Version Control Management for Federated Service-oriented File Sharing

Michael Sobolewski[a,b] and Amaresh Ghosh[a]

SORCER Research Group, SORCERsoft.org
[a]Texas Tech University, Lubbock, Texas
[b]Polish-Japanese Institute of IT, Warsaw, Poland

Abstract. The major objective of the Service Oriented Computing Environment (SORCER) is to form dynamic federations of network services that provide for concurrent engineering systems: shared data, applications, tools, and utilities on a service grid along with exertion-oriented programming. To meet the requirements of these services in terms of design 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 for distributed collaborative applications, where users create service-oriented programs and can access data files in the same way they use their local file system. However, there was no efficient management of file explicit versions during complex design in related concurrent engineering systems. Thus, a separate Version Control Management Framework was developed to fit with the SORCER metacomputing philosophy and to manage effectively versions of all files in a uniform way.

Keywords: version control management, service-oriented architectures, metacomputing

1 Introduction

Building on the OO paradigm is the service-object oriented (SOO) paradigm, in which the service objects are distributed, or more precisely they are remote objects that play some predefined roles. Before we delve into the proposed new metacomputing and metaprogramming concepts, the introduction of some terminology used throughout the paper is required:

- A ***computation*** is a process following a well-defined model that is understood and can be symbolically expressed and physically accomplished (physically expressed). There are many ways of expressing a process in a logic circuit, function, algorithm, message, protocol, network topology, virtual organization, etc. Four orthogonal classes of computations can be distinguished: *digital* vs. *analog*, *sequential* vs. *parallel* vs. *concurrent*, *batch* vs. *interactive*, *monolithic* vs. *distributed*. A computation can be seen as a purely physical phenomenon occurring inside a system called a ***computer***.
- Computing requires a ***computing platform*** (runtime) to operate. Computing platforms that allow programs to run require a *processor*, *operating system*,

and *programming environment* with related tools to create symbolic process expressions—***programs***. Usually, a computation is physically expressed by a processor and symbolically expressed by a program created in the relevant programming environment. Thus, a computation is the *actualization* of a program by operating system on its processor.

- A ***distributed computation*** allows for sharing computing resources usually collocated on several remote computers (compute nodes) to collaboratively run a single complex computation in a transparent and coherent way. In distributed computing a computation is divided into subcomputations that execute on a collection of compute nodes. Thus, in distributed computing, computations are decomposed into programs, processes, and compute nodes. A ***metacomputer*** is an interconnected and balanced set of compute nodes that operate as a single unit, which is accessible by its computing platform (*metaprocessor*, *metaoperating system*, and *metaprogramming environment*).
- A ***metacomputation*** is a form of distributed computation (a computation of computations) determined by *collaborating computations* that a metacomputer can interpret and execute. In *metacomputing* computations are decomposed into *services*, *service providers,* and *processors*. The *service provider* selected at runtime by a metaoperating system defines a required service—a metainstruction being a provider's program. A collection of all service providers selected and managed for a metacomputation is called a ***virtual metaprocessor***.
- A ***metaprogram*** is an expression of metacomputation, represented in a *programming language*, which a *metacomputer* follows in processing shared data for a ***service collaboration*** (*workflow*) managed by its *metaoperating system* on its virtual *metaprocessor*.
- A ***service-oriented architecture*** (*SOA*) is a software architecture using loosely coupled service providers that integrates them into a distributed computing system by means of service-oriented programming. Service providers in service-oriented programming are made available as independent service components that can be accessed without a priori knowledge of their underlying platform, implementation, and location. While the client-server architecture separates a client from a server, SOA introduces a third component, a service registry. The registry allows metaoperating system to find service providers with no need to define their static locations on the overlay network.

Therefore, every metacomputer ***requires a platform*** that allows software to run utilizing multiple autonomous computing nodes that communicate through a computer network. Different distributed platforms can be distinguished along with corresponding metaprocessors—virtual organizations of computing nodes. SORCER [10], [13] is a metacomputing platform for concurrent engineering applications.

