Federated Collaborations with Exertions

Michael Sobolewski
Computer Science, Texas Tech University
sobol@cs.ttu.edu

Abstract

This paper describes a service-oriented P2P architecture and related federated metaprogramming model to support development of highly scalable and reliable distributed collaborative applications. In the proposed architecture, autonomic service providers, corresponding to various activities that occur in the collaborative process, reside on the overlay network and are discovered dynamically during the execution of the process. To execute a specific collaboration, a set of services that map into the collaboration specification (exertion) are federated together and executed in a choreographed workflow. All services (peers) implement a standardized top-level interface and this allows any service to be seamlessly replaced with another service without affecting the performance of the federation. The paper describes a service object-oriented environment (SORCER) and presents how it supports programming of three collaboration types.

1. Introduction

Design of complex engineering systems, such as aircraft engines, requires simulation tools that are computationally expensive. Such systems are composed of hundreds of mutually interacting components that should ideally be designed concurrently. Concurrent design of all components, however, is computationally and logistically infeasible. So, the design is usually decomposed into smaller *activities* such that components and subsystems are designed independently. As the design progressively gets decomposed into increasingly smaller activities, the process map progressively defines a *collaboration*—a process by which relevant activities work together to accomplish a common endeavor. An *exertion* is the specification of a process map that defines how collaboration is realized by set of service providers and their associations playing specific roles in specific activities. Thus, an exertion represents, for example, the design process that needs to be executed in order to complete the engineering design. The design process is dynamic and evolves over time as new advances are made in technology and analysis techniques.

In this paper, we present the most recent version of exertion-oriented programming [7] that has evolved over several years from the initial form in the FIPER project [1] and enhanced continuously on multiple projects at the SORCER Laboratory [8]. Several attempts have been made in the past to create a collaborative design environment through the automation of activities in the design process, and consequently reduce the design cycle time and improve performance. These automation efforts, though robust at the individual activity level where the process is fairly standardized are very brittle at the process level. Brittleness here refers to inflexibility and inability to adapt to changes. The reason for brittleness at the process level is that analysis codes, as well as the process, change and render the couplings between activities ineffectively thereby breaking the process map.

In this paper we describe a system architecture that supports an adaptive collaborative environment through use of modular services in a federated service-to-service (S2S) framework. The architecture constitutes of well-known autonomic services that represent specific activities on the service grid. The service grid is an overlay network of service providers above the underlying network of computing devices. The services have standardized top-level and domain-specific types (interfaces) allowing them to be located by searching for complementary attributes associated with types. The standardization of interfaces also ensures that one service can be seamlessly replaced by another service without requiring reconfiguration of the overlay network. The services have standardized interfaces and data formats; strong coupling by cascading data from one activity to the next is unnecessary. Once the collaboration defined by an exertion is complete, the services disperse and join other federations to perform other activities. Changes to any individual service on the grid are usually transparent to the exertion. In this architecture, services can enter and leave the service grid at will. Resilience in the service grid is achieved due to the redundancy in the overlay network whereby several services can exist for the same activity. The standardized interfaces allow seamless substitution of one service with another.

The paper presents a service object-orient architecture and programming paradigm for managing collaborative processes and hypothesizes that such service providers can facilitate modeling of complex business processes and provide greater flexibility in managing process changes and improved resilience to system failures. Architectural qualities like flexibility, scalability, and reliability of this architecture were demonstrated in multiple applications and case studies in concurrent engineering [1, 2, 6, 9]. In this paper we describe the version of programming collaborations with exertions developed at the SOCER Laboratory [8].

The rest of the paper is organized as follows: Section 2 describes the architecture of the SORCER system, Section 3 describes exertion-oriented programming model, Section 4 is focused on service messaging with exertions, Section, 5 presents push and pull collaborations, and Section 6 provides concluding remarks.

2. SORCER

SORCER is a federated service-to-service (S2S) metacomputing environment that treats service providers as network objects with well-defined semantics of a federated service object-oriented architecture. It is based on Jini semantics of services in the network and Jini programming model with explicit leases, distributed events, transactions, and discovery/join protocols [5]. While Jini focuses on service management in a networked environment, SORCR is focused on exertion-oriented programming and the execution environment for exertions. SORCER uses Jini discovery/join protocols to implement its *exertion-oriented architecture* (EOA) using *federated method invocation* (FMI) [7], but hides all low-level programming details of the Jini programming model.

