1

Metacomputing with Federated Method Invocation

Michael Sobolewski
Texas Tech University
Lubbock, TX

1. Introduction

The term "grid computing" originated in the early 1990s as a metaphor for accessing computer power as easy as an electric power grid. Today there are many definitions of grid computing (Foster et al., 2001) with a varying focus on architectures, resource management and access, virtualization, provisioning, and sharing between heterogeneous compute domains. Thus, diverse compute resources across different administrative domains form a *compute grid* for the shared and coordinated use of resources in dynamic, distributed, and virtual computing organizations. These organizations are dynamic subsets of departmental grids, enterprise grids, and global grids, which allow programs to use shared resources—collaborative compute federations.

A *computer* as a programmable device that performs symbol processing, especially one that can process, store and retrieve large amounts of data very quickly, requires a computing platform (runtime) to operate. *Computing platforms* that allow software to run require a *processor*, *operating system*, and *programming environment* with related runtime libraries or user agents. Therefore, the grid requires a *platform* that describes a kind of framework to allow software to run utilizing virtual organizations. Different platforms of grids can be distinguished along with corresponding types of virtual federations. However, in order to make any grid-based computing possible, computational modules have to be defined in terms of *platform data*, *operations*, and *relevant control strategies*.

For a grid program, the control strategy is a plan for achieving the desired results by applying the platform operations to the data in the required sequence and by leveraging the dynamically federating resources. We can distinguish three generic grid platforms, which are described below.

Programmers use abstractions all the time. The source code written in a software language is an abstraction of machine language. From machine language to object-oriented programming, layers of abstractions have accumulated like geological strata. Every generation of programmers uses its era's programming languages and tools to build programs of next generation. Each programming language reflects a relevant abstraction, and usually the type and quality of the abstraction implies the complexity of problems we are able to solve.

Procedural languages provide an abstraction of an underlying machine language. An executable file represents a computing component whose content is interpreted as a program by the underlying native processor. A request can be submitted to a *grid resource broker* to execute a machine code in a particular way, e.g., by parallelizing and collocating it dynamically to the right processors in the grid. That can be done, for example, with the Nimrod-G grid resource broker scheduler ("Nimrod", n.d.) or the Condor-G high-throughput scheduler (Thain, 2003). Both rely on Globus/GRAM (Grid Resource Allocation and Management) protocol (Sotomayor & Childers, 2005). In this type of grid, called a *compute grid*, executable files are moved around the grid to form virtual federations of required processors. This approach is reminiscent of batch processing in the era when operating systems were not yet developed. A series of programs ("jobs") is executed on a computer without human interaction or the possibility to view any results before the execution is complete.

We consider a true grid program as the abstraction of hierarchically organized collection of component programs that makes decisions about when and how to run them. This abstraction is a *metaprogram*—a program that manipulates other programs as its data. Nowadays the same computing abstraction is usually applied to the program executing on a single computer as to the metaprogram executing in the grid of computers, even though the executing environments (platforms) are structurally completely different. Most grid programs are still written using software languages (generating native processor code) such as FORTRAN, C, C++, Java, and interpreted languages such as Perl and Python the way it usually works on a single host. The current trend is still to have these programs and scripts define grid computational modules as *services*. Thus, most grid computing modules are developed using the same abstractions and, in principle, run the same way on the grid as on a single processor. There is presently no grid programming methodologies to deploy a metaprogram that will dynamically federate all needed services in the grid according to a control strategy aligned with the consistent *service algorithmic logic*. Applying the same programming abstractions to the grid as to a single computer does not foster transitioning from the current phase of early grid adopters to public recognition and then to mass adoption phases.

The reality at present is that grid resources are still very difficult for most users to access, and that detailed programming must be carried out by the user through command line and script execution to carefully tailor jobs on each end to the resources on which they will run, or for the data structure that they will access. This produces frustration for the user, delays in the adoption of grid techniques, and a multiplicity of specialized "grid-aware" tools that are not, in fact, aware of each other that defeat the basic purpose of the compute grid. Thinking more explicitly about grid programming languages than software languages may be our best tool for dealing with real world complexity. By understanding the principles that run across languages, appreciating which language traits are best suited for which type of application (service), and knowing how to craft the relevant infrastructure we can bring these languages to synergistic life and deal efficiently with the evolving complexity of distributed computing.

Instead of moving executable files around the compute grid, we can autonomically provision the corresponding computational components as uniform services on the grid. All grid services can be interpreted as instructions (metainstructions) of the *metacompute grid*. Now we can submit a metaprogram in terms of metainstructions to the *grid platform* that

manages a dynamic federation of service providers and related resources, and enables the metaprogram to interact with the service providers according to the metaprogram control strategy. We consider a *service* as interface type, for example identified as a Java interface. A provider can implement multiple interfaces, thus can provide multiple services. While grid computing is about utilizing virtual organization, metacomputing is about utilizing virtual processor with its instruction set in terms of services.

The term "metacomputing" was coined around 1987 by NCSA Director, Larry Smarr ("Metacomputing", n.d.). "The metacomputer is, simply put, a collection of computers held together by state-of-the-art technology and *balanced* so that, to the individual user, it looks and acts like a single computer. The constituent parts of the resulting *metacomputer* could be housed locally, or distributed between buildings, even continents ("Metacomputer", n.d.).

We can distinguish three types of grids depending on the nature of computational platforms: *compute grids* (*cGrids*), *metacompute grids* (*mcGrids*), and the hybrid of the previous two—*intergrids* (*iGrids*). Note that a cGrid is a virtual federation of processors (roughly CPUs) that execute submitted executable codes with the help of a grid resource broker. However, a mcGrid is a federation of service providers managed by the mcGrid operating system. Thus, the latter approach requires a metaprogramming methodology while in the former case the conventional procedural programming languages are used. The hybrid of both cGrid and mcGrid abstractions allows for an iGrid to execute both programs and metaprograms as depicted in Fig. 1, where platform layers P1, P2, and P3 correspond to resources, resource management, and programming environment correspondingly.

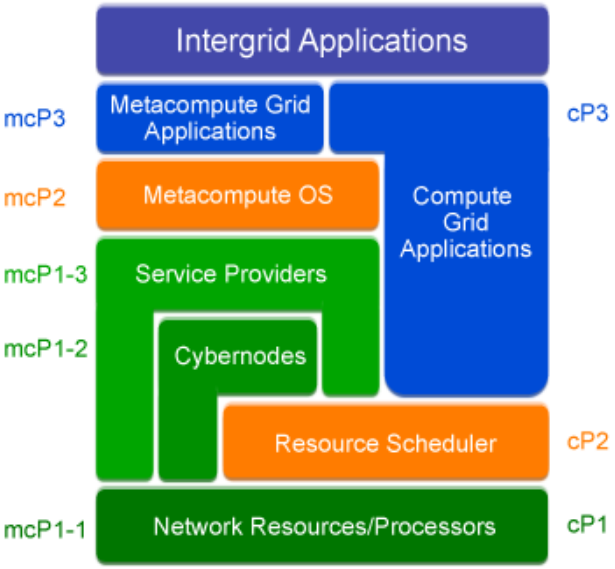

Fig. 1. Three types of grids: metacompute grid, compute grid, and intergrid. A cybernode provides a lightweight dynamic virtual processor, turning heterogeneous compute resources into homogeneous services available to the metacomputing OS ("Project Rio", n.d.)

One of the first mcGrids was developed under the sponsorship of the National Institute for Standards and Technology (NIST)—the Federated Intelligent Product Environment (FIPER)

(Röhl et al., 2000; Sobolewski, 2002). The goal of FIPER is to form a federation of distributed services that provide engineering data, applications, and tools on a network. A highly flexible software architecture had been developed (1999-2003), in which engineering tools like computer-aided design (CAD), computer-aided engineering (CAE), product data management (PDM), optimization, cost modeling, etc., act as federating service providers and service requestors.

The Service-ORiented Computing EnviRonment (SORCER) builds on the top of FIPER to introduce a metacomputing operating system with all system services necessary, including a federated file system and autonomic resource management, to support service-oriented metaprogramming. It provides an integrated solution for complex metacomputing applications. The SORCER metacomputing environment adds an entirely new layer of abstraction to the practice of grid computing—exertion-oriented (EO) programming with complementary federated method invocation. The EO programming makes a positive difference in service-oriented programming primarily through a new metaprogramming abstraction as experienced in many service-oriented computing projects including systems deployed at GE Global Research Center, GE Aviation, Air Force Research Lab, and SORCER Lab.

This chapter is organized as follows. Section 2 gives overview of RPC generations; Section 3 provides a brief description of two service-oriented architectures used in grid computing with a related discussion of distribution transparency; Section 4 describes the SORCER metacomputing philosophy and its federated method invocation; Section 5 describes the SORCER compute grid; Section 6 describes federated file system; Section 7 presents autonomic resource management; Section 8 explains the notion of intergrid and future development, and Section 9 provides concluding remarks.

