SORCER: COMPUTING AND METACOMPUTING INTERGRID

Michael Soblewski
Texas Tech University, Lubbock, TX, U.S.A.

Keywords: Metacomputing, metaprogramming, grid computing, service-oriented architectures, service-oriented programming.

Abstract: This paper investigates grid computing from the point of view three basic computing platforms. Each platform considered consists of virtual compute resources, a programming environment allowing for the development of grid applications, and a grid operating system to execute user programs and to make solving complex user problems easier. Three platforms are discussed: compute grid, metacompute grid and intergrid. Service protocol-oriented architectures are contrasted with service object-oriented architectures, then the current SORCER metacompute grid based on a service object-oriented architecture and a new metacomputing paradigm is described and analyzed. Finally, we explain how SORCER, with its core services and federated file system, can also be used as a traditional compute grid and an intergrid—a hybrid of compute and metacompute grids.

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 with a varying focus on architectures, resource management, access, virtualization, provisioning, and sharing between heterogeneous compute domains. Thus, diverse compute resources across different administrative domains form a *grid* for the shared and coordinated use of resources in dynamic, distributed, and virtual computing organizations (Foster, 2002). Therefore, the grid requires a *platform* that describes some sort of framework to allow software to run utilizing virtual organizations. These organizations are dynamic subsets of departmental grids, enterprise grids, and global grids, which allow programs to use shared resources—collaborative federations.

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, 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 programming language is an abstraction of the 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. For example, a procedural language provides an abstraction of an underlying machine language. An executable file represents a computing component whose content is meant to be 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., parallelizing it 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", 2008) or the Condor-G high-throughput scheduler (Thain, 2003). Both rely on Globus/GRAM (Grid Resource Allocation and Management) protocol (Foster, 2002). In this type of grid, called a *compute grid*, executable files are moved around the grid to form virtual federations of

Soblewski M. (2008).
SORCER: COMPUTING AND METACOMPUTING INTERGRID.
In *Proceedings of the Tenth International Conference on Enterprise Information Systems - ISAS*, pages 74-85
DOI: 10.5220/0001690700740085

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.

A grid programming language is the abstraction of hierarchically organized networked processors running a grid computing program—*metaprogram*—that makes decisions about component programs such as when and how to run them. 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 are structurally completely different entities. Most grid programs are still written using compiled languages such as FORTRAN, C, C++, Java, and scripting languages such as Perl and Python the way it usually works on a single host. The current trend is to have these programs and scripts define grid computational modules. 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 resources in the grid according to a control strategy using a kind of *grid 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 on the part of the user, delays in 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 grid.

Instead of moving executable files around the 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 can distinguish three types of grids depending on the nature of computational components: *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 files with the help of a grid resource broker. However, an 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 (intergrid applications) as depicted in Figure 1, where platform layers P1, P2, and P3 correspond to resources, resource management, and programming environment correspondingly.

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) ("FIPER", 2008; Röhl, 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.

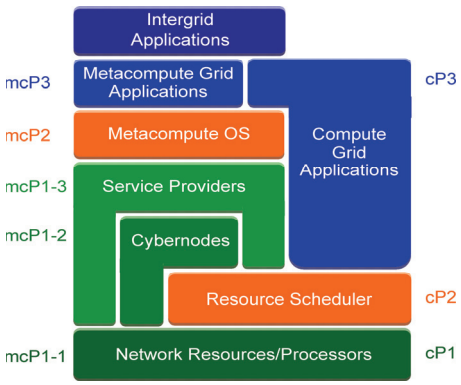

Figure 1: Three types of grids: compute grid, metacompute grid, and intergrid. A cybernode provides a lightweight dynamic virtual processor, turning heterogeneous compute resources into homogeneous services available to the metacomputing OS ("Project Rio", n.d).

The Service-ORiented Computing EnviRonment (SORCER) (Sobolewski, 2007; Sobolewski, 2008) builds on the top of FIPER to introduce a metacomputing operating system with all basic services necessary, including a federated file system, 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. The EO programming makes a positive difference in service-oriented programming primarily through a new metaprogramming abstraction as experienced in many grid-computing projects including systems deployed at GE Global Research Center, GE Aviation, Air Force Research Lab, and SORCER Lab (Burton, 2002; Kolonay, 2002; Sampath, 2002; Kao, 2003; Lapinski, 2003; Khurana, 2005; Sobolewski, 2006; Berger, 2007; Turner, 2007; Goel, 2005; Goel, 2007; Kolonay, 2007; "SORCER Research", 2008).

The paper is organized as follows. Section 2 provides a brief description of two service-oriented architectures used in grid computing with a related discussion of distribution transparency; Section 3 describes the SORCER metacomputing philosophy and mcGrid; Section 4 describes the SORCER cGrid, Section 5 the metacomputing file system, and Section 6 the SORCER iGrid; Section 7 provides concluding remarks.

