File Location Management in Federated Computing Environments

Chris Hard[1] and Michael Sobolewski[2]
[1] Texas Tech University. Lubbock, TX
Email: chris.hard@ttu.edu
[2]Texas Tech University, Lubbock, TX
Email: sobol@cs.ttu.edu

Abstract—**The major objective of the Service Oriented Computing Environment (SORCER) is to form dynamic federations of network services that provide shared data, applications, and tools on a service grid along with exertion-oriented programming. To meet the requirements of these services in terms of data sharing and managing in the form of data files, a corresponding federated file system was developed. The file system fits the SORCER philosophy of interactive exertion-oriented programming, where users create service-oriented programs and can access data files in the same way they use their local file system. The federated file system provides data redundancy in the form of file replication. However, there is no efficient management of these file replicas after creation and the location to which they are replicated to is not taken into account. Thus, a separate File Location Management Framework was developed to fit with the SORCER metacomputing philosophy and to manage autonomically file replication.**

Index Terms—**Federated file system, autonomic file location, exertion-oriented programming, metacomputing, service-oriented systems**

I. Introduction

Building on the OO paradigm is the service-object-oriented (SOO) paradigm, in which the objects are distributed, or more precisely they are remote (network) objects that play some predefined roles. A service provider is an object that accepts remote messages, called service exertions, from service requestors to execute an item of work. A task exertion is an elementary service request – a kind of elementary remote instruction (statement) executed by a service provider. A composite exertion, called a job exertion, is defined in terms of tasks and other jobs - a kind of procedure executed by a service provider. The executing exertion is a SOO program that is dynamically bound to all relevant and currently available service providers on the network. This collection of providers identified in runtime is called an exertion federation, or a service space. While this sounds similar to the OO paradigm, it really isn't. In the OO paradigm, the object space is a program itself; here the service space is the execution environment for the exertion, which is a network OO program. This changes the game completely. In the former case, the object space is hosted by a single computer, but in the latter case the service providers are hosted by the network of computers. The overlay network of service providers is called the service grid [13] and an exertion federation is called a virtual metacomputer. The metainstruction set of the metacomputer consists of the method set defined by all service providers in the grid. Do you remember the eight fallacies of network computing [4]? Creating and executing a SO program in terms of metainstructions requires a completely different approach than creating a regular OO program [12].

The SORCER environment [13] provides the means to create interactive SOO programs and execute them without writing a line of source code via zero-install, interactive service interfaces. Exertions can be created using interactive user interfaces downloaded directly from service providers, allowing the user to execute and monitor the execution of exertions in the SOO metacomputer. The exertions can also be persisted for later reuse. This feature allows the user to quickly create new applications or programs on the fly in terms of existing tasks and jobs. SORCER introduces federated method invocation based on peer-to-peer (P2P) [7], [9] and dynamic service-oriented Jini architecture [1].

SILENUS is a federated file system which builds on top of the SORCER philosophy. Federating services work together to provide the functionality of the file system. These services can be broadly categorized into gateway services, management services, and data services [1], [2].

SILENUS provides data reliability and availability in the form of file replication. However, once a file is created and replicated there is no management of these replicas. Also by taking into account the user access behavior, performance may be increased by strategically choosing a location to replicate a file to. Thus, to dynamically manage the locations of replicas and to provide quality of service to data store providers a separate framework was developed called LOCO (Location Optimizer).

This paper is organized as follows: Section 2 briefly describes the SORCER metacomputing system; Section 3 introduces service messaging and exertions; Section 4 presents federated file system methodology; Section 5 describes the LOCO architecture; and Section 6 provides concluding remarks.

II. SORCER

SORCER (Service Oriented Computing EnviRonment) [13] 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 [1]. While Jini focuses on service management in a networked environment, SORCER focuses 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* [12], but hides all the 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 collaboration. These messages are called service exertions and 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, each exertion's operation has its corresponding method (code) available on the network. 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 to be elementary (tasks) or compound (jobs) activities 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 that will run the collaboration as specified in the EO program and return the resulting exertion back to 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*