In SORCER a *service provider* is a service object that accepts requests from service requestors to execute a collaborative work. A specification of the collaborative work is called an *exertion*. An exertion *exerts* the service providers dynamically federating (virtual metaprocessor) for its service collaboration. A *task exertion* is an elementary service request—a kind of elementary remote

metaroutine, being a program, executed by a service provider. A composite exertion, called a *job exertion*, is defined in terms of tasks and other jobs—a kind of metacoroutine executed by collaborating providers managed by the metaoperating system. The executing exertion is a SOO metaprogram that is dynamically bound to all relevant and currently available service providers on the network. This collection of collaborating providers identified in runtime is called an *exertion federation*. The overlay network of all service providers is called the *service grid* and the exertion federation forms a *virtual metaprocessor* at runtime. The metainstruction set of the metaprocessor consists of the operations defined by all service providers in the service grid. Creating and executing a SOO program in terms of metainstructions requires a completely different approach than creating a regular OO program [9], [10].

The SORCER environment provides the means to create interactive SOO programs and execute them as complex concurrent engineering applications. 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 virtual 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) and dynamic service-oriented Jini architecture [5].

SILENUS [1], [2] builds on top of the SORCER philosophy and provides data reliability and availability in the form of file replication. However, once a file version is created and replicated there is no management of these replica versions (revisions). Thus, to manage the versions of replicas a separate framework was developed called FVS (**F**ederated **V**ersioning for service-oriented file **S**ystem).

This paper is organized as follows: Section 2 briefly describes the SORCER metacomputing system; Section 3 presents federated file system methodology; Section 4 describes the version management architecture; and Section 5 provides concluding remarks.

2 SORCER

SORCER (Service Oriented Computing EnviRonment) is a federated service-to-service (S2S) metacomputing environment that treats service providers as service objects with well-defined semantics of a federated service-object oriented architecture. It is based on Jini [5] semantics of services in the network and Jini programming model with explicit leases, distributed events, transactions, and discovery/join protocols. While Jini focuses on service management in a networked environment, SORCER focuses on exertion-oriented programming and the execution environment for exertions [10]. SORCER uses Jini discovery/join protocols to implement its *exertion-oriented architecture* (EOA) using *federated method invocation* [10], but hides all the low-level programming details of the Jini programming model.

In EOA, a service provider is a service object that accepts requests from service requestors to execute collaboration. These requests 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 executed by a single service provider or a small-scale federation managed by the receiving provider 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 large-scale federation managed by the SORCER OS executes a job. 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 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 all service providers is called the *service grid* and an exertion federation is in fact a *virtual metaprocessor*. The metainstruction set of the metaprocessor 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 *data context*. The data 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. 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 service 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 data context.

Domain specific servicers within the federation, or task peers (*taskers*), execute task exertions. *Rendezvous* peers (jobbers and spacers) [10] coordinate execution of job exertions. Providers of the `Tasker`, `Jobber`, and `Spacer` type are basic *system providers* of the SORCER operating system; 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

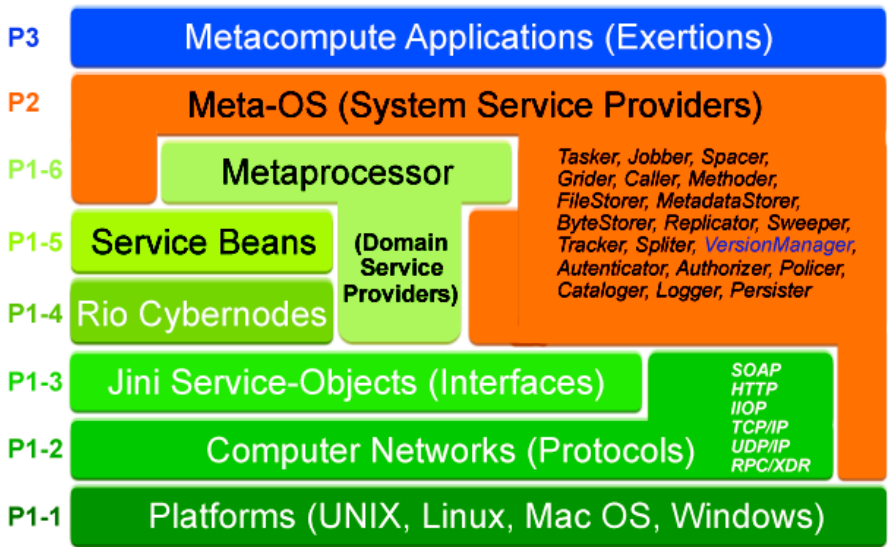

Figure 1. SORCER platform: metaprocessor (green shades), meta-OS (orange), programming environment-exertions (blue).