2. Generations of Remote Procedure Call

Socket-based communication forces us to design distributed applications using a read/write (input/output) interface, which is not how we generally design non-distributed applications based on procedure call (request/response) communication. In 1983, Birrell and Nelson devised remote procedure call (RPC) (Birrell & Nelson, 1983), a mechanism to allow programs to call procedures on other hosts. So far, six RPC generations can be distinguished:

1. First generation RPCs—Sun RPC (ONC RPC) and DCE RPC, which are language, architecture, and OS independent;
2. Second generation RPCs—CORBA (Ruh & Klinker, 1999) and Microsoft DCOM-ORPC, which add distributed object support;
3. Third generation RPC—Java RMI (Pitt & McNiff, 2001) is conceptually similar to the second generation but supports the semantics of object invocation in different address spaces that are built for Java only. Java RMI fits cleanly into the language with no need for standardized data representation, external interface definition language, and with behavioral transfer that allows remote objects to perform operations that are determined at runtime;
4. Fourth generation RPC—next generation of Java RMI, Jini Extensible Remote Invocation ("Package net.jini.jeri", n.d) with dynamic proxies, smart proxies, network security, and

with dependency injection by defining exporters, end points, and security properties in deployment configuration files;
5. Fifth generation RPCs—Web/OGSA Services RPC (McGovern et al., 2003; Sotomayor & Childers, 2005) and the XML movement including Microsoft WCF/.NET;
6. Sixth generation RPC—Federated Method Invocation (FMI) (Sobolewski, 2007), allows for concurrent invocations on multiple federating compute resources (virtual metaprocessor) in the evolving SORCER environment (Sobolewski, 2008b).

All the RPC generations listed above are based on a form of service-oriented architecture (SOA) discussed in Section 3. However, CORBA, RMI, and Web/OGSA service providers are in fact object-oriented wrappers of network interfaces that hide object distribution and ignore the real nature of network through classical object abstractions that encapsulate network connectivity by using existing network technologies. The fact that object-oriented languages are used to create these object wrappers does not mean that developed distributed objects have a great deal to do with object-oriented distributed programming. For example, CORBA defines many services, and implementing them using distributed objects does not make them well structured with core object-oriented features: encapsulation, instantiation, and polymorphism. Similarly in Java RMI, marking objects with the Remote interface does not help to cope with network-centric messaging, for example when calling on a dead stub. Network centricity here means that sending a message to a remote object, in fact is sending it onto the network in the first place, and then dispatching it to a live remote object provided by the network in runtime and uniformly. Network-centric messaging should encapsulate object discovery, fault detection, recovery, partial failure, and others.

Each platform and its programming language used reflect a relevant abstraction, and usually the type and quality of the abstraction implies the complexity of problems we are able to solve. For example, a procedural language provides an abstraction of an underlying machine language. Building on the object-oriented distributed paradigm is the service object-oriented infrastructure exemplified by the Jini service architecture ("Jini architecture specification", n.d.; Edwards, 2000) in which the network objects come together on-the-fly to play their predefined roles. In the Service-ORiented Computing EnviRonmet (SORCER) developed at Texas Tech University ("SORCER Research Group", n.d.)), a service provider is a remote object that accepts network requests to participate in a collaboration—a process by which service providers work together to seek solutions that reach beyond what any one of them could accomplish on their own. While conventional objects encapsulate explicitly *data* and *operations*, the network requests called *exertions* encapsulate explicitly *data*, *operations*, and *control strategy*. An exertion can federate concurrently and transparently on multiple hosts according to its *control strategy* by hiding all low-level Jini networking details as well.

The SORCER metacomputing environment adds an entirely new layer of abstraction to the practice of grid computing—*exertion-oriented (EO) programming*. The EO programming makes a positive difference in service-oriented programming primarily through a new metacomputing platform as experienced in many grid-computing projects including applications deployed at GE Global Research Center, GE Aviation, Air Force Research Lab, and SORCER Lab. The new abstraction is about managing object-oriented distributed system complexity laid upon the complexity of the network of computers—metacomputer.

An exertion submitted onto the network dynamically binds to all relevant and currently available service providers in the object-oriented distributed system. The providers that dynamically participate in this invocation are collectively called an *exertion federation*. This

federation is also called a *virtual metaprocessor* since federating services are located on multiple processors held together by the EO infrastructure so that, to the requestor submitting the exertion, it looks and acts like a single processor.
The SORCER environment provides the means to create interactive EO programs (Sobolewski & Kolonay, 2006) and execute them using the SORCER runtime infrastructure presented in Section 4. Exertions can be created using interactive user agents downloaded on-the-fly from service providers. Using these interfaces, the user can create, execute, and monitor the execution of exertions within the EO platform. The exertions can be persisted for later reuse, allowing the user to quickly create new applications or EO programs on-the-fly in terms of existing, usually persisted for reuse exertions.
SORCER is based on the evolution of concepts and lessons learned in the FIPER project ("FIPER", n.d.), a $21.5 million program founded by NIST (National Institute of Standards and Technology). Academic research on FMI and EO programming has been established at the SORCER Laboratory, TTU, ("SORECE Research Group", n.d) where twenty-eight SORCER related research studies have been investigated so far ("SORCER Research Topics", n.d.).

3. SOA and Distribution Transparency

Various definitions of a Service-Oriented Architecture (SOA) leave a lot of room for interpretation. In general terms, SOA is a software architecture using loosely coupled software services that integrates them into a distributed computing system by means of service-oriented programming. Service providers in the SOA environment are made available as independent service components that can be accessed without a priori knowledge of their underlying platform or implementation. While the client-server architecture separates a client from a server, SOA introduces a third component, a service registry, as illustrated in Fig. 2 (the left chart). In SOA, the client is referred to as a service requestor and the server as a service provider. The provider is responsible for deploying a service on the network, publishing its service to one or more registries, and allowing requestors to bind and execute the service. Providers advertise their availability on the network; registries intercept these announcements and collect published services. The requestor looks up a service by sending queries to registries and making selections from the available services. Requestors and providers can use discovery and join protocols to locate registries and then publish or acquire services on the network.

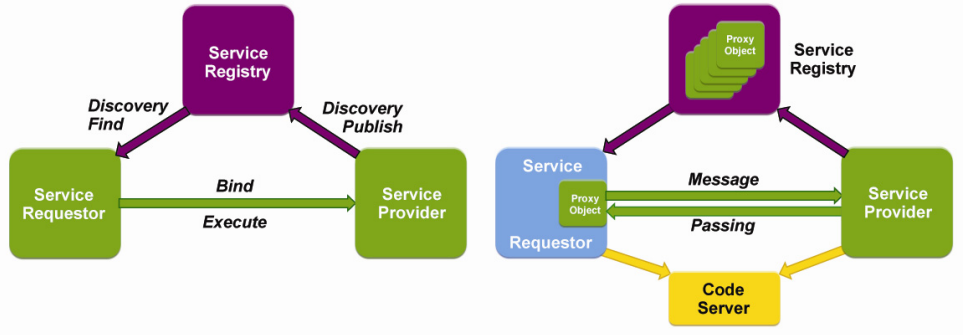

Fig. 2. SOA versus SOOA

We can distinguish the *service object-oriented architecture* (SOOA), where providers are network (*call/response*) objects accepting remote invocations, from the *service protocol oriented architecture* (SPOA), where a communication (*read/write*) protocol is fixed and known beforehand by the provider and requestor. Based on that protocol and a service description obtained from the service registry, the requestor can bind to the service provider by creating a proxy used for remote communication over the fixed protocol. In SPOA a service is usually identified by a name. If a service provider registers its service description by name, the requestors have to know the name of the service beforehand.

In SOOA, a proxy—an object implementing the same service interfaces as its service provider—is registered with the registries and it is always ready for use by requestors. Thus, in SOOA, the service provider publishes the proxy as the active surrogate object with a codebase annotation, e.g., URLs to the code defining proxy behavior (RMI and Jini ERI). In SPOA, by contrast, a passive service description is registered (e.g., an XML document in WSDL for Web/OGSA services, or an interface description in IDL for CORBA); the requestor then has to generate the proxy (a stub forwarding calls to a provider) based on a service description and the fixed communication protocol (e.g., SOAP in Web/OGSA services, IIOP in CORBA). This is referred to as a bind operation. The proxy binding operation is not required in SOOA since the requestor holds the active surrogate object obtained via the registry. The surrogate object is already bound to the provider that registered it with its appropriate network configuration and its code annotations.

Web services and OGSA services cannot change the communication protocol between requestors and providers while the SOOA approach is protocol neutral ("Waldo", n.d.). In SOOA, how the object proxy communicates with a provider is established by the contract between the provider and its published proxy and defined by the provider implementation. The proxy's requestor does not need to know who implements the interface or how it is implemented. So-called smart proxies (Jini ERI) can grant access to local and remote resources; they can also communicate with multiple providers on the network regardless of who originally registered the proxy. Thus, separate providers on the network can implement different parts of the smart proxy interface. Communication protocols may also vary, and a single smart proxy can also talk over multiple protocols including efficient application-specific protocols.

SPOA and SOOA differ in their method of discovering the service registry (see Fig. 2). SORCER uses dynamic discovery protocols to locate available registries (lookup services) as defined in the Jini architecture ("Jini architecture specification", n.d.). Neither the requestor who is looking up a proxy by its interfaces nor the provider registering a proxy needs to know specific locations. In SPOA, however, the requestor and provider usually do need to know the explicit location of the service registry—e.g., the IP address of an ONC/RPC portmapper, a URL for RMI registry, a URL for UDDI registry, an IP address of a COS Name Server—to open a static connection and find or register a service. In deployment of Web and OGSA services, a UDDI registry is sometimes even omitted when WSDL descriptions are shared via files; in SOOA, lookup services are mandatory due to the dynamic nature of objects identified by service types (e.g., Java interfaces). Interactions in SPOA are more like client-server connections (e.g., HTTP, SOAP, IIOP), often in deployment not requiring to use service registries at all.

Let us emphasize the major distinction between SOOA and SPOA: in SOOA, a proxy is created and always owned by the service provider, but in SPOA, the requestor creates and

owns a proxy which has to meet the requirements of the protocol that the provider and requestor agreed upon a priori. Thus, in SPOA the protocol is always a generic one, reduced to a common denominator—one size fits all—that leads to inefficient network communication in many cases. In SOOA, each provider can decide on the most efficient protocol(s) needed for a particular distributed application.