In EOA, a service provider is an object that accepts remote messages from service requestors to execute a collaboration. These messages are called service exertions that describe *service data, operations* and provider's *control strategy.* An *exertion task* (or simply a *task*) is an elementary service request, a kind of elementary remote instruction executed by a single service provider or a small-scale federation for the same service data. A composite exertion called an *exertion job* (or simply a *job*) is defined hierarchically in terms of tasks and other jobs, a kind of network procedure executed by a large-scale federation. The executing exertion is dynamically bound to all required and currently available service providers on the network. This collection of providers identified in runtime is called an *exertion federation*. The federation provides the implementation for the collaboration as specified by its exertion. When the federation is formed, then each exertion's operation has its corresponding method (code) on the network available. Thus, the network *exerts* the collaboration with the help of the dynamically formed service federation. In other words we send the request onto the network implicitly, not to a particular service provider explicitly.

The overlay network of service providers is called the service grid and an exertion federation is in fact a virtual metacomputer. The metainstruction set of the metacomputer consists of all operations offered by all service providers in the grid. Thus, an exertion-oriented (EO) program is composed of *metainstructions* with its own *control strategy* and a *service context* representing the metaprogram data. The service context describes the data that tasks and jobs work on. Each service provider offers services to other service peers on the object-oriented overlay network. These services are exposed *indirectly* by operations in well-known public remote interfaces and considered as elementary (tasks) or compound activities (jobs) in EOA. Indirectly means here, that you cannot invoke any operation defined in provider's interface directly. These operations can be specified in a requestor's exertion only, and the exertion can be passed on to any service provider via the top-level `Servicer` interface implemented by all service providers called *servicers*—service peers. Thus all service providers in EOA implement the `service(Exertion, Transaction):Exertion` operation of the `Servicer` interface. When the servicer accepts its received exertion, then the exertion's operations can be invoked by the servicer itself, if the requestor is authorized to do so. Servicers do not have mutual associations prior to the execution of an exertion; they come together dynamically (federate) for a collaboration as defined by its exertion. In EOA requestors do not have to lookup for any network provider at all, they can submit an exertion, onto the network by calling `Exertion.exert(Transaction):Exertion` on the exertion. The `exert` operation will create a required federation autonomically that will run the collaboration as specified in the EO program and return the resulting exertion back the exerting requestor. Since an exertion encapsulates everything needed (data, operations, and control strategy) for the collaboration, all results of the execution can be found in the returned exertion's service contexts.

Domain specific servicers within the federation, or task peers (*taskers*), execute task exertions. *Rendezvous* peers (jobbers and spacers) coordinate execution of job exertions. Providers of the `Tasker`, `Jobber`, and `Spacer` type are three of SORCER main infrastructure servicers, see Figure 1. In view of the P2P architecture defined by the `Servicer` interface, a job can be sent to any servicer. A peer that is not a `Jobber` type is responsible for forwarding the job to one of available *rendezvous* peers in the SORCER environment and returning results to the requestor.

Thus implicitly, any peer can handle any job or task. Once the exertion execution is complete, the federation dissolves and the providers disperse to seek other collaborations to join. Also, SORCER supports a traditional approach to grid computing similar to those found for example in Condor [10]. Here, instead of exertions being executed by services providing business logic for invoked exertions, the business logic comes from the service requestor's executable programs that seek compute resources on the network.

Grid-based services in the SORCER environment include `Grider` services collaborating with `Jobber` and `Spacer` services for traditional grid job submission. `Caller` and `Methoder` services are used for task execution. `Callers` execute conventional programs via a system call as described in the service context of submitted task. `Methoders` can download required Java code (task method) from requestors to process any submitted context accordingly with the code downloaded. In either case, the business logic comes from requestors; it is a conventional executable code invoked by `Callers` with the standardized `Caller`'s service context, or mobile Java code executed by `Methoders` with a matching service context provided by the requestor.

3. Exertion-oriented Programming

Each programming language provides a specific computing abstraction. Procedural languages are abstractions of assembly languages. Object-oriented languages abstract entities in the problem domain that refer to "objects", communicating via message passing, as their representation in the corresponding solution domain. However, we cannot just take an object-oriented program developed without distribution in mind and make it a distributed system, ignoring the unpredictable network behavior. The EO programming is a form of object-oriented distributed programming that allows us to describe the distributed problem in terms of the intrinsic unpredictable network domain instead of in terms of distributed objects hiding the notion of the network domain that in reality cannot be hidden.

