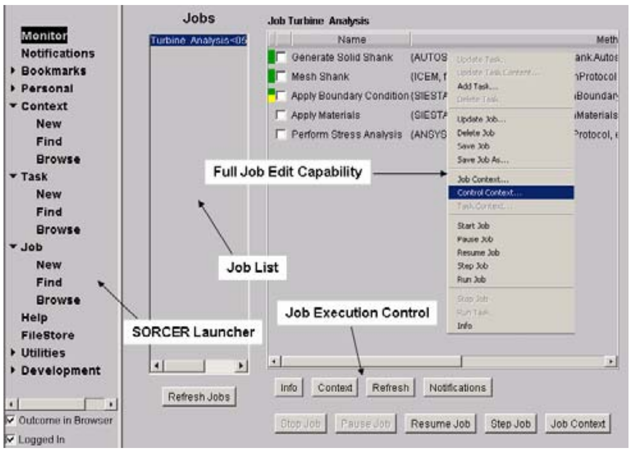


Figure 7 Monitoring GUI

The SORCER Launcher is the main menu for all of the GUI tools that are needed to build contexts, tasks, jobs, and run and monitor SORCER jobs. Once a user has been authenticated the launcher menu is displayed in the left-hand frame of the user agent (Figure-7) and the right hand frame is initially empty. The user is then free to begin accessing the environment. The user selects the Monitor item from the launcher menu to get the Monitor GUI, to the right of the launcher as shown in Figure 7.

The Job Monitor displays a list of jobs submitted by the user and now the user is authorized to monitor all his/her runtime jobs persisted in the data store. On selection of any runtime job in the list, the job worksheet displays multiple rows, each row representing a component exertion of the selected job. Every exertion of the Job in the GUI has a colored indicator which shows the current status of the selected job. Any SUSPENDED, FAILED, or DONE job could be edited and resubmitted for execution. Also the job control-context can be edited to change the execution strategy of the job (see Figure 8). The job context can also be viewed for the job and all the information regarding exceptions can be seen in the Job Context user agent. Once the cause of the failure is found out, problems can be fixed and the job can be resubmitted for execution. Figure-7 shows the editing capabilities of Job Monitor.

The monitor worksheet has a check box corresponding to every exertion in the Job. This is used to mark the point from which a FAILED/SUSPENDED job is to be executed. If no check box is checked in, the job is resumed from the point where it failed or was suspended. If the job is a sequential, only one check box can be selected. If the Job is parallel, multiple check boxes can be selected. The Job Monitor inherits all the functionality of a Job-Editor user agent. In the worksheet the three columns indicate the name of the exertion, the type of service requested and the description of the exertion. The provider column is the name of the SORCER service provider that was executing a given exertion. The info button in the popup menu gives additional information like the time of execution, when the job started to run, when it completed, which host it ran, etc. In the Job Editor, the exertions can be edited and changes saved when the job is not running.

The control-context shown in Figure-8 is used to define the execution strategy of a job. A job can be marked for execution as sequential or parallel; can build a federation using catalog, exertion space, or direct access to object registries. Also some exertions can be marked as skipped in the job. The dispatcher associated with the job would disregard those marked as skipped.

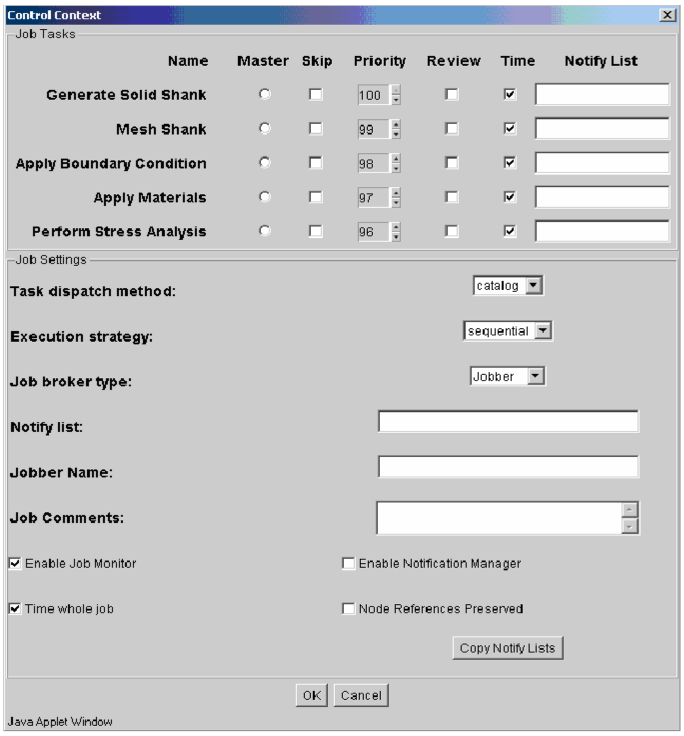
When the job is running, Job Monitor starts a monitoring thread that refreshes the job worksheet for every 30 seconds. This way the user can view the current status in the worksheet. The user can also refresh manually the jobs by clicking on the refresh button shown in the GUI.

The Notification button would bring up the Notification GUI which would display all the notifications submitted by the federated providers for the selected job. Thus the client can see the status and notifications for the job at the same time.