Service providers in SOOA can be considered as independent network objects finding each other via service registries and communicating through message passing. A collection of these objects sending and receiving messages—the only way these objects communicate with one another—looks very much like a service object-oriented distributed system.

However, do you remember the eight fallacies ("Fallacies", n.d.) of network computing? We cannot just take an object-oriented program developed without distribution in mind and make it a distributed system ignoring the unpredictable network behavior. Most RPC systems, except Jini, hide the network behavior and try to transform local communication into remote communication by creating distribution transparency based on a local assumption of what the network might be. However every single distributed object cannot do that in a uniform way as the network is a heterogeneous distributed system and cannot be represented completely within a single entity.

The network is dynamic, cannot be constant, and introduces latency for remote invocations. Network latency also depends on potential failure handling and recovery mechanisms, so we cannot assume that a local invocation is similar to remote invocation. Thus, complete distribution transparency—by making calls on distributed objects as though they were local—is impossible to achieve in practice. The distribution is simply not just an object-oriented implementation of a single distributed object; it is a metasystemic issue in object-oriented distributed programming. In that context, Web/OGSA services define distributed objects, but do not have anything common with object-oriented distributed systems that, for example, the Jini programming model and service architecture emphasize.

Object-oriented programming can be seen as an attempt to abstract both *data* representing a managed state of computational module and related *operations* in an entity called *object*. Thus, object-oriented program be seen as a collection of cooperating *objects* communicating via *message* passing, as opposed to a traditional view in which a program may be seen as a list of instructions to the computer. Instead of *objects* and *messages*, in EO programming *service providers* and *exertions* constitute a program. An *exertion* is a kind of meta-request sent onto the network. Thus, the exertion is considered as the *specification of a collaboration* that encapsulates *data* for the collaboration, related *operations,* and *control strategy*. The operations specify implicitly the required service providers on the network. The active exertion creates at runtime a federation of providers to execute service collaboration according to the exertion's control strategy. Thus, the active exertion is the *metaprogram* and its *metashell* (by analogy to the Unix shell, but here distributed) that submits the request onto the network to run the collaboration in which all providers pass to one other the component exertions only. This type of metashell was created for the SORCER metacompute operating system (see Fig. 3)—the exemplification of SOOA with autonomic management of system resources and domain-specific service providers to run EO programs.

No matter how complex and polished the individual operations are, it is often the quality of the *glue* that determines the power of the distributed computing system. SORCER defines the object-oriented distribution and glue for EO programming (Sobolewski, 2008a). It uses indirect federated remote method invocation (Sobolewski, 2007) with no location of service

provider explicitly specified in exertions. A specialized infrastructure of distributed services supports discovery/join protocols for the metashell, federated file system, autonomic resource management, and the rendezvous providers responsible for coordination of executing federations. The infrastructure defines SORCER's *object-oriented distributed* modularity, extensibility, and reuse of providers and exertions—key features of object-oriented distributed programming that are usually missing in SPOA programming environments. Object proxying with discovery/join protocols provides for provider protocol, location, and implementation neutrality missing in SPOA programming environments as well.

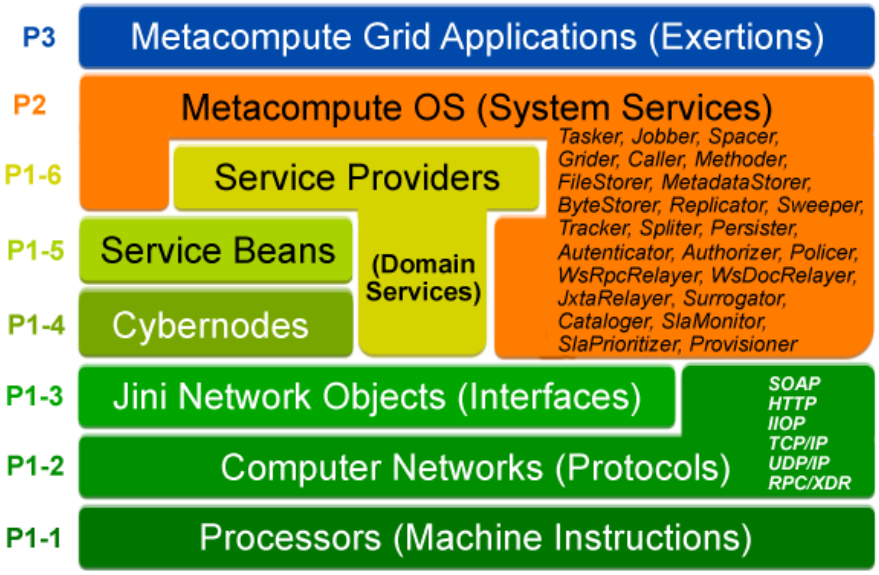

Fig. 3. SORCER layered platform, where P1 resources, P2 resource management, P3 programming environment

4. Metacompute Grid

SORCER is a federated service-to-service (S2S) metacomputing environment that treats service providers as network peers with well-defined semantics of a federated service object-oriented architecture (FSOOA). It is based on Jini semantics of services ("Jini Architecture", n.d.) in the network and the Jini programming model (Edwards, 2000) with explicit leases, distributed events, transactions, and discovery/join protocols. While Jini focuses on service management in a networked environment, SORCER is focused on EO programming and the execution environment for exertions (see Fig. 3).

An *exertion* is a metaprogram that specifies how a *collaboration* is realized by a collection (federation) of service providers and associations playing specific roles used in a specific way (Sobolewski, 2008c). An *exertion collaboration* specifies a view of cooperating providers and their services—a projection of service federation. It describes the required links between providers that play the roles of collaboration, as well as the attributes required that specify

the participating providers. Several exertions may describe different projections of the same collection of providers—*federation*. Please note that conventional objects encapsulate explicitly *data* and *operations*, but *exertions* encapsulate explicitly *data*, *operations*, and *control strategy*. The exertion participants in the federation collaborate transparently according to its *control strategy* managed by the SORCER metacompute OS based on the Triple Command Pattern presented at the end in this Section.

The exertion collaboration defines an *exertion interaction*. The *exertion interaction* specifies how invocations of operations are sent between service providers in a collaboration to perform a specific behavior. The interaction is defined in the context of exertion's control strategy. From the computing platform point of view, exertions are entities considered at the programming level, interactions at the operating system level, and federations at the processor level. Thus exertions are programs that define collaborations. The operating system manages collaborations as interactions in its virtual processor—the dynamically formed federations (see Fig. 3).

As described in Section 3, SOOA consists of four major types of network objects: providers, requestors, registries, and proxies. The provider is responsible for deploying the service on the network, publishing its proxy to one or more registries, and allowing requestors to access its proxy. Providers advertise their availability on the network; registries intercept these announcements and cache proxy objects to the provider services. The requestor looks up proxies by sending queries to registries and making selections from the available service types. Queries generally contain search criteria related to the type and quality of service. Registries facilitate searching by storing proxy objects of services and making them available to requestors. Providers use discovery/join protocols to publish services on the network; requestors use discovery/join protocols to obtain service proxies on the network. The SORCER metacompute OS uses Jini discovery/join protocols to implement its FSOOA.

In FSOOA, a service provider is an object that accepts exertions from service requestors to form a collaboration. An exertion encapsulates service data, operations, and control strategy. A *task exertion* is an elementary service request, a kind of elementary remote instruction (elementary statement) executed by a single service provider or a small-scale federation. A composite exertion called a *job exertion* is defined hierarchically in terms of tasks and other jobs, including control flow exertions. A job exertion is a kind of network procedure executed by a large-scale federation. Thus, the executing exertion is a service-oriented program that is dynamically bound to all required and currently available and on-demand provisioned, if needed, service providers on the network. This collection of providers identified at runtime is called an *exertion federation*. While this sounds similar to the object-oriented paradigm, it really is not. In the object-oriented paradigm, the object space is a program itself; here the exertion federation is the *execution environment* for the exertion, and the exertion is the *specification* of service collaboration. This changes the programming paradigm completely. In the former case a single computer hosts the object space, whereas in the latter case the parent and its component exertions along with bound service providers are hosted by the network of computers.

The overlay network of all service providers is called the *service grid* and an exertion federation is called a *virtual metaprocessor* (see Fig. 4). The *metainstruction set* of the metaprocessor consists of all operations offered by all providers in the grid. Thus, a service-oriented program is composed of metainstructions with its own service-oriented control strategy and service context representing the metaprogram data. Service signatures specify

metainstructions—collaboration participants in SORCER. Each signature primarily is defined by a service type (interface name), operation in that interface, and a set of optional attributes. Four types of signatures are distinguished: `PROCESS`, `PREPROCESS`, `POSTPROCESS`, and `APPEND`. A `PROCESS` signature—of which there is only one allowed per exertion—defines the dynamic late binding to a provider that implements the signature's interface. The service context (Zhao & Sobolewski, 2001; Sobolewski, 2008a) describes the data that tasks and jobs work on. An `APPEND` signature defines the context received from the provider specified by this signature. The received context is then appended in runtime to the service context later processed by `PREPROCESS`, `PROCESS`, and `POSTPROCESS` operations of the exertion. Appending a service context allows a requestor to use actual network data in runtime not available to the requestor when the exertion is submitted. A metacompute OS allows for an exertion to create and manage dynamic federation and transparently coordinate the execution of all component exertions within the federation. Please note that these metacomputing concepts are defined differently in traditional grid computing where a job is just an executing process for a submitted executable code with no federation being formed for the executable.