What intrinsic distributed abstractions are defined in SORCER? Well, service providers are "objects", but they are specific objects—they are network objects with a network state, network behavior, and network types. Service providers act also as network peers (servicers)*;* they are replicated and dynamically provisioned for reliability to compensate for network failures. Servicers can be found in runtime transparently by types they implement. They can federate for an exertion submitted onto the network and participate in the collaboration outlined by the exertion. The exertion encapsulates service *data*, *operations*, and *control strategy* used by the collaboration. The component exertions may need to share context data of ancestor exertions, and the top-level exertion is complete only if all nested exertions are successful. Thus, a collaboration is a *process*, an exertion is the *specification* of collaboration, and a dynamic federation of peers is the *implementation* of collaboration,

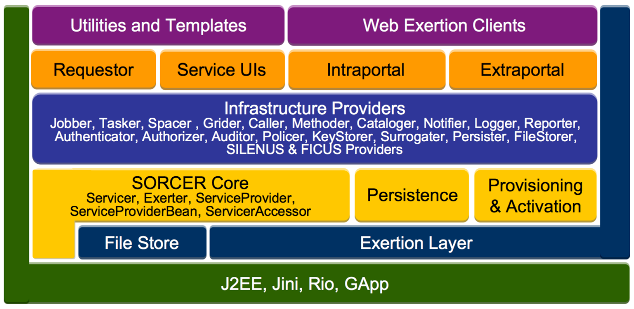
Utilities and Templates
Web Exertion Clients
Requestor
Service UIs
Intraportal
Extraportal
Infrastructure Providers
Jobber, Tasker, Spacer , Grider, Caller, Methoder, Cataloger, Notifier, Logger, Reporter, Authenticator, Authorizer, Auditor, Policer, KeyStorer, Surrogater, Persister, FileStorer, SILENUS & FICUS Providers
SORCER Core
Servicer, Exerter, ServiceProvider, ServiceProviderBean, ServicerAccessor
Persistence
Provisioning & Activation
File Store
Exertion Layer
J2EE, Jini, Rio, GApp

Figure 1. The SORCER layered functional architecture.

With that very concise introduction to the abstractions of EO programming let's look into a simple analogy to a Unix shell script execution and then in detail at how network-centric messaging is defined in EOA.

Let's first look at the EO approach to see how it works. Exertion-oriented programs consist of *exertion* objects called tasks and jobs. An exertion *task* corresponds to an individual network request to be executed on a service provider. An exertion *job* consists of a structured collection of tasks and other jobs. The data upon which to execute a task or job is called a s*ervice context*. Tasks are analogous to executing a single program or command on a computer, and the service context would be the input and output streams that the program or command uses. A job is analogous to a batch script that can contain various commands and calls to other scripts. Pipelining Unix commands allows us to perform complex activities without writing complex programs. As an example, consider a script `sort.sh` connecting simple processes in a pipeline as follows:

```
cat hello.txt | sort | uniq > bye.txt
```

The script is similar to an exertion job in that it consists of individual tasks that are organized in a particular fashion. Also, other scripts can call the script `sort.sh`. An exertion job can consist of tasks and other jobs, much like a script can contain calls to commands and other scripts.

Each of the individual commands, such as `cat`, `sort`, and `uniq`, would be analogous to a task. Each task works with a particular service context. The input context for the `cat` "task" would be the file `hello.txt`, and the "task" would return an output context consisting of the contents of `hello.txt`. This output context can then be used as the input context for another task, namely the `sort` command. Again the output context for `sort` could be used as the input context for the `uniq` task, which would in turn give an output service context in the form of `bye.txt`.

To further clarify what an exertion is, an exertion consists mainly of three parts: a set of *service signatures*, which is a description of operations in collaboration, the associated *service context* upon which to execute the exertion, and control strategy (default provided) that defines how signatures are applied in the collaboration. A *service signature* specifies at least the provider's interface that the service requestor would like to use and a selected operation to run within that interface. There are four types of signatures that can be used for an exertion: `PREPROCESS`, `PROCESS`, `POSTPROCESS`, and `APPEND`. An exertion must have one and only one `PROCESS` signature that specifies what the exertion should do and who works on it. An exertion can optionally have multiple `PREPROCESS`, `POSTPROCESS`, and `APPEND` signatures that are primarily used for formatting the data within the associated service context. A *service context* consists of several data nodes used for either input, output, or both. A task may work with only a single service context, while a job may work with multiple service contexts since it can contain multiple tasks. The programmer can define a control strategy as needed for the underlying exertion by choosing relevant exertion types and configuring attributes of service signatures and service contexts accordingly [7].