other collaborations to join. Also, SORCER supports a traditional approach to grid computing similar to those found, for example in Condor [15]. 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 [13] 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 data context accordingly with the code downloaded from the network. In either case, the business logic comes from requestors; it is a conventional executable code invoked by Callers with the standard Caller's data context, or mobile Java code executed by Methoders with a matching data context provided by the requestor.

3 SILENUS File System

SILENUS [1], [2], [16] is a federated file system, which builds on top of the SORCER philosophy. It provides dynamic access to files referenced in data contexts of exertions. SILENUS 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].

1. 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 [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 [2].

2. 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 [7], 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 standard service that 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.

3. 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.

4 Version Control Management Architecture

An important element in the modern CE process is version control (also known as revision or source control). Cooperating designers commit their changes incrementally to a common source repository, which allows them to collaborate on data without resorting to crude file-sharing techniques (shared directories, drives, emails). Source control tools track all prior versions of all files, allowing designers to "time travel" backward and forward in their data to determine when and where changes are introduced. These tools also identify conflicting simultaneous modifications made by two (poorly-communicating) team members, forcing them to work out the correct solution (rather than blindly overwriting one or the other original submission).

The FVS (**F**ederated **V**ersioning for service-oriented file **S**ystem) system is collaboration of three services described below. The FVS architecture in the form of the UML component diagram is depicted in Figure 2.

4.1 FVS Version Manager

The FVS Version Manager is similar to SILENUS metadata store. It contains metadata information of various versions of files. Metadata information of federated versioning system includes information about:

a. all changed paths
b. log message
c. name of the author of the commit
d. the timestamp when the commit was made
e. special character describing how the path was changed ('A' - added, 'D' - deleted or 'M' - modified).
f. information about SVN ENTRY UUID
g. information about committed revision number
h. information about SVN ENTRY CHECKSUM

Each version of file contains a file name and unique Version ID (VID). FVS uses this service same way as SILENUS metadatastore does except the FVS Version Manager points to the FVS tracker rather than the SILENUS tracker. The version management service is responsible for storing all the metadata information for the managed files. Metadata information is stored in a database running in this service. The version management store provides attributes for the files stored in the file system. The analogy in a traditional storage system is the file system. The metadata information creates the well-known hierarchical structure. Files in the version management store are identified by UUIDs and VIDs. The metadata provides mapping from and to file names. Version manager services are synchronized while connected. All version manager services contain the same information. Should a version manager services be disconnected while its information changes, it will be resynchronized when it is connected back to the other metadata version manager services.

As in internal database, an embedded Berkeley database is used. Using an embedded database makes installation much easier; it does not require the installation of external database management system. The database access itself is