An exertion can be activated by calling exertion's `exert` operation:

```
Exertion.exert(Transaction):Exertion,
```

where a parameter of the `Transaction` type is required when a transactional semantics is needed for all participating nested exertions within the parent one. Thus, EO programming

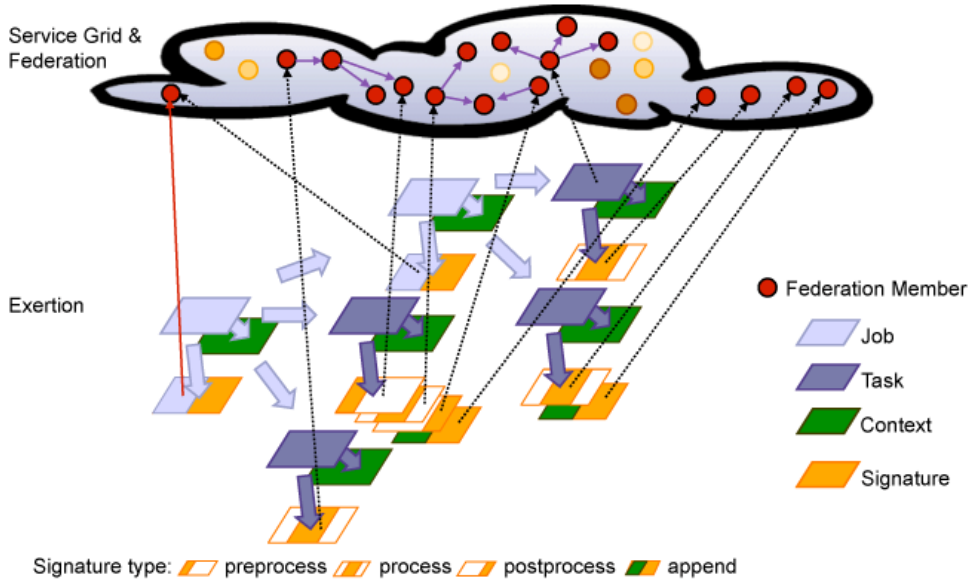

Fig. 4. An exertion federation. The solid line (the first from the left) indicates the originating invocation: `Exertion.exert(Transaction)`. The top-level exertion with component exertions is depicted below the service grid (a cloud). Late bindings to all participants defined by signatures are indicated by dashed lines. The participants form the exertion federation (metaprocessor).

allows us to submit an exertion onto the network and to perform executions of exertion's signatures on various service providers indirectly (see Fig. 4), 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, 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.

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 implicitly out onto the network—`Exertion.exert()` —and any service provider can pick it up. The provider can then look at the interface and operation requested within the exertion, and if it doesn't implement the desired interface or provide the desired method, it can continue forwarding it to another service 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 user 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 user would be able to pause a job between tasks, analyze it and make needed updates. To figure out where to resume an exertion, the executing provider would simply have to look at the exertion's completion states and resume the first one that wasn't completed yet. In other words, EO programming allows the user, *not programmer* to update the metaprogram on-the-fly, what practically translates into creating new collaborative applications during the exertion runtime.

Despite the fact that every Servicer can accept any exertion, Servicers have well defined roles in the S2S platform (see Fig. 3):

a) Taskers – process service tasks
b) Jobbers – process service jobs
c) Spacers – process tasks and jobs via exertion space for space-based computing (Freeman, 1999). See also secure space computing with exertions in (Kerr & Sobolewski, 2008).
d) Contexters – provide service contexts for `APPEND` Signatures
e) FileStorers – provide access to federated file system providers (Sobolewski, 2005; Berger, & Sobolewski, 2007) (see Section 6 for details)
f) Catalogers – Servicer registries
g) SlaMonitors – provide management of SLAs for QoS exertions (see Section 7);
h) Provisioners – provide on-demand provisioning of services by SERVME (see Section 7);
i) Persisters – persist service contexts, tasks, and jobs to be reused for interactive EO programming
j) Relayers – gateway providers; transform exertions to native representation, for example integration with Web services (McGovern et al., 2003) and JXTA ("JXTA", n.d.)
k) Autenticators, Authorizers, Policers, KeyStorers – provide support for service security
l) Auditors, Reporters, Loggers – support for accountability, reporting, and logging
m) Griders, Callers, Methoders – support compute grid (see Section 4)

n) Generic ServiceTasker, ServiceJobber, and ServiceSpacer implementations are used to configure domain-specific providers via dependency injection—configuration files for smart proxying and embedding business objects, called service beans, into service providers;
o) Notifiers – use third party services for collecting provider notifications for time consuming programs and disconnected requestors (Lapinski & Sobolewski, 2003).

An exertion can be created interactively (Sobolewski, 2006) or programmatically (using SORCER APIs), and its execution can be monitored and debugged (Soorianarayanan & Sobolewski, 2006) in the overlay service network via service user interfaces ("The Service UI Project", n.d.) attached to providers and installed on-the-fly by generic service browsers ("Inca X", n.d). Service providers do not have mutual associations prior to the execution of an exertion; they come together dynamically (federate) for all nested tasks and jobs in the exertion. Domain specific providers within the federation, or *task peers*, execute service tasks. Rendezvous peers coordinate exertion jobs: *Jobber* or *Spacer* are two of the SORCER platform control-flow services. However, a job can be sent to any peer. A peer that is not a rendezvous peer is responsible for forwarding the job to an available rendezvous peer and returning results to its direct requestor. Thus implicitly, any peer can handle any exertion type. Once the exertion execution is complete, the federation dissolves and the providers disperse to seek other exertions to join.

An Exertion is activated by calling its `exert` operation. The SORCER API defines the following three related operations:

1. `Exertion.exert(Transaction):Exertion` – join the federation; the activated exertion binds to the available provider specified by the exertion's `PROCESS` signature;
2. `Servicer.service(Exertion, Transaction):Exertion` – request a service in the federation initiated by any bounding provider; and
3. `Exerter.exert(Exertion, Transaction):Exertion` – execute the argument exertion by the provider accepting the service request in 2) above. Any component exertions of the exerted one are processed as in 1) above.

This above Triple Command pattern (Sobolewski, 2007) defines various implementations of these interfaces: `Exertion` (metaprogram), `Servicer` (generic peer provider), and `Exerter` (service provider exerting a particular type of `Exertion`). This approach allows for the P2P environment (Sobolewski, 2008a) via the `Servicer` interface, extensive modularization of Exertions and Exerters, and extensibility from the triple design pattern so requestors can submit onto the network any EO programs they want with or without transactional semantics. The Triple Command pattern is used as follows:

1. An exertion can be activated by calling `Exertion.exert()`. The `exert` operation implemented in `ServiceExertion` uses `ServiceAccessor` to locate in runtime the provider matching the exertion's `PROCESS` signature.
2. If the matching provider is found, then on its access proxy the `Servicer.service()` method is invoked.
3. When the requestor is authenticated and authorized by the provider to invoke the method defined by the exertion's `PROCESS` signature, then the provider calls its own `exert` operation: `Exerter.exert()`.
4. `Exerter.exert()` operation is implemented by `ServiceTasker`, `ServiceJobber`, and `ServiceSpacer`. The `ServiceTasker` peer calls by reflection the application method specified in the `PROCESS` signature of the task exertion. All application domain methods

of provider interface have the same signature: a single `Context` type parameter and a `Context` type return value. Thus an application interface implemented by the application provider looks like an RMI (Pitt, 2001) interface with the above simplification on the common signature for all interface operations.

The exertion activated by a service requestor can be submitted directly or indirectly to the matching service provider. In the direct approach, when signature's access type is `PUSH`, the exertion's `ServicerAccessor` finds the matching service provider against the service type and attributes of the `PROCESS` signature and submits the exertion to the matching provider. The execution order of signatures is defined by signature priorities, if the exertion's flow type is `SEQUENTIAL`; otherwise they are dispatched in parallel. EO programming has a branch exertion (`IfExertion`) and loop exertions (`WhileExertion`, `ForExertion`) as well as two mechanisms for nonlinear control flow (`BreakExertion`, `ContinueExertion`) (Sobolewski, 2008a).

A Job instance specifies a "block" of component tasks and other jobs. It is the distributed analog of a procedure in conventional programming languages. However, in EO programming it is a composite of exertions that makeup the network interaction. A Job can reflect a workflow with branching and looping by applying control flow exertions (Sobolewski, 2008a).

To illustrate a very flexible distributed control strategy of EO programs, let's consider the case presented in Fig. 5. This control strategy defines a virtual mapping of the control flow defined by the exertion e_1 onto the interactions of dynamically created federation. A rectangular frame outlines providers in that federation.

The following control flow exertions are defined in SORCER:

1. $|(e_1, \ldots, e_n)$ – sequential exertion;
2. $\|(e_1, \ldots, e_n)$ – parallel exertion;
3. $\dashv(e_1, e_2, e_3)$ – the `if` exertion: if e_1`.isTrue() then do` e_2 `else do` e_3; and
4. $*(e_1, e_2)$ or $*(e)$, if $e_1 = e_2$ and $e_1 = e$ – the `while` exertion: `do` e_2 `while` e_1`.isTrue()`.

Using the above notation the federation in Fig. 5 can be described by as follows: exertion $e_1 = |\,|(e_2, *(e_3))$, where $e_2 = |(e_4, e_5)$ and $e_5 = *(\dashv(e_6, e_7, e_8))$, and e_6 evaluates to true.

Alternatively, when signature's access type is `PULL`, a `ServiceAccessor` can use a `Spacer` provider and simply drops (writes) the exertion into the shared exertion space to be pulled by a matching provider. In Fig. 6 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. 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. 6. 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 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,

Spacers provide efficient load balancing for processing the exertion space. The fastest available servicer 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.