Here's the basic structure of the EO program that is analogous to the `sort.sh` script.

```
1.  // Create service signatures
2.  Signature catSignature, sortSignature,
3.     uniqSiganture;
4.  catSignature =
5.  new ServiceSignature("Reader", "cat");
6.  sortSignature =
7.  new ServiceSignature("Sorter", "sort");
8.  uniqSiganture =
9.  new ServiceSignature("Filter", "uniq");
10.
11. // Create component exertions
12. Task catTask, sortTask, uniqTask;
13. catTask =
14. new ServiceTask("cat", catSignature);
15. sortTask =
16. new ServiceTask("sort", sortSignature);
17. uniqTask =
18. new ServiceTask("uniq", uniqSiganture);
19.
20. // Create top-level exertion
21. Job sortJob =
22. new ServiceJob("main-sort");
23. sortJob.addExertion(catTask);
24. sortJob.addExertion(sortTask);
25. sortJob.addExertion(uniqTask);
26.
27. // Create service contexts
28. Context catContext, sortContext,
29.    uniqContext;
30. catContext =
31. new ServiceContext("cat");
32. sortContext =
33. new ServiceContext("sort");
34. uniqContext =
35. new ServiceContext("uniq");
36.
37. catContext.putInValue("/text/in/URL",
38.    "http://host/hello.txt");
39. catContext.putOutValue(
40.    "/text/out/contents", null);
41.
42. sortContext.putInValue(
43.    "/text/in/contents", null);
44. sortContext.putOutValue(
45.    "/text/out/sorted", null);
46.
47. uniqContext.putInValue(
48.    "/text/in/sorted", null);
49. uniqContext.putOutValue(
50. "/text/out/URL","http://host/bye.txt");
51.
52. //Map context outputs to inputs
53. catContext.map("/text/out/contents",
54.    "/text/in/contents", sortContext);
55. sortContext.map("/text/out/sorted",
56.    "/text/in/sorted", uniqContext);
57.
58. catTask.setContext(catContext);
59. sortTask.setContext(sortContext);
60. uniqTask.setContext(uniqContext);
61.
62. // exert collaboration
63. sortJob.exert(null);
```

In the above EO program we create three signatures (lines 2-9), each signature is defined by an interface name and an operation name that we want to run by any servicer implementing the interface. We use the three signatures to create three tasks (lines 12-18) and by line 19, we have three separate commands `cat`, `sort`, and `uniq` to be used in the `sort.sh` script. The three tasks are combined into the job by analogy to piping Unix commands in the `sort.sh` script. Thus, by line 26, we have added these commands to `sort.sh` script, but have not provided input/output parameters nor piped them together. Lines 28-50 create and define three service contexts for our three tasks. By line 51, we have specified some input and output parameters, but still no piping. Lines 53-56 define mapping of context output parameters to the related context input parameters. The parameters are defined by context paths from a source context to a target context. The target context is the last parameter in the `map` operation. By line 57, we have piping setup and by the analogy our `sort.sh` script is complete now.

On line 63, we execute `sortJob`. If we use the Tenex C shell (`tcsh`), invoking the Unix script is equivalent to: "`tcsh sort.sh`", i.e., passing the script `sort.sh` on to `tcsh`. Similarly, to invoke the exertion `sortJob`, we call "`sortJob.exert()`". Thus, the exertion is the program and the network shell at the same time, which might first come as a surprise, but close evaluation of this fact shows it to be consistent with the meaning of object-oriented distributed programming. Here, the virtual metacomputer is an ad hock federation that does not exist when the exertion is created. Thus, the notion of the virtual metacomputer is encapsulated in the exertion (specification) that creates the required federation on-the-fly (implementation) to execute the collaboration (process).

4. Service Messaging and Exertions

In object-oriented terminology, a message is the single means of passing control to an object. If the object responds to the message, it has an operation and its implementation (method) for that message. Because object data is encapsulated and not directly accessible, a message is the only way to send data from one object to another. Each message specifies the name (identifier) of the receiving object, the name of operation to be invoked, and its parameters. In the unreliable network of objects; the receiving object might not be present or can go away at any time. Thus, we should postpone receiving object identification as late as possible. Grouping related messages per one request for the same data set makes a lot of sense due to network invocation latency and common errors in handling. These observations lead us to service-oriented messages called exertions. An exertion encapsulates multiple *service signatures* that define operations, a *service context* that defines data, and a *control strategy* that defines how signature operations flow in collaboration. Different types of control exertions (`IfExertion, ForExertion, WhileExertion`) [7] can be used to define flow of control that can also be configured additionally with adequate signature attributes (*flow type* and *access type*—see Section 5).