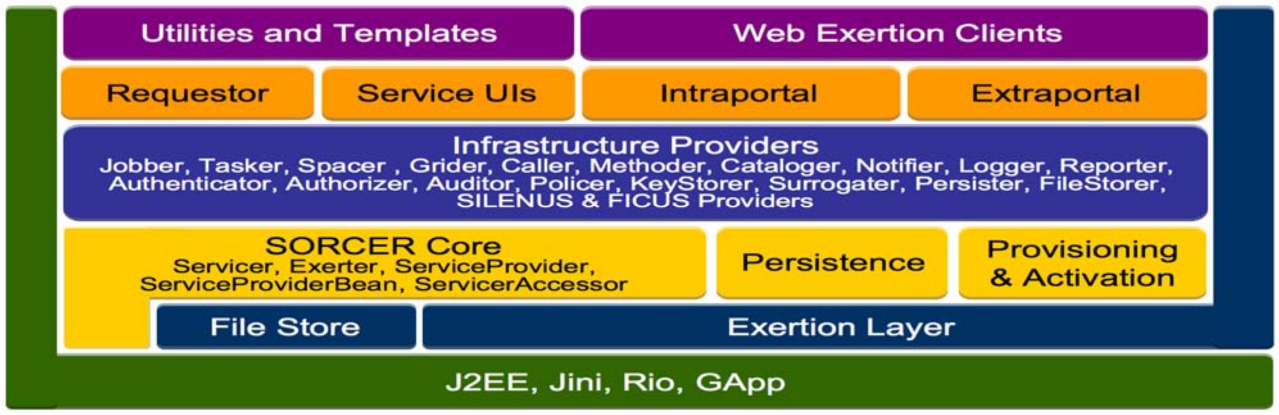
Utilities and Templates
Web Exertion Clients
Requestor
Service UIs
Intraportal
Extraportal
Infrastructure Providers
Jobber, Tasker, Spacer , Grider, Caller, Methoder, Cataloger, Notifier, Logger, Reporter, Authenticator, Authorizer, Auditor, Policer, KeyStorer, Surrogater, Persister, FileStorer, SILENUS & 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.

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 [16]. Here, instead of exertions being executed by services providing business logic for invoked exertions, the business logic comes from the service requestor's executable codes 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 standard `Caller`'s service context, or mobile Java code executed by `Methoders` with a matching service context provided by the requestor.

III. 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 the 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, and WhileExertion`) [12] can be used to define flow of control that can also be configured additionally with adequate signature attributes (*flow type* and *access type*).

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)`.
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 federated method invocation framework is described in detail in [12].

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 composition 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 rendezvous service would simply have to look at the task's completion states and resume the first one that wasn't completed yet.

IV. SILENUS FILE SYSTEM

SILENUS [1], [2] is a federated file system based on service messaging introduced in Section 3. It provides dynamic access to files referenced in service contexts of exertions. It consists of several services that federate together to provide the functionality of the file system. Each service may be replicated on as many hosts as needed. These services may be categorized into gateway services, data services, and management services. The service oriented nature of SILENUS makes it very easy for someone to create new functionality for the file system by implementing additional services.

The SILENUS file system makes a few assumptions about the data being stored. First, file metadata is very small. Second, file data is relatively large therefore it should be replicated for reliability and availability but not onto every data store [1], [2].

A.Data services

The data services consist of a metadata store service and a byte store service. The metadata store service stores attributes that can be derived from the files themselves. This includes name, creation date, size, file type, location, etc. The metadata service provides functionality to create, list, and traverse directories [1], [2].

The byte store service is used for storing the actual file data. It does not provide for storing attributes about the file but does allow for retrieving attributes of a file e.g., retrieving the file size and checksum to verify integrity of the file. Stored files are usually encrypted but may be stored unencrypted for performance reasons [1], [2].

B.Management services

SILENUS includes several management services such as the SILENUS Façade, Jini Transaction Manager, Byte Replicator, and other optimizer services. The SILENUS Façade manages the coordination and provides a dynamic entry point between the metadata stores and byte stores [1], [2]. The Façade also provides a zero install user interface, through the use of a Service UI [10], which allows the users to view the files in the system similar to the way they would view files in a traditional file system.

The Transaction Manager is a Jini [1] standard service which the SILENUS Façade uses to ensure two-phase commit semantics for file uploads and downloads. The Byte Replicator and other optimizer services are used for autonomic administration. The optimizer services may make decisions on where to move files, which services should be started or shutdown, and where to store replicas. Each optimizer service is a separate component so it makes it very easy for and administrator to create more optimizer services. In traditional file systems an administrator has to provide some management of the data but in SILENUS an administrator may select which kind of optimizer services to deploy and where to deploy them [1], [2]. This also makes SILENUS highly scalable.