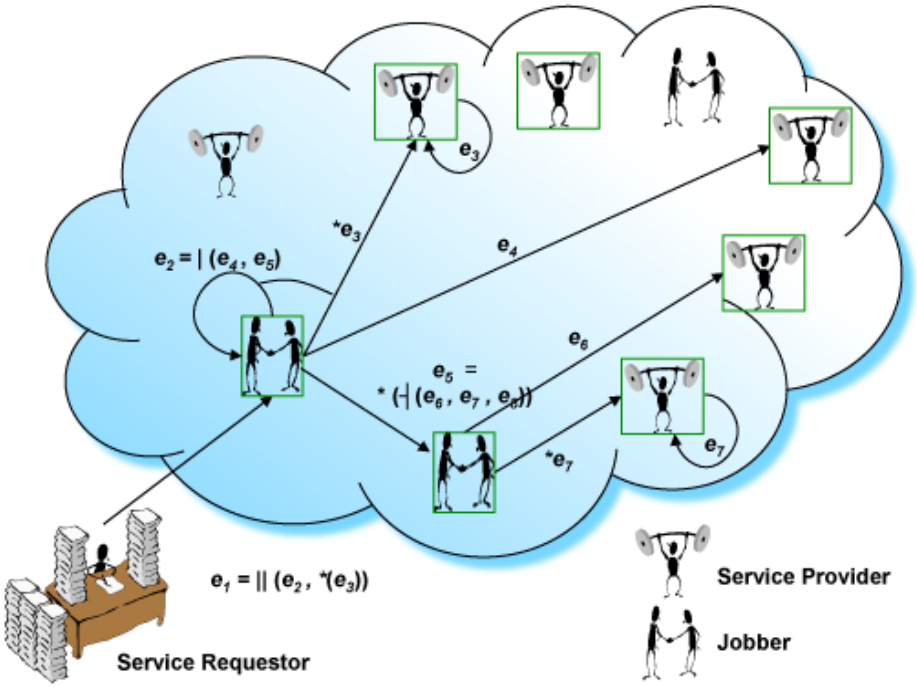

Fig. 5. SORCER metacompute OS finds the right service provider to whom a component exertion has to be bound as defined by its `PROCES` signature at runtime.

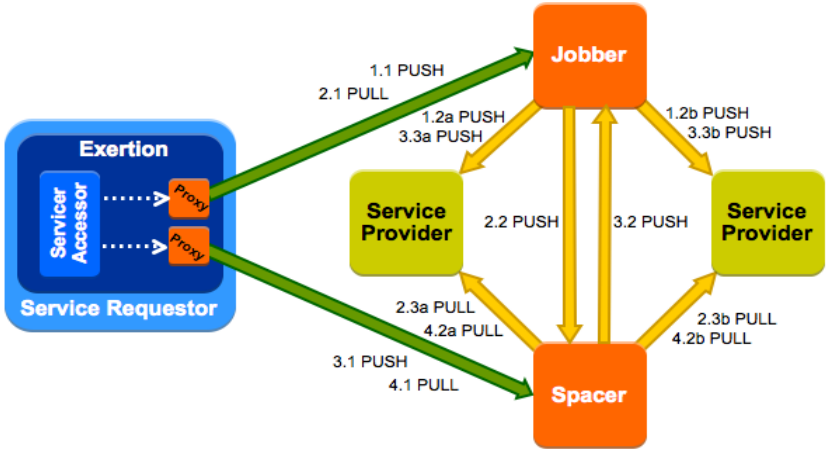

Fig. 6. Push vs. pull exertion processing

5. Compute Grid

To use legacy applications, SORCER supports a traditional approach to grid computing similar to those found in Condor (Thain, 2003) and Globus (Sotomayor, 2005). Here, instead of exertions being executed by services providing business logic for collaborating exertions, the business logic comes from the service requestor's executable codes that seek compute resources on the network.

The cGrid services in the SORCER environment include Griders accepting exertions and collaborating with Jobbers and Spacers as cGrid schedulers. Caller and Methoder services are used for task execution received from Jobbers or pulled up from exertion space via Spacers. Callers execute provided codes via a system call as described by the standardized Caller's service context of the submitted task. Methoders download required Java code (task method) from requestors to process any submitted service context with the downloaded code accordingly. In either case, the business logic comes from requestors; it is executable code specified in the service context invoked by Callers, or mobile Java code executed by Methoders that is annotated by the exertion signature. The architecture of the SORCER cGrid, called SGrid is depicted in Fig. 7.

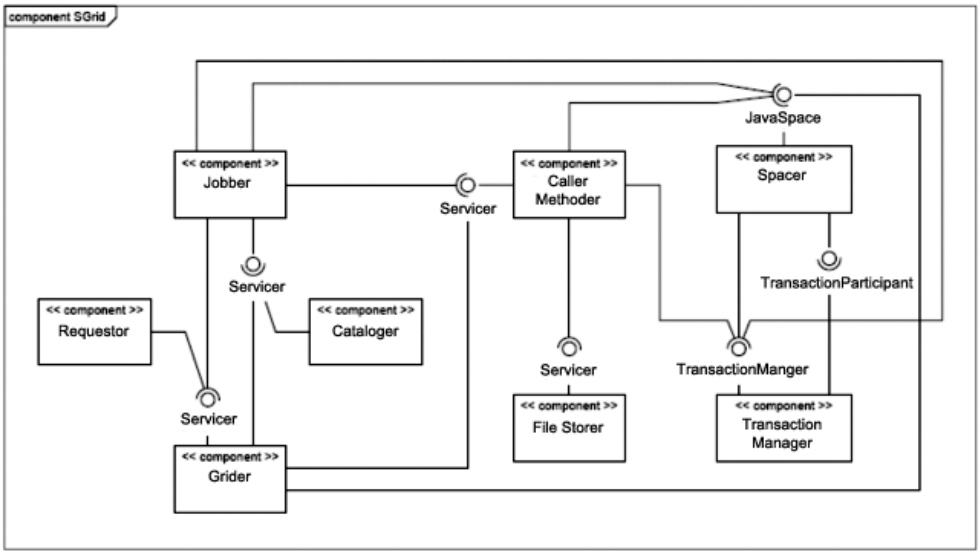

Fig. 7. SGrid component diagram

The SORCER cGrid with Methoders was used to deploy an algorithm called Basic Local Alignment Search Tool (BLAST) (Alschul, 1990) to compare newly discovered, unknown DNA and protein sequences against a large database with more than three gigabytes of known sequences. BLAST (C++ code) searches the database for sequences that are identical or similar to the unknown sequence. This process enables scientists to make inferences about the function of the unknown sequence based on what is understood about the similar sequences found in the database. Many projects at the USDA–ARS Research Unit, for example, involve as many as 10,000 unknown sequences, each of which must be analyzed via the BLAST algorithm. A project involving 10,000 unknown sequences requires about

three weeks to complete on a single desktop computer. The S-BLAST implemented in SORCER (Khurana et al., 2005), a federated form of the BLAST algorithm, reduces the amount of time required to perform searches for large sets of unknown sequences. S-BLAST is comprised of BlastProvider (with the attached BLAST Service UI), Jobbers, Spacers, and Methoders. Methoders in S-BLAST download Java code (a service task method) that initializes a required database before making a system call on the BLAST code. Armed with the S-BLAST's cGrid and seventeen commodity computers, projects that previously took three weeks to complete can now be finished in less than one day.
The SORCER cGrid with Griders, Jobbers, Spacers, and Callers has been successfully deployed with the Proth program (C code to search for large prime factors of Fermat numbers) and easy-to-use zero-install service UI attached to a Grider using the SILENUS federated file system.

6. Federated File System

The SILENUS federated file system (Berger & Sobolewski, 2005; Berger & Sobolewski, 2007a) was developed to provide data access and persistence storage for metaprograms. It complements the centralized file store developed for FIPER (Sobolewski et al., 2003) with the true P2P services. The SILENUS system itself is a collection of service providers that use the SORCER exertion-oriented framework for communication.

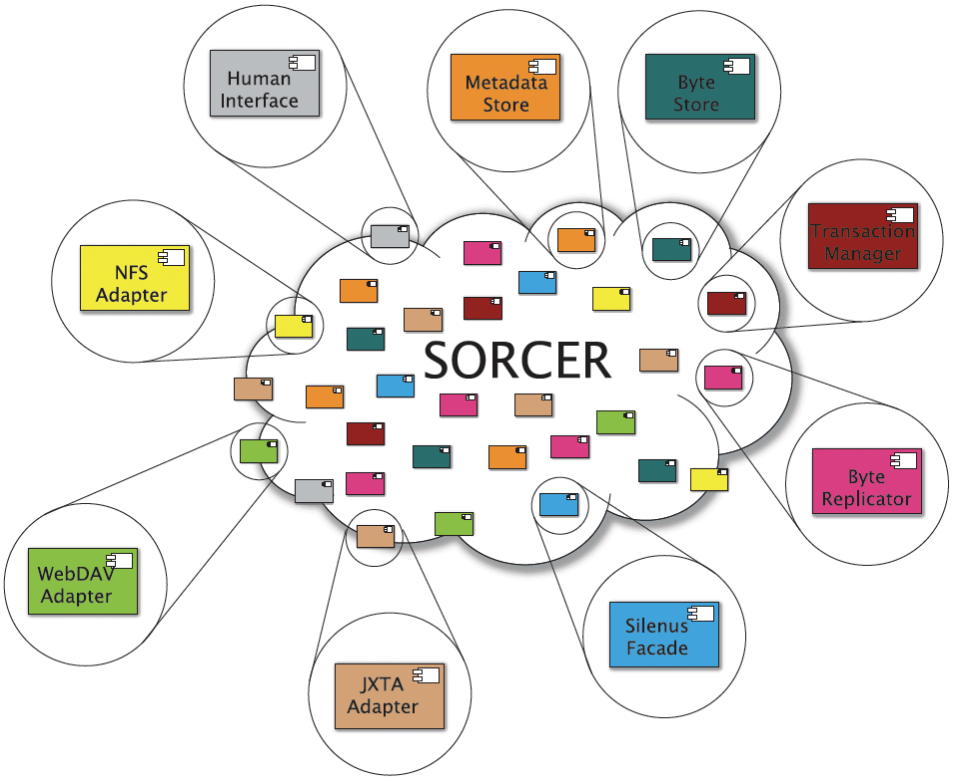

Fig. 8. SILENUS file system components for the SORCER platform

In classical client-server file systems, a heavy load may occur on a single file server. If multiple service requestors try to access large files at the same time, the server can be overloaded. In P2P architecture, every provider is a client and a server at the same time. The load can be balanced between all peers if files are spread across all of them. The SORCER architecture splits up the functionality of the metaprocessor into smaller service peers (Servicers), and this approach was applied to the distributed file system as well.