An exertion can be invoked by calling exertion's `exert` operation: `Exertion.exert(Transaction) :Exertion`, where a parameter of the `Transaction` type is required when the transactional semantics is needed for all participating nested exertions within the parent one, otherwise can be `null`. Thus, EO programming allows us to submit an exertion onto the network and to perform executions of exertion's signatures on various service providers indirectly, but where does the service-to-service communication come into play? How do these services communicate with one another if they are all different? Top-level communication between services, or the sending of service requests (exertions), is done through the use of the generic `Servicer` interface and the operation `service` that all SORCER services are required to provide (`Servicer.service(Exertion, Transaction) :Exertion`). This top-level service operation takes an exertion as an argument and gives back an exertion as the return value. How this operation is used in the FMI framework is described in detail in [7].

So why are exertions used rather than directly calling on a provider's method and passing service contexts? There are two basic answers to this. First, passing exertions helps to aid with the network-centric messaging. A service requestor can send an exertion out onto the network (`Exertion.exert()`) and any servicer can pick it up. The servicer can then look at the interface and `PROCESS` operation requested within the exertion, and if it doesn't implement the desired interface or provide the desired operation, it can continue forwarding it to another provider who can service it. Second, passing exertions helps with fault detection and recovery. Each exertion has its own completion state associated with it to specify if it has yet to run, has already completed, or has failed. Since full exertions are both passed and returned, the requestor can view the failed exertion to see what method was being called as well as what was used in the service context input nodes that may have caused the problem. Since exertions provide all the information needed to execute a task including its control strategy, a requestor would be able to pause a job between tasks, analyze it and make needed updates. To figure out where to resume a job, a `Jobber` service would simply have to look at the task's completion states and resume the first one that wasn't completed yet.

5. Push and Pull Collaborations

SORCER also extends exertion execution abilities through the use of a rendezvous service implementing the `Spacer` interface. The `Spacer` service can drop exertions into a shared object space, implemented using JavaSpaces [2], in which collaborating servicers can retrieve matching exertions, execute them, and return the resulting exertions back to the object space. When the attribute *access type* of a `PROCESS` signature is set to `PULL` then the associated exertion is passed onto a `Spacer`, otherwise (access type is `PUSH`) the exertion is passed directly on to the servicer specified by the `PROCESS` signature. Another signature attribute—*flow type* manages the flow of control (`SEQUENTIAL`, `PARALLEL`, or `CONCURRENT`) for all component exertions at the same level.

In Figure 2 four use cases are presented to illustrate push vs. pull exertion processing with either `PUSH` or `PULL` access types. We assume here that an exertion is a job with two component exertions executed in parallel (sequence numbers with a and b), i.e., the job's signature flow type is `PARALLEL`. The job can be submitted directly to either `Jobber` (use cases: 1—access is `PUSH`, and 2—access is `PULL`) or `Spacer` (use cases: 3 —access is `PUSH`, and 4—access is `PULL`) depending on the interface defined in its `PROCES` signature, the `Jobber` or `Spacer` interface respectively. Thus, in cases 1 and 2 the signature's interface is `Jobber` and in cases 3 and 4 the signature's interface is `Spacer` as shown in Fig. 2. The exertion's `ServicerAccessor` delivers the right service proxy dynamically, either for a `Jobber` or `Spacer`. If the access type of the parent exertion is `PUSH`, then all the component exertions are directly passed on to servicers matching their `PROCESS` signatures (case 1 and 3), otherwise they are written into the exertion space by a `Spacer` (case 2 and 4). In the both cases 2 and 4, the component exertions are pulled from the exertion space by servicers matching their signatures as soon as they are available. Thus, `Spacer`s provide efficient load balancing for processing the exertion space. The fastest available sevicer gets an exertion from the space before other overloaded or slower servicers can do so. When an exertion consists of component jobs with different access and flow types, then we have a *hybrid* case when the collaboration potentially executes concurrently with multiple *pull* and *push* subcollaborations at the same time.