The Pause and Stop buttons are selectable only for jobs that are running. Pause would suspend the job. The client can choose to suspend the job at any particular point once the job starts executing. Clicking Pause would send the message to the providers to suspend the execution of the job. The providers would suspend the job and return the suspended job to the user. The Stop button would completely halt the job and delete the job from the Job Monitor as well as from the runtime data store

The Resume and Step buttons are selectable only when the job is in SUSPENDED/FAILED state. The reason for this is that any job which is currently running can not be resumed or stepped. Resume would resume from the point at which the job was suspended or failed unless any check box is selected in the Monitor Worksheet indicating from what point to start. The Step button would flag all component exertions to be reviewed. This will cause the job to suspend after execution of every exertion in the job. All interactions with the SORCER.grid are made by the user agent via an application servlet which is run in by a SORCER portal.

The Resume and Step buttons are selectable only when the job is in SUSPENDED/FAILED state. The reason for this is that any job which is currently running can not be resumed or stepped. Resume would resume from the point at which the job was suspended or failed unless any check box is selected in the Monitor Worksheet indicating from what point to start. The Step button would flag all component exertions to be reviewed. This will cause the job to suspend after execution of every exertion in the job. All interactions with the SORCER.grid are made by the user agent via an application servlet which is run in by a SORCER portal.


Control Context

Job Tasks

Name	Master	Skip	Priority	Review	Time	Notify List
Generate Solid Shank	○	☐	100	☐	☑	
Mesh Shank	○	☐	99	☐	☑	
Apply Boundary Condition	○	☐	98	☐	☑	
Apply Materials	○	☐	97	☐	☑	
Perform Stress Analysis	○	☐	96	☐	☑	

Job Settings

Task dispatch method:	catalog
Execution strategy:	sequential
Job broker type:	Jobber
Notify list:	
Jobber Name:	
Job Comments:	

☑ Enable Job Monitor ☐ Enable Notification Manager

☑ Time whole job ☐ Node References Preserved

Figure 8 Control Context

6 CONCLUSION

The paper describes a versatile architecture to support monitoring CE grids. The SORCER monitoring architecture uses ubiquitous services that reside on the network and federate dynamically to map into a specific engineering process. The services have standardized interfaces allowing one service to be seamlessly replaced by another service in the CE grid. The service based computing model requires registries where services can register themselves and allow them to be discovered in real time for a federation. The conceptual model is the CME generic paradigm. This federated approach reduces the brittleness in existing systems that break when the processes and tasks in the processes evolve over time. Several applications have been created in the domain of engineering at GE Aircraft Engines to demonstrate the concepts of this architecture. The federated service-oriented programming has reduced the design cycle time for preliminary aerodynamic analysis at GE by a factor of 10. It has also reduced the nozzle combustor development time from a few days to a few minutes. The service monitoring solutions that have been developed as a part of this work proved to be user friendly and efficient to monitor and debug CE job federations.

REFERENCES

[1] The Federated Intelligent Product Environment (FIPER) – Project Brief (1999), Available at http://jazz.nist.gov/atpcf/prjbriefs/prjbrief.cfm?ProjectNumber=99-01-3079

[2] Sanjay G., Sobolewski M., 2003, Trust and Security in Enterprise Grid Computing Environment, Proceedings of the IASTED Intl., Conference on Communication, Network, and Information Security, Dec 10-12, 2003, New York, NY.

[3] Sobolewski M., Soorianarayanan S, Malladi-Venkata R-K., 2003, Service-Oriented File Sharing, Proceedings of the IASTED Intl., Conference on Communications, Internet, and Information technology, pp. 633-639, Nov 17-19, 2003, Scottsdale, AZ.

[4] Lapinski M., Sobolewski M., 2002, Managing Notifications in a Federated S2S Environment, International Journal of Concurrent Engineering: Research & Applications, Dec 2002.

[5] Sobolewski, 2002. Federated P2P Services in CE Environments, *Advances in Concurrent Engineering, A.A. Balkema Publishers, 2002*, ISBN 90 5809 502 9, pp. 13-22.

[6] Sobolewski, 2002. FIPER: The Federated S2S Environment, *JavaOne, Sun's 2002 Worldwide Java Developer Conference*, (http://servlet.java.sun.com/javaone/sf2002/conf/sessions/display-2420.en.jsp).

[7] Zhao, Shuo, and Michael Sobolewski, 2001, Context Model Sharing in the FIPER Environment, *Proc. of the 8th Int. Conference on Concurrent Engineering: Research and Applications*, Anaheim, CA.

[8] Röhl, P.J. & Kolonay, R.M., et al. (2000). Federated Intelligent Product Environment, American Institute of Aeronautics and Astronautics Inc.

[9] Sobolewski, M. 1991. Percept Conceptualizations and Their Knowledge Representation Schemes. Z.W. Ras and M. Zemankova (Eds.) *Methodologies for Intelligent Systems, Lecture Notes in AI 542*, Berlin: Springe-Verlag, pp. 236-245.

[10] Edwards, W.K. (2000). Core Jini, 2nd ed., Prentice Hall, ISBN: 0-13-089408.

[11] Jini Architecture Specification. Available at URL: http://www.sun.com/jini/specs/jini1_1.pdf.

[12] Freeman, E., Hopfer, S., & Arnold, K.(1999), Javaspaces™ Principles, Patterns, and Practice, Addison-Wesley, ISBN: 0-201-30955-6.

[13] Grand, M. (1999). Patterns in Java, Volume 1, Wiley, ISBN: 0-471-25841-5