C.Gateway services

The gateway services provided by SILENUS are client modules that provide access to the SILENUS file system. Some examples of gateway services are the NFS Adapter, JXTA Adapter, WebDAV Adapter, and Mobile Adapter. The NFS Adapter provides a mapping from the NFS protocol to SILENUS for older UNIX systems that do not have WebDAV support. A WebDAV Adapter was developed to provide support for newer operating systems that support WebDAV such as Windows, Mac OS X, and newer versions of UNIX [1], [2]. These are just a few of the gateway services that have been created. The service oriented nature of SORCER makes it very easy for someone to create new services for SILENUS.

V. LOCO

To achieve availability and reliability of files, SILENUS provides data redundancy in the form of file replication. It uses an active replication scheme which means that all replicas are treated as if they are the original. The drawback of this scheme is that it requires a lot of coordination in that if an update occurs on one replica then all of the replicas need to be updated. The coordination is currently implemented in SILENUS; however there is no management of these replicas after creation. This section describes how SILENUS handles file replication and describes the LOCO framework.

Currently when a file is created it is replicated to a default number of byte stores which is two. The user may interactively change this number through the Service UI [10] so as to add priority to files if needed.

SILENUS also does not take into account the file size or available system resources, which is not efficient e.g., replicating a file that is extremely large may not be feasible if there is not enough storage space. If a file is going to be replicated then the available storage space of each provider should be taken into account.

When a file is replicated in SILENUS, a random byte store is chosen to replicate the file to. By choosing a byte store based on some criteria, such as available size, location, and user habits, it can increase performance by lowering network traffic and decreasing download/upload time.

A separate framework called LOCO has been developed to autonomically manage these issues and to provide quality of service for data store providers. It monitors user's access habits so that it can make logical decisions on where to replicate the files to. It will also dynamically manage the number of times each file is replicated depending on file size, available storage space at each byte store provider, and the byte store host type (e.g., server, desktop, laptop).

LOCO will replicate a file for several reasons, if a byte store becomes unavailable then all of the files that were located there will be replicated and if a file is uploaded into the system LOCO will decide on an appropriate number of times to replicate the file. LOCO may also delete certain replicas, for example, if a byte store becomes unavailable and all of the files stored there are replicated, then when that byte store becomes available again LOCO may choose to delete some of the replicas.

The LOCO framework is an extension to SILENUS and is comprised of four services (see Figure 2), which is discussed in the following sections, and runs in the SORCER environment. It includes a Locator service, Sweeper service, Replicator service, and a Resource Usage Store service described in detail below.

LOCO also makes several qualities of service guarantees to data store providers. First, a file will not be replicated to a storage location that already contains the file or replica of the file. Second, a minimum number of replicas, which may be specified by the user or the locator service, will be maintained as long as there are enough storage locations present in the network to satisfy the number.

D.Resource usage store service

The Resource Usage Store is a database service which stores information about byte stores and tracks user's behaviors. The Resource Usage Store connects to a metadata store and retrieves information about existing files in the file system. It will then register a listener with the metadata store so that it will receive FileStoreEvents when a user uploads, downloads, or deletes a file. If an upload or download occurs, timing data is collected and stored in the Resource Usage Store. This information is used when choosing which byte store to replicate a file to.

The Resource Usage Store also provides a zero-install user interface (ServiceUI [10]) that can be downloaded by a service browser such as Inca X [5] and allows an administrator to view all of the tables in the Resource Usage Store. It also allows the administrator to delete records that may be invalid or not useful anymore.

E.Locator service

The Locator service can examine the state of the file system at a specified interval and choose to replicate or delete files if necessary. An administrator may set this time interval in a configuration file. When a user changes the minimum number of replicas for a file, the locator will receive this update and either delete or replicate the file.

The Locator is also responsible for managing a cache of byte stores and monitoring whether any appear or disappear. If a byte store appears then the file table in the Resource Usage Store needs to be updated and the Locator needs to check if any of the files contained in this byte store need to be replicated or deleted. If a byte store disappears then the files that were stored there may need to be replicated.