2 SOA = SPOA+SOOA

Various definitions of a Service-Oriented Architecture (SOA) leave a lot of room for interpretation. Nowadays SOA becomes the leading architectural approach to most grid developments. 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. 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 add published services. The requestor looks up a service by sending queries to registries and making selections from the available services. Queries generally contain search criteria related to the service name/type and quality of service. Registries facilitate searching by storing the service representation and making it available to requestors. Requestors and providers can use discovery and join protocols to locate registries and then publish or acquire services on the network.

We can distinguish the *service object-oriented architecture* (SOOA), where providers, requestors, and proxies are network objects, from the *service protocol oriented architecture* (SPOA), where a communication protocol is fixed and known beforehand by the both provider and requestor. Using SPOA, a requestor can use this fixed protocol and a service description obtained from a service registry to create a proxy for binding to the service provider and 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.

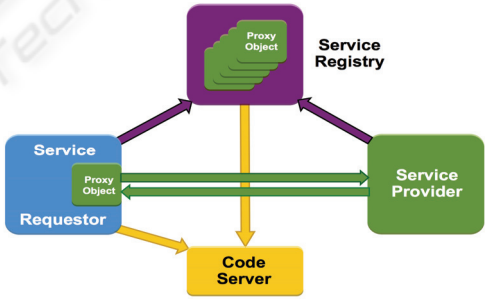

Figure 2: Service object-oriented architecture.

In SOOA (see Figure 2), 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, the service provider publishes the proxy as the active surrogate object with a codebase annotation, e.g., URLs to the code defining proxy behavior in Jini ERI ("Package net.jini.jeri", n.d.). In SPOA, by contrast, a passive service description is registered (e.g., an XML document in WSDL for Web/OGSA services (McGovern, 2003; Sotomayor, 2005) or an interface description in IDL for CORBA (Ruth, 1999)); the requestor then has to generate the proxy

(a stub forwarding calls to a provider) based on the 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 binding operation is not needed in SOOA since the requestor holds the active surrogate object obtained from the registry.

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, the way an 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, how it is implemented, or where the provider is located—three neutralities of SOOA. So-called smart proxies (e.g., provided by Jini ERI) 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(s). Communication protocols may also vary, and a single smart proxy can also talk over multiple protocols including application specific protocols.

SPOA and SOOA differ in their method of discovering the service registry. SORCER uses dynamic discovery protocols to locate available registries (lookup services) as defined in the Jini architecture ("Jini Architecture", 2001). Neither the requestor who is looking up a proxy by its interfaces nor the provider registering a proxy needs to know specific registry locations. In SPOA, however, the requestor and provider usually do need to know the explicit location of the service registry—e.g., a URL for RMI registry (Pitt, 2001), a URL for UDDI registry (McGovern, 2003), an IP address and port of a COS Name Server (Ruh, 1999)—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; in SOOA, lookup services are mandatory due to the dynamic nature of objects identified by service types. Interactions in SPOA are more like static client-server connections (e.g., HTTP, SOAP, IIOP) in many cases with no need to use service registries at all.

Crucial to the success of SOOA is interface standardization. Services are identified by interfaces (service types, e.g., Java interfaces) and additional provider's specific properties if needed; the exact identity of the service provider is not crucial to the architecture. As long as services adhere to a given set of rules (common interfaces), they can collaborate to execute published operations, provided the requestor is authorized to do so.

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 a service registry using object types and communicating through message passing. A collection of these object sending and receiving messages—the only way these objects communicate with one another—looks very much like a service object-oriented distributed system.

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, with notable exception of Jini (Edwards, 2000) and SORCER, 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 *dynamic 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 architecture emphasizes.

Object-oriented programming can be seen as an attempt to abstract both *data* and related *operations* in an entity called *object*. Thus, object-oriented program may 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. The exertion can be considered as the *specification of collaboration* that encapsulates *data*, related *operations,* and *control strategy*. The operations specify implicitly the required service providers on the network. The activated exertion creates at runtime a federation of providers to execute a service collaboration according to the exertion's control strategy. Thus, the exertion is the *metaprogram* and its *metashell* 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 Figure 3)—the exemplification of SOOA with autonomic management of system and domain-specific service providers to run EO programs.

SORCER defines the object-oriented distribution for EO programming (Sobolewski, 2008). 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 providers and the exertion shell, federated file system, and the system brokers responsible for coordination of executing federations. That 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.

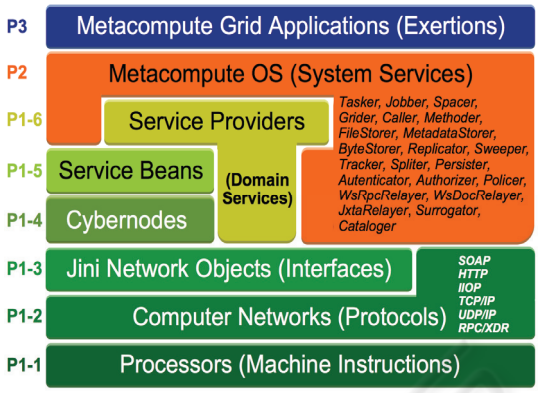

Figure 3: SORCER layered platform, where P1 resources, P2 resource management, P3 programming environment.