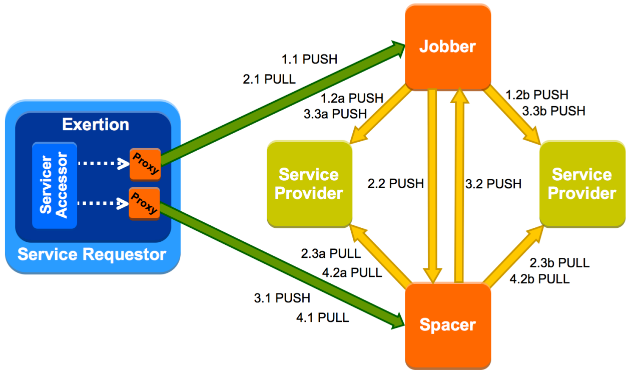

Figure 2. Push vs. pull exertion processing

6. Conclusions

A collaborative distributed system is not just a collection of distributed objects—it's the network of dynamic objects that come and go. From an object-oriented point of view, the network of collaborating object peers is the *problem domain* of object-oriented distributed programming that requires relevant abstractions in the *solution space*. The exertion-based programming introduces the new abstraction of the solution space with *servicers* and *exertions* instead of object-oriented conventional *objects* and *messages*. An exertion not only encapsulates operations, data, and control strategy, it also encapsulates a related federation of servicers that provide implementation for the exertion's collaboration.

Executing a collaboration implicitly, by sending its exertion onto the network—`Exertion.exert()`, means binding in runtime autonomically to currently available servicers on the network. The federation becomes the implementation of the exertion—a truly P2P collaborative program. When the federation is formed then each exertion operation has its corresponding method (code) on the network available. Services, as specified by exertion signatures, are invoked only indirectly by passing exertions on to servicers via service object proxies that in fact are access proxies allowing servicers to enforce security policies on access to required operations. If the access to use the operation is granted, then the operation defined by an exertion's `PROCESS` signature is invoked by reflection. Service providers can be easily deployed in SORCER by injecting configurable implementation of domain-specific interfaces. The providers register proxies, including smart proxies, via dependency injection using twelve methods investigated already in SORCER.

The EO framework (exertion—specification, collaboration—process, federation—implementation) allows for the P2P computing via the `Servicer` interface, extensive modularization of `Exertion`s and `Servicer`s, and extensibility from the applied Command design pattern [4]. Various elements of the presented EO programming methodology has been successfully deployed and tested in multiple concurrent engineering and large-scale distributed application [1, 2, 6, 9].

7. References

[1] FIPER: Federated Intelligent Product EnviRonmet. Available at: http://sorcer.cs.ttu.edu/fiper/fiper.html. Accessed on: March 1, 2008.
[2] Freeman, E., Hupfer, S., & Arnold, K. JavaSpaces™ Principles, Patterns, and Practice, Addison-Wesley, ISBN: 0-201-30955-6 (1999)
[3] Goel, S., Shashishekara, Talya S.S., Sobolewski M., Service-based P2P overlay network for collaborative problem solving, Decision Support Systems, Volume 43, Issue 2, March 2007, pp. 547-568 (2007)
[4] Grand, M., Patterns in Java, Volume 1, Wiley, ISBN: 0-471-25841-5 (1999)
[5] Jini architecture specification, Version 2.1. Available at: http://www.sun.com/software/jini/specs/jini1.2html/jini-title.html. Accessed on: March 1, 2008.
[6] Kolonay, R.M., Sobolewski, M., Tappeta, R., Paradis, M., Burton, S. 2002, Network-Centric MAO Environment. The Society for Modeling and Simulation International, Western Multiconference, San Antonio, TX (2002)
[7] Sobolewski, M., Service-oriented Programming, SORCER Technical Report SL-TR-13 (2008). Available at: http://sorcer.cs.ttu.edu/publications/papers/2008/SL-TR-13.pdf. Accessed on: March 1, 2008.
[8] SORCER Research Group. Available at: http://sorcer.cs.ttu.edu/. Accessed on: March 1, 2008.
[9] SORCER Research Topics. Available at: http://sorcer.cs.ttu.edu/theses/ Accessed on: March 1, 2008.
[10] Thain D., Tannenbaum T., Livny M. Condor and the Grid. In Fran Berman, Anthony J.G. Hey, and Geoffrey Fox, editors, Grid Computing: Making The Global Infrastructure a Reality. John Wiley (2003).