The SILENUS federated file system is comprised of several network services that run within the SORCER environment as illustrated in Fig. 8.

These services include a byte store service for holding file data, a metadata service for holding metadata information about the files, several optional optimizer services, and façade (Grand, 1999) services to assist in accessing federating services. SILENUS is designed so that many instances of these services can run on a network, and the required services will federate together to perform the necessary functions of a file system. In fact the SILENUS system is completely decentralized, eliminating all potential single points of failures. SILENUS services can be broadly categorized into gateway components, data services, and management services. Each byte store provides fast access to the underlying native file system on the provider's host while each metadata provider allows creating, listing, and traversing directories persisted in the provider's embedded relational database. All metadata databases are synchronized in runtime for all file modifications and updates.

The SILENUS façade service provides a gateway service to the SILENUS federations for requestors that want to use the file system. Since the metadata and actual file contents are stored by different services, there is a need to coordinate communication between these two services. The façade service itself is a combination of a control component, called the coordinator, and a smart proxy component that contains needed inner proxies provided dynamically by the coordinator. These inner proxies facilitate direct P2P communications for file upload and download between the requestor and SILENUS federating services such as metadata and byte stores, if needed with the participation of the Jini transaction manager when transactional semantics is required for updates to both metadata and byte store service concurrently.

Core SILENUS services have been successfully deployed as SORCER services along with WebDAV and NFS adapters. The SILENUS file system scales very well with a virtual disk space adjusted as needed by the corresponding number of required byte store providers and the appropriate number of metadata stores required to satisfy the needs of current users and service requestors. The system handles several types of network and computer outages by utilizing disconnected operation and data synchronization mechanisms. It provides a number of user agents including a zero-install file browser attached to the SILENUS façade. Also a simpler version of SILENUS file browser is available for smart MIDP phones.

In most file systems it is impossible or impracticable to ask the user which conflicting option to choose. The SILENUS file system provides support for disconnected operation. A new dual-time vector clock based synchronization mechanism (Berger & Sobolewski, 2007b) detects and orders events properly. It detects also possible conflicts and resolves them in a consistent manner without user interactions. To solve a conflict, the SILENUS system uses virtual duplication. Virtual duplication addresses the issue of local consistency and requires no direct user interaction. The approach based on dual-time vector clock synchronization provides complete and consistent support for dynamically federating services and changing overlay networks.

The FICUS framework (Turner & Sobolewski, 2007) is an extension to SILENUS. FICUS supports storing very large files (Turner & Sobolewski, 2007) by providing two services: a splitter service and a tracker service. When a file is uploaded to the file system, the splitter service determines how that file should be stored. If a file is sufficiently large, the file will be split into multiple parts, or chunks, and stored across many byte store services. Once the upload is complete, a tracker service keeps a record of where each chunk was stored. When a user requests to download the full file later on, the tracker service can be queried to determine the location of each chunk and the file can be reassembled in parallel to the original form.

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. A separate framework called LOCO (Hard & Sobolewski, 2009) has been developed to dynamically manage replicas 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). The LOCO framework is an extension to SILENUS and is comprised of four services: a Locator service, Sweeper service, Replicator service, and a Resource Usage Store service.

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.

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.

7. Autonomic Resource Management

Federated computing environments offer requestors the ability to dynamically invoke services offered by collaborating providers in the entire service grid. Without an efficient resource management, however, the assignment of providers to customer's requests cannot be optimized and cannot offer high reliability without relevant SLA guarantees. A SLA-based SERViceable Metacomputing Environment (SERVME) (Rubach & Sobolewski, 2009) capable of matching providers based on QoS requirements and performing autonomic provisioning and deprovisioning of services according to dynamic requestor needs has beed developed for the SORCER metaoperating system. In SERVME an exertion signature includes an SLA Context that encapsulates all QoS/SLA related data. SERVME builds on the SORCER environment by extending its interfaces and adding new QoS/SLA service

providers. It is a generic resource management framework in terms of common QoS/SLA data structures and extensible communication interfaces hiding all implementation details.
Along with the QoS/SLA object model SERVME defines basic components and communication interfaces as depicted in the UML component diagram illustrated in Fig. 9. We distinguish two forms of autonomic provisioning: monitored and on-demand. In monitored provisioning the provisioner (Rio Provisioner (Rio Project, n.d.)) deploys a requested collection of providers, then monitors them for presence and in the case of any failure in the collection, the provisioner makes sure that the required number of providers is always on the network as defined by a provisioner's deployment descriptor. On-demand provisioning refers to a type of provisioning (On-demand Provisioner) where the actual provider is presented to the requestor, once a subscription to the requested service is successfully processed. In both cases, if services become unavailable, or fail to meet processing requirements, the recovery of those service providers to available compute resources is enabled by Rio provisioning mechanisms.
The basic components are defined as follows:

- QosProviderAccessor is a component used by the service requestor (customer) that is responsible for processing the exertion request containing QosContext in its signature. If the exertion type is Task then QosCatalog is used, otherwise a relevant rendezvous peer: Jobber, Spacer is used.
- QosCatalog is an independent service that acts as an extended Lookup Service (QoS LUS). The QosCatalog uses the functional requirements as well as related non-functional QoS requirements to find a service provider from currently available in the network. If a matching provider does not exist, the QosCatalog may provision the needed one.
- SlaDispatcher is a component built into each service provider. It performs two roles. On one hand, it is responsible for retrieving the actual QoS parameter values from the operating system in which it is running, and on the other hand, it exposes the interface used by QosCatalog to negotiate, sign and manage the SLA with its provider.
- SlaPrioritizer is a component that allows controlling the prioritization of the execution of exertions according to the organizational requirements of SlaContext.
- QosMonitor UI provides an embedded GUI that allows the monitoring of provider's QoS parameters at runtime.
- SlaMonitor is an independent service that acts as a registry for negotiated SLA contracts and exposes the user interface (UI) for administrators to allow them to monitor, update or cancel active SLAs.
- On-demandProvisioner is a SERVME provider that enables on-demand provisioning of services in cooperation with the Rio Provisioner ("Rio", n.d.) The QosCatalog uses it when no matching service provider can be found that meets requestor QoS requirements. We distinguish two forms of autonomic provisioning: monitored and on-demand. In monitored provisioning the Rio Provisioner deploys a requested collection of providers, then monitors them for presence and in the case of any failure in the collection, the Provisioner makes sure that the required number of providers is always on the network as defined by a provisioner's deployment descriptor. On-demand provisioning refers to a type of provisioning (On-demand Provisioner) when the actual provider is presented to the requestor, once a subscription to the requested service is successfully processed. In both cases, if services become unavailable, or fail to meet

processing requirements, the recovery of those service providers to available compute resources is enabled by Rio Project provisioning mechanisms.

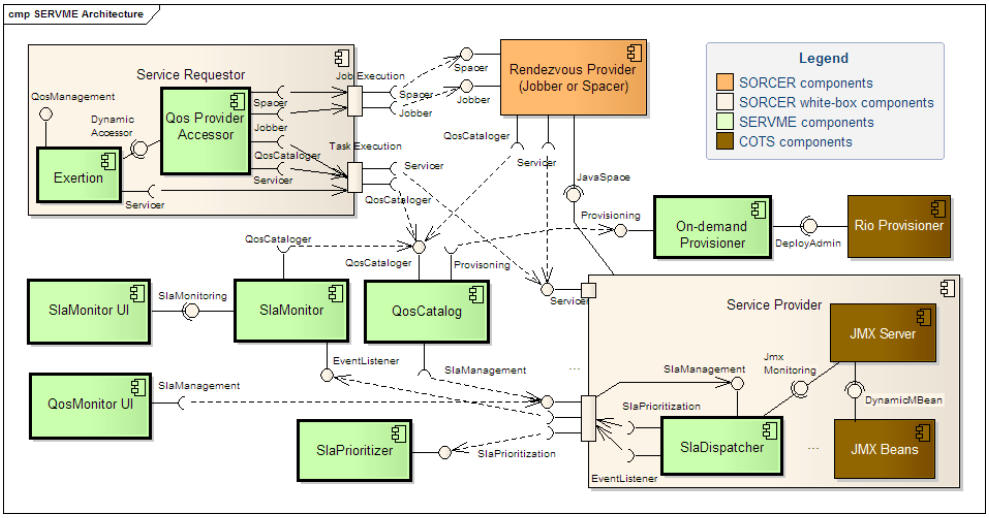

Fig. 9. SERVEME architecture

The SERVME framework is integrated directly into the federated metacomputing environment. As described in Section 4, the service requestor submits the exertion with QoS requirements (QosContext) into the network by invoking `Exertion.exert()` operation. If the exertion is of Task type, then QosProviderAccessor via QosCalatog finds in runtime a matching service provider with a corresponding SLA.
If the SLA can be directly provided then the contracting provider approached by the QosCalatog returns it in the form of SlaContext, otherwise a negotiation can take place for the agreeable SlaContext between the requestor and provider. The provider's SlaDispatcher drives this negotiation in cooperation with SlaPrioritizer and the exertion's requestor.
If the task contains multiple signatures then the provider is responsible for contracting SLAs for all other signatures of the task before the SLA for its `PROCESS` signature is guaranteed.
However, if the submitted exertion is of Job type, then QosProviderAccessor via QosCalatog finds in runtime a matching rendezvous provider with a guaranteed SLA. Before the guaranteed SLA is returned, the rendezvous provider recursively acquires SLAs for all component exertions as described above depending on the type (Task or Job) of component exertion.