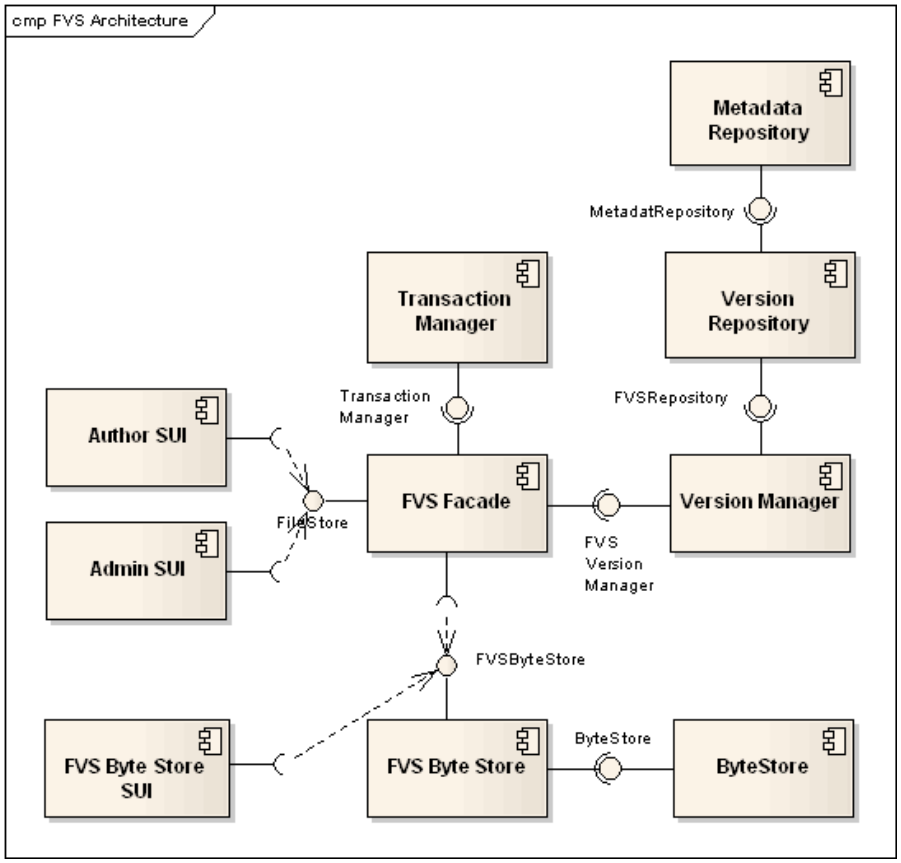

Figure 2. FVS Architecture; SUI-Service UI [7].

implemented using the data access object pattern [6] for extensibility and support for other database management systems if needed.

4.2 FVS Byte Store

The FVS byte store service persists file content of all versions within the SIELNUS file system. A byte store service holds no metadata information about the files; it maintains other than physical data that can be calculated directly from the files it stores, such as file size or checksums. Even file names are replaced with a type of UUID and VID when stored to one of these storage nodes. A byte store service provides the ability to store and retrieve file data based on the file's UUID and VID, which can be obtained from a version manager.

The ID of the byte store and an entry ID in the byte store identify files in a byte store uniquely. These ID numbers never change. This makes the file storage independent from file metadata such as the file name. The byte store services provide nothing but support for file storage. The advantage is that this service can be then optimized for performance. Unlike the version manager, the byte stores are

not synchronized. File data is much larger than file metadata. Would the file data be replicated on every node the storage capacity would be filled very quickly? It is the job of the optimizer services to provide file data replication.

4.3 FVS Façade

The FVS Façade service acts as an entry point to the file system and provides access to basic file system operations. Its main function is to coordinate with the various services found on the network, combining the abilities of each to perform requested tasks on the behalf of a user.

FVS includes the FVS Façade service that helps coordinate the other services to provide the flexible file system functionality. For example, when a user wants to download a file, the façade service contacts with version manager to get the version and storage location of the file, then it contacts the given FVS byte store to begin transferring the file data to the user's local machine.

The FVS Façade is split up into two parts: a provider and a smart proxy. In contrast to a dumb proxy that provides business logic to requestor only on the provider side, a smart proxy provides business logic on the both: the requestor and provider. The FVS Façade provider is responsible for doing a lot of background processing and service discovery that end-user machines should not have to manage. In particular, the FVS Façade provider will check with a Registrar service to find any available version manager available on the network and for reliability would maintain a cache of proxies for each required service that is found. When a FVS Façade proxy is requested from the network (for example, by a service browser), the Registrar service provides the smart proxy registered by the FVS Facade provider. The FVS Façade proxy is the component responsible for doing much of the coordination between the different services to perform versioning system operations. When the Façade smart proxy obtained from the Registrar service, the proxy provides a Service UI [7] (the FVS user agents, see Figure 2) to allow the user to interact with the file system via a file browser. The proxy asks its parent FVS Façade provider for a version manager proxy from its cache to allow the FVS Facade proxy to browse and display the file system to the user. When a user asks to save or store a file to the file system, the FVS Facade proxy obtains the necessary service proxies to carry out the transactional request.

Acknowledgments. This work was partially supported by Air Force Research Lab, Air Vehicles Directorate, Multidisciplinary Technology Center, the contract number F33615-03-D-3307, Service-Oriented Optimization Toolkit for Distributed High Fidelity Engineering Design Optimization.