The Locator service also provides a zero-install user interface (ServiceUI [10]) that can be obtained through the use of a service browser such as Inca X [5]. The user interface will allow an administrator to view information about all of the available and unavailable byte stores in the network such as host name, available space, and used space. The interface will also allow an administrator to view records on all of the users that have accessed files within the system. It displays the user's IP address and user name along with which files they accessed, action they took (upload, download or delete), and time

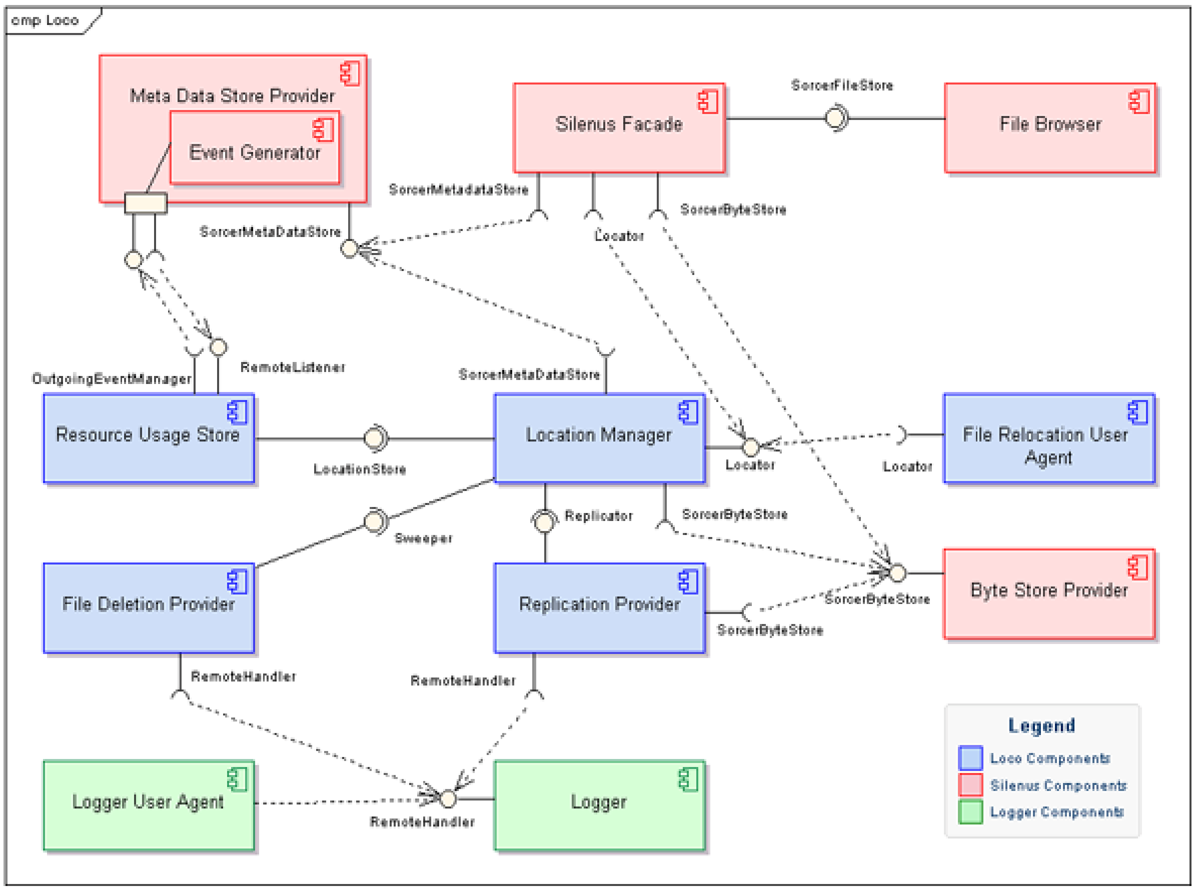

Figure 2 LOCO architecture (UML component diagram)

and date that they accessed them. The interface also allows an administrator to remove a user from being tracked.

F.Replicator service and sweeper service

The Replicator extends the SILENUS Replicator to replicate files. It will log all replications to the Remote Logger service in SORCER. The Locator also displays these records in the service UI.

When the Locator service decides a file is to be replicated it sends a message to the Replicator with the file ID and the number of times to replicate the file. The Replicator then needs to decide which byte stores to replicate the file to. First, the byte store should not already contain the file. Second, the byte stores should contain enough storage space to hold the file. Third, the replicated files should be close to the users that use the file the most to increase download times. The Replicator will query a Resource Usage Store for the users that use the file and then it will choose the users that use the file the most to replicate near.