8. SORCER iGrid and Future Development

In Section 4 and 5 two complementary platforms: metacompute grid and compute grid are described respectively. As indicated in Fig. 1 the hybrid of both types of grids is feasible to create *intergrid* (iGrid) applications that take advantage of both platforms synergistically. Legacy applications can be reused directly in cGrids and new complex, for example concurrent engineering applications (Sobolewski & Ghodous, 2005) can be defined in mcGrids, for example using EO programming.

Relayers are SORCER gateway providers that transform exertions to native representations and vice versa. The following exertion gateways have been developed: JxtaRelayer for JXTA ("JXTA", n.d.), and WsRpcRelayer and WsDocRelayer for for RPC and document style Web services, respectively (SORCER Research Topics, n.d.). Relayers exhibit native and mcGrid behavior by implementing dual protocols. For example a JxtaRelayer (1) in Fig. 10 is at the same time a Servicer (1-) in the mcGrid and a JXTA peer (1--) implementing JXTA interfaces. Thus it shows up also in SORCER mcGrid and in the JXTA cGrid as well. Native cGrid providers can play the SORCER role (as SORCER wrappers), thus are available in the iGrid along with mcGrid providers. For example, a JAXTA peer 4-- implements the Servicer interface, so shoes up in the JXTA iGrid as provider 4. Also, native cGrid providers via corresponding relayers can access iGrid services (bottom-up in Fig. 10). Thus, the iGrid is a projection of Servicers onto mcGrids and cGrds.

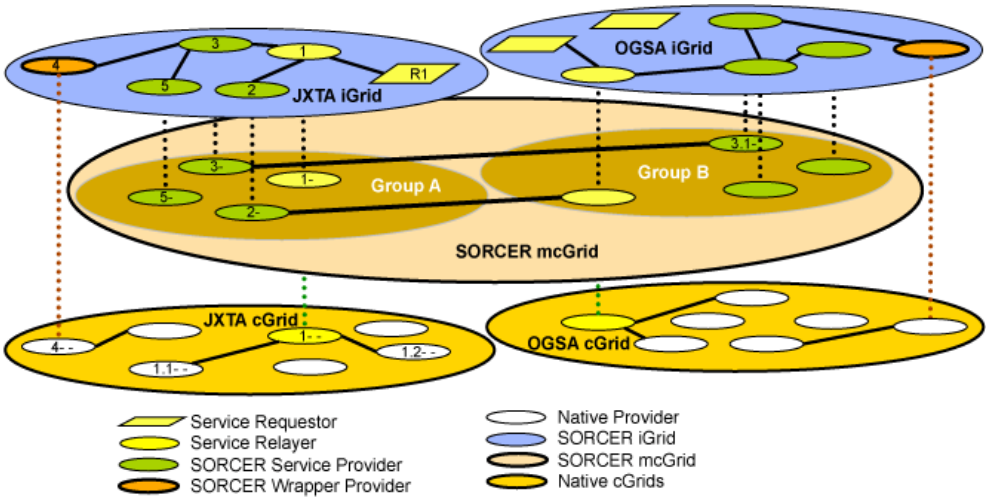

Fig. 10. Integrating and wrapping cGrids with SORCER mcGrids. Two requestors, one in JXTA iGrid, one in OGSA iGrid submits exertion to a corresponding relayer. Two federations are formed that include providers from the two horizontal layers below the iGrid layer (as indicated by continuoues and dashed links).

The iGrid-integrating model is illustrated in Fig. 10, where horizontal native technology grids (bottom) are seamlessly integrated with horizontal SORCER mcGrids via the SORCER operating system services. Through the use of open standards-based communication—Jini, Web Services, Globus/OGSA, and Java interoperability—iGrid leverages mcGrid's FSOOA with its inherent provider proxy protocol, location, and implementation neutrality along with the flexibility of EO programming with its complementary metacompute federated OS. To clarify iGrid interactions, let's consider the requestor R1 submitting the exertion to provider 1. The JxtaRelayer 1 is a version of Jobber that is able to route exertions directly to JXTA peers. At the level of iGrid a federation consisting of five providers is formed (1,2, 3, 4, and 5). The relayer interacts with two JXTA peers 1.1-- and 1.2-- for two-component exertion from R1. Providers 2, 3, and 5 are projected into the SORCER mcGrid, and the provider 3- collaborates with the provider 3.1-. The native JXA peer 4-- is used via the Servicer

wrapper 4. Thus, the federation of five iGrid services (1, 2, 3, 4, and 5) projects a federation of ten federating services (1-, 1--, 1.1--, 1.2--, 2-, 3-, 3.1-, 4, 4--, and 5-) in the SORCER mcGrid and JXTA cGrid. In fact, service 1-, 2-, 3-, and 5- are the same Servicers as services 1, 2, 3, and 5 correspondingly.

9. Conclusions

A distributed system is not just a collection of static distributed objects—it is the network of dynamic objects that come and go. From the object-oriented point of view, the network of dynamic objects is the *problem domain* of object-oriented distributed system that requires relevant abstractions in the *solution space*—metacomputing with FMI. The exertion-based programming introduces the new abstraction of the solution space with *service providers* and *exertions* instead of object-oriented conventional *objects* and *messages*. Exertions not only encapsulate operations, data, and control strategy, they encapsulate related collaborations of dynamic service providers as well. From the metacomputing platform point of view, exertions are entities considered at the programming level, collaborative interactions at the operating system level, and federations at the processor level. Thus, exertions are programs that define dynamic collaborations. The SORCER operating system manages collaborations as interactions in its virtual processor—the dynamically formed federations that use FMI.

Service providers can be easily deployed in SORCER by injecting implementation of domain-specific interfaces into the FMI framework. The providers register proxies, including smart proxies, via dependency injection using twelve methods investigated already. Executing a top-level exertion, by sending it onto the network, means forming a federation of currently available and on-demand provisioned, if needed, domain-specific providers at runtime. The federation processes service contexts of all nested exertions collaboratively as specified by control strategies of the top-level and component exertions. The fact that control strategy is exposed directly to the user in a modular way allows him/her to create new applications on-the-fly. For the updated control strategy only, the new federation becomes the new implementation of the updated exertion—a truly creative metaprogramming. 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 providers via service object proxies that in fact are access proxies allowing for service providers to enforce security policies on access to required services. If the access to use the operation is granted, then the operation defined by an exertion's `PROCESS` signature is invoked by reflection.

The FMI framework allows the P2P computing via the Servicer interface, extensive modularization of Exertions and Exerters, and extensibility from the Triple Command design pattern. The presented EO programming methodology with SORCER metacompute OS with its federated file system (SILENUS/FICUS/LOCO) and resource management (SERVME) has been successfully deployed and tested in multiple concurrent engineering and large-scale distributed applications (Röhl et al., 2000; Burton et al., 2002; Kolonay et al., 2002; Kao et al., 2003; Goel & Sobolewski, 2005; Goel et al., 2007; Kolonay et al., 2007; Goel et al., 2008).

To work effectively in large, distributed environments, concurrent engineering teams need a service-oriented programming methodology along with common design process, discipline-independent representations of designs, and general criteria for decision making.

Distributed multidisciplinary analysis and optimization are essential for decision making in engineering design that provide a foundation for service-oriented concurrent engineering (Sobolewski & Ghodous, 2005). It is believed that incremental improvements will not suffice, and so we plan to continue the development of *Service-Oriented Optimization Toolkit with EO Programming for Distributed High Fidelity Engineering Design Optimization* as the validation test case for SORCER iGrid. This approach brings together many ideas and results from twenty eight research studies completed at the SORCER Lab and AFRL, to investigate how *Variable-Filter-Evaluation Design Pattern* for distributed functional composition, comprehensive security, and *Dynamic Proxying with Dependency Injection* can be used to address several fundamental challenges posed by the emerging metacomputing based on FMI for *Distributed High Fidelity Engineering Design Optimization* in real world complex and high performance computing applications.

10. Acknowledgments

This work was partially supported by Air Force Research Lab, Air Vehicles Directorate, Multidisciplinary Technology Center, the contract number F33615-03-D-3307, Algorithms for Federated High Fidelity Engineering Design Optimization. I would like to express my gratitude to all those who helped me in my SORCER research. I would like to thank all my colleagues at GE Global Research Center, AFRL/RBSD, and my students at the SORCER Lab, TTU. They shared their views, ideas, and experience with me, and I am very thankful to them for that. Especially I would like to express my gratitude to Dr. Ray Kolonay, my technical advisor at AFRL/RBSD for his support, encouragement, and advice.