5 Conclusions

Till now through the course of FVS research, several file version systems have been examined to investigate how they are setup, what sort of computing environments they use, and what additional functionality they provide. Several modern day file versioning systems are unable to effectively cope with a high

volume of very large files, especially in a metacomputing environment where several federations may want to access the same set of files concurrently.

By leveraging Jini network technology, SORCER allows for various services to run on a network, dynamically discover each other, and collaborate with one another to provide larger overarching services to the end user. Through the use of such services, it is possible to break a file system and a version system down into its separate functional parts (services), and then have these functional parts collaborate with each other to provide file version system capabilities. SORCER allows for these services to be managed, maintained, and even used in a consistent manner, and with the use of smart proxies, there is no need to install such services on every machine that will use them. Through the use of dependency injection, small and simple configuration files can be used to launch various services in a variety of different ways in a fairly simple manner. Autonomic provisioning services can also help to regulate the health of service-oriented federations by making sure that all the required services for a metaprogram are available at all times.

FICUS [16] was designed to utilize SORCER and SILENUS as a basis for creating a service-to-service based file system with file tracking and file splitting capabilities. FVS extends the FICUS functionality by adding file version control management capabilities.

FVS provides dedicated, cohesive and decoupled FVS version manager to maintain file history for the FVS service requestor. The Version Management store contains all versions for each file and persists those versions using the SILENUS [1] framework. So we can easily roll back to earlier version of file based on retained history of files.

Adding the file versioning capabilities the FVS framework helps to enhance and expand the benefits provided by SILENUS. Replicating significantly large version files in their entirety on different storage nodes may not be feasible because it may be difficult to find any storage nodes with enough free space to hold the entire file. By splitting such file version up, fractions of the version of a file can be stored much more easily across multiple machines. An increase in file replication can help to increase version system reliability in the event that a storage node goes down. Since split file versions are stored across multiple storage nodes, each of the nodes can contribute bandwidth to download the file for reassembly rather than relying on the shared bandwidth of a single file server. This means that multiple parts of a large file version can be downloaded from multiple sources simultaneously rather than retrieving the full file from a single server where bandwidth may be shared among all the connected clients.

Since file version can be downloaded from multiple sources at once, bottlenecks in transfer speeds are fewer as the same file can be provided by a different FVS service collaboration. Splitting files into chunks can also help to reduce the cost of transfer errors. When transferring a full file to a single source, an error in communication can often mean that the entire file has to be transferred again. However if a file splits then a communication error occurs; only part of the file would need to be retransferred. Overall, FVS provides many benefits by using a service-oriented architecture when compared to the client-server model employed by many versioning systems in use today.

6 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>. Accessed on: February 20, 2010.

[5] Jini Architecture Specification. Available at: <http://www.sun.com/software/jini/specs/jini1.2html/jini-title.html>. Accessed on: April 20, 2010.

[6] Nock, C., Data Access Patterns: Database Interactions in Object-Oriented Applications, Addison-Wesley Professional, ISBN: 0321555627, 2003.

[7] The ServiceUI Project. Available at: <http://www.artima.com/jini/serviceui/>. Accessed on: April 20, 2010.

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

[9] M. Sobolewski, "Object-Oriented Metacomputing with Exertions," *Handbook On Business Information Systems*, A. Gunasekaran, M. Sandhu, World Scientific, ISBN: 978-981-283-605-2, 2010.

[10] M. Sobolewski, "Metacomputing with Federated Method Invocation", Advances in Computer Science and IT, edited by M. Akbar Hussain, In-Tech, intechweb.org, ISBN 978-953-7619-51-0, s. 337-363, 2009. Available from: http://sciyo.com/articles/show/title/metacomputing-with-federated-method-invocation

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

[12] M. Sobolewski, "Federated Collaborations with Exertions," Proceedings of the 2008 IEEE 17th Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, IEEE Computer Society, 2008, pp. 127-132.

[13] Sobolewski M., SORCER: Computing and Metacomputing Intergrid, 10th International Conference on Enterprise Information Systems, Barcelona, Spain, 2008. Available at: http://sorcersoft.org/publications/papers/2008/C3_344_Sobolewski.pdf.

[14] SORCER Lab. Available at: <http://sorcersoft.org/>. Accessed on: April 20, 2010.

[15] 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.

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