To compare the locations of the byte stores, the Replicator looks up the Autonomous System number associated with the byte store's IP address by querying a whois [17] database. It then uses a table to calculate how many different Autonomous Systems are between the user and the byte store. It chooses the one that has the lowest amount of Autonomous Systems to travel through.

The method of determining the proximity of the byte stores to the user address currently is an open research project at the SORCER Laboratory [14]. The project was aimed at finding the closest peers to a BitTorrent user to make downloading and uploading faster and to cut down on network traffic.

The Sweeper service can delete a given file from a given byte store. The locator service will determine that there are too many replicas of a file in the system and tell the sweeper to delete one or some of them. The Sweeper will log all of the deletions to a Logging Service where the administrator will be able to view the deletions that have taken place.

VI. CONCLUSIONS

File replication in a distributed file system provides availability, fault tolerance, and may enhance performance. However, it comes at a price, as replication uses up more storage space and also adds overhead for the coordination of these replicated files. Replicated file systems also require a substantial amount of administration.

LOCO adds scalability to SILENEUS by dynamically managing file replicas in the system. As the number of users and files grow in SILENUS, LOCO manages replicating files and the location of these files autonomically so that an administrator will not manually have to.

LOCO provides SORCER with high availability and reliability of data by replicating files and then autonomically managing these replicas. The LOCO framework was deployed and tested successfully with the SILENUS file system.

REFERENCES

[1] Berger, M., Sobolewski, M. "Lessons Learned From the SILENUS Federated File System", Springer Verlag, Sao Jose Dos Campos, Brazil, July 16-20, 2007.

[2] Berger, M., Sobolewski, M. "SILENUS – A Federated Service Oriented Approach to Distributed File Systems", Next Generation Concurrent Engineering, New York, 2005.

[3] Coulouris, G., Dollimore, J., Kindberg, T. *Distributed Systems Concepts and Designs*, Addison-Wesley, London and Palo Alto, June 2000.

[4] Deutsch, P. (1994). The Eight Fallacies of Distributed Computing, Available at: <http://blogs.sun.com/jag/resource/Fallacies.html>, Retrieved 29 February 2008.

[5] Inca X Service Browser. Available at: <http://www.incax.com/>, Retrieved 30 March 2008.

[6] Jini Architecture Specification. Available at: <http://www.sun.com/software/jini/specs/jini1.2html/jini-title.html>, Retrieved 27 March 2008.

[7] Li, Sing. JXTA *Peer-to-Peer Computing with Java*, Wrox Press Ltd., 2001.

[8] Newmarch, J. "Jan Newmarch's Guide to Jini Technologies", Available at:<http://www.sorcer.cs.ttu.edu/newmarch/docs/Jini.html>, Retrieved 29 March 2008.

[9] Oram A (ed). Peer-to-Peer: Harnessing the Benefits of Disruptive Technology, O'Reilly, 2001.

[10] The ServiceUI Project. Available at: <http://www.artima.com/jini/serviceui/>, Retrieved 28 March 2008.

[11] Silberschatz, A., Galvin, P.B., & Gagne, G. *Operating System Concepts* (7th ed.). Hoboken, NJ: John Wiley & Sons, Inc, 2005.

[12] Sobolewski, M., Exertion Oriented Programming, IADIS, vol. 3 no. 1, pp. 86-109, ISBN: ISSN: 1646-3692 (2008).

[13] Sobolewski, M., SORCER: Computing and Metacomputing Intergrid, 10th International Conference on Enterprise Information Systems, Barcelona, Spain (2008). Available at: http://sorcer.cs.ttu.edu/publications/papers/2008/iceis-intergrid-08.pdf.

[14] SORCER Lab. Retrieved April 20, 2008, from: http://sorcer.cs.ttu.edu/.

[15] Sztajnberg, A., Loques, O. "Bringing QoS Specifications to the Architectural Level", Available at: <http://citeseer.ist.psu.edu/cache/papers/cs/26895/http:zSzzSzwww.gta.ufrj.brzSzftpzSzgtazSzTechReportszSzSzLo00b.pdf/bringing-qos-specifications-to.pdf>, Retrieved: 29 March 2008.

[16] 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).

[17] Team Cymru, Available at: <http://www.team-cymru.org/?sec=8&opt=26>, Retrieved: 19 April 2008.

[18] Turner, A., Sobolewski, M. "A Federated Service-Oriented File Transfer Framework", Springer Verlag, Sao Jose Dos Campos, Brazil, July 16-20, 2007.