11. References

Berger, M. & Sobolewski, M. (2005). SILENUS – A Federated Service-oriented Approach to Distributed File Systems, In: *Next Generation Concurrent Engineering*, Sobolewski, M., and Ghodous, P. (Ed.), pp. 89-96, ISPE Inc./Omnipress, ISBN 0-9768246-0-4

Berger, M. & Sobolewski, M. (2007a). Lessons Learned from the SILENUS Federated File System, In: *Complex Systems Concurrent Engineering*, Loureiro, G. and L.Curran, R. (Ed.), pp. 431-440, Springer Verlag, ISBN: 978-1-84628-975-0

Berger M. & Sobolewski, M. (2007b). A Dual-time Vector Clock Based Synchronization Mechanism for Key-value Data in the SILENUS File System, *IEEE Third International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems* (SRMPDS '07), Hsinchu, Taiwan

Birrell, A. D. & Nelson, B. J. (1983). Implementing Remote Procedure Calls, XEROX CSL-83-7

Burton, S. A.; Tappeta, R.; Kolonay, R. M. & Padmanabhan, D (2002). Turbine Blade Reliability-based Optimization Using Variable-Complexity Method, 43rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, Denver, Colorado. AIAA 2002-1710

Edwards W.K. (2000) *Core Jini*, 2nd ed., Prentice Hall

Fallacies of Distributed Computing. Accessed on: April 24, 2009. Available at: http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing

FIPER: Federated Intelligent Product EnviRonmet. Available at: http://sorcer.cs.ttu.edu/fiper/fiper.html. Accessed on: April 24, 2009.

Foster I.; Kesselman C. & Tuecke S. (2001). The Anatomy of the Grid: Enabling Scalable Virtual Organizations, *International J. Supercomputer Applications*, 15(3)

Freeman, E.; Hupfer, S. & Arnold, K. *JavaSpaces™ Principles, Patterns, and Practice*, Addison-Wesley, ISBN: 0-201-30955-6

Goel, S.; Talya S. & Sobolewski, M. (2005). Preliminary Design Using Distributed Service-based Computing, *Proceeding of the 12th Conference on Concurrent Engineering: Research and Applications*, In: *Next Generation Concurrent Engineering*, Sobolewski, M., and Ghodous, P. (Ed.), pp. 113-120, ISPE Inc./Omnipress, ISBN 0-9768246-0-4

Goel S., Shashishekara, Talya S.S., Sobolewski M. (2007). Service-based P2P overlay network for collaborative problem solving, *Decision Support Systems*, Volume 43, Issue 2, March 2007, pp. 547-568

Goel, S.; Talya, S.S. & Sobolewski, M. (2008). Mapping Engineering Design Processes onto a Service-Grid: Turbine Design Optimization, *International Journal of Concurrent Engineering: Research & Applications*, Concurrent Engineering, Vol.16, pp 139-147

Hard, C. & Sobolewski, M (2009). File Location Management in Federated Computing Environments, *International Journal of Recent Trends in Engineering (Computer Science)*, Vol. 1, No. 1, June 2009, pp. 512-517.

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

Inca X™ Service Browser for Jini Technology. Available at: http://www.incax.com/index.htm?http://www.incax.com/service-browser.htm. Accessed on: April 24, 2009.

JXTA. Available at: https://jxta.dev.java.net/. Accessed on: April 24 2009.

Jini architecture specification, Version 2.1. Available at: http://www.sun.com/software/jini /specs/jini1.2html/jini-title.html. Accessed on: January 15, 2008(2001)

Kao, K. J.; Seeley, C.E.; Yin, S.; Kolonay, R.M.; Rus, T. & Paradis, M.J. (2003). Business-to-Business Virtual Collaboartion of Aircraft Engine Combustor Design, Proceedings of *DETC'03* ASME 2003 Design Engineering Technical Conferences and Computers and Information in Engineering Conference, Chicago, Illinois

Kerr, D. & Sobolewski, M. (2008). Secure Space Computing with Exertions, *3rd Annual Symposium on Information Assurance* (ASIA'08), Albany, NY.

Khurana, V.; Berger, M. & Sobolewski, M. (2005). A Federated Grid Environment with Replication Services. *Next Generation Concurrent Engineering*. ISPE/Omnipress, ISBN 0-9768246-0-4, pp. 97-103.

Kolonay, R.M.; Sobolewski; M., Tappeta, R.; Paradis. M. & Burton, S. (2002). Network-Centric MAO Environment. The Society for Modeling and Simulation International, Westrn Multiconference, San Antonio, TX

Kolonay, R. M.; Thompson, E.D.; Camberos, J.A. & Eastep, F. (2007). Active Control of Transpiration Boundary Conditions for Drag Minimization with an Euler CFD Solver, AIAA-2007-1891, 48th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, Honolulu, Hawaii

Lapinski, M. & Sobolewski, M. (2003). Managing Notifications in a Federated S2S Environment, *International Journal of Concurrent Engineering: Research & Applications*, Vol. 11, pp. 17-25

Metacomputing: Past to Present, Retrieved April 24, 2009, from: http://archive.ncsa.uiuc.edu/Cyberia/MetaComp/MetaHistory.html

Metacomputer: One from Many, April 24, 2009, from: http://archive.ncsa.uiuc.edu/Cyberia/MetaComp/MetaHome.html

McGovern J.; Tyagi S.; Stevens M.E. & Mathew S. (2003). Morgan Kaufmann

Nimrod: Tools for Distributed Parametric Modelling. Retrieved April 24, 2009, from: http://www.csse.monash.edu.au/~davida/nimrod/nimrodg.htm.

Package net.jini.jeri. Available at: http://java.sun.com/products/jini/2.1/doc/api/net/jini/jeri/package-summary.html. Accessed on: April 24, 2009

Pitt E. & McNiff K. (2001). *java.rmi: The Remote Method Invocation Guide*, Addison-Wesley Professional

Project Rio, A Dynamic Service Architecture for Distributed Applications. Available at: https://rio.dev.java.net/. Accessed on: April 24, 2009

Röhl, P.J.; Kolonay, R.M.; Irani, R.K.; Sobolewski, M. & Kao, K. (2000). A Federated Intelligent Product Environment, AIAA-2000-4902, 8th AIAA/USAF/NASA/ ISSMO Symposium on Multidisciplinary Analysis and Optimization, Long Beach, CA

Rubach, P. & Sobolewski, M. (2009). Autonomic SLA Management in Federated Computing Environments, *Proceedings of the 2009 International Conference on Parallel Processing Workshops (ICPPW 2009)*, Vienna, Austria. IEEE Computer Society, ISBN 978-0-7695-3803-7, pp. 314-321.

Ruh W.A.; Herron T. & Klinker P. (1999). IIOP Complete: Understanding CORBA and Middleware Interoperability, Addison-Wesley

Sampath, R.; Kolonay, R. M. & Kuhne, C. M. (2002). 2D/3D CFD Design Optimization Using the Federated Intelligent Product Environment (FIPER) Technology. AIAA-2002-5479, 9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Atlanta, GA

Sobolewski M. (2002). Federated P2P services in CE Environments, Advances in Concurrent Engineering, A.A. Balkema Publishers, 2002, pp. 13-22

Sobolewski, M., Soorianarayanan, S., Malladi-Venkata, R-K. (2003). Service-Oriented File Sharing, Proceedings of the IASTED Intl., Conference on Communications, Internet, and Information technology, pp. 633-639, ACTA Press

Sobolewski, M. & Ghodous, P. (Ed.). (2005). *Next Generation Concurrent Engineering: Smart and Concurrent Integration of Product Data, Services, and Control Strategies,* ISPE, Inc., ISBN 0-9768246-0-4

Sobolewski M., Kolonay R. (2006). Federated Grid Computing with Interactive Service-oriented Programming, International Journal of Concurrent Engineering: Research & Applications, Vol. 14, No 1, pp. 55-66

Sobolewski M. (2007). Federated Method Invocation with Exertions, Proceedings of the IMCSIT Conference, PTI Press, ISSN 1896-7094, pp. 765-778

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

Sobolewski, M (2008b). SORCER: Computing and Metacomputing Intergrid, 10th International Conference on Enterprise Information Systems, Barcelona, Spain (2008).

Sobolewski, M. 2008c). Federated Collaborations with Exertions, 17h IEEE International Workshop on Enabling Technologies: Infrastructures for Collaborative Enterprises, Rome, Italy

Soorianarayanan, S. & Sobolewski, M. (2004). Monitoring Federated Services in CE, *Concurrent Engineering: The Worldwide Engineering Grid*, Tsinghua Press and Springer Verlag, pp. 89-95

SORCER Research Group. Available at: http://sorcer.cs.ttu.edu/ or http://sorcersoft.org. Accessed on: April 24, 2009.

SORCER Research Topics. Available at: http://sorcer.cs.ttu.edu/theses/ or http://sorcersoft.org/theses/. Accessed on: April 24, 2009

Sotomayor B. & Childers L. (2005). *Globus® Toolkit 4: Programming Java Services*, Morgan Kaufmann

Thain D.; Tannenbaum T.; Livny M. (2003). Condor and the Grid. In: Grid Computing: Making The Global Infrastructure a Reality, Fran Berman, Anthony J.G. Hey, and Geoffrey Fox, (Ed.),. John Wiley

Turner, A. & Sobolewski, M. (2007). FICUS—A Federated Service-Oriented File Transfer Framework, *Complex Systems Concurrent Engineering*, Loureiro, G. and L.Curran, R. (Ed.). Springer Verlag, ISBN: 978-1-84628-975-0, pp. 421-430

The Service UI Project. Available at: http://www.artima.com/jini/serviceui/index.html. Accessed on: April 24, 2009

Waldo J. The End of Protocols, Available at: http://java.sun.com/developer/technicalArticles/jini/protocols.html. Accessed on: April 24 15, 2009.

Zhao, S.; and Sobolewski, M. (2001). Context Model Sharing in the FIPER Environment, Proc. of the 8th *Int. Conference on Concurrent Engineering: Research and Applications*, Anaheim, CA