3 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.

As described in Section 2, 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 execute 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 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 *object-oriented* program—*specification* of service collaboration. This changes the programming paradigm completely. In the former case the object space is hosted by a single computer, but in the latter case the parent and its component exertions along with related 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 metacomputer*. The *metainstruction set* of the metacomputer 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 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, 2001; Sobolewski, 2008) 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. An EO program allows for a dynamic federation to transparently coordinate the execution of all component exertions within the grid. 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 allows us to submit an exertion onto the network and to perform executions of exertion's signatures on various service providers indirectly, but where does the service-to-service communication come into play? How do these services communicate with one another if they are all different? Top-level communication between services, or the sending of service requests, 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 creation new collaborative applications during the exertion runtime.

Despite the fact that every `Servicer` can accept any exertion, `Servicer`s have well defined roles in the S2S platform (see Figure 3):

a) `Tasker`s – process service tasks
b) `Jobber`s – process service jobs
c) `Spacer`s – process tasks and jobs via exertion space for space-based computing (Freeman, 1999)
d) `Contexter`s – provide service contexts for `APPEND Signature`s
e) `FileStorer`s – provide access to federated file system providers (Sobolewski, 2003, Berger, 2005, Berger 2007)
f) `Catalogers` – `Servicer` registries
g) `Persister`s – persist service contexts, tasks, and jobs to be reused for interactive EO programming
h) `Relayer`s – gateway providers; transform exertions to native representation, for example integration with Web services and JXTA ("JXTA", n.d.)
i) `Autenticator`s, `Authorizer`s, `Policer`s, `KeyStorer`s – provide support for service security
j) `Auditor`s, `Reporter`s, `Logger`s – support for accountability, reporting, and logging
k) `Grider`s, `Caller`s, `Methoder`s – support compute grid (see Section 4)
l) 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.

An exertion can be created interactively (Sobolewski, 2006) or programmatically (using SORCER APIs), and its execution can be monitored and debugged (Soorianarayanan, 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. Jobs are coordinated by *rendezvous peers*: a *Jobber or Spacer*, two of the SORCER platform core 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 the 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` method. The SORCER API defines the following three related operations:

1. `Exertion.exert(Transaction):Exertion` – join the federation; the activated exertion binds to an 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.

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 (Oram, 2001) via the `Servicer` interface, extensive modularization of `Exertion`s and `Exerter`s, 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`, or `ServiceSpacer`. The `ServiceTasker` peer calls by reflection the application method specified in the `PROCESS` signature of the task exertion. All application domain methods of any interface have the same signature: a single `Context` type parameter and a `Context` type return vale. Thus an application interface looks like an RMI (Pitt, 2001) interface with the above simplification on the common signature for all interface methods.

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. Alternatively, when signature's access type is `PULL`, a `ServiceAccessor` can use a `Spacer` provider and simply drops the exertion into the shared exertion space to be pulled by a matching provider. The execution order of signatures is defined by signature priorities, if the exertion's flow type is `SEQUENTIAL`, otherwise they are dispatch in parallel. In Figure 4 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.

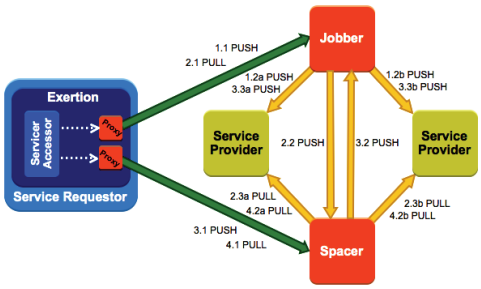

Figure 4: Push vs. pull exertion processing.

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 Figure 2.
The exertion's `ServicerAccessor` delivers the right service proxy dynamically, either for a `Jobber` or `Spacer`. If the access type of the parent exertion is `PUSH`, then all the component exertions are directly passed 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 sevicer gets an exertion from the space before other overloaded or slower servicers can do so. When an exertion consists of component jobs with different access and flow types, then we have a *hybrid* case when the collaboration potentially executes concurrently with multiple *pull* and *push* subcollaborations at the same time.

4 COMPUTE GRID

Also, 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` in the role of grid scheduler. `Caller` and `Methoder` services are used for task execution received from `Jobbers` or picked 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 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, 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 system call for 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) and easy-to-use zero-install service UI attached to a Grider and the federated file system.

5 FEDERATED FILE SYSTEM

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

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 will be overloaded. In a 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 metacomputer 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. 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 point failures. SILENUS services can be broadly categorized into gateway components, data services, and management services.

The SILENUS façade service provides a gateway service to the SILENUS grid 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 like metadata and byte stores.

Core SILENUS services have been successfully deployed as SORCER services along with WebDAV and NFS adapters. The SILENUS file system scales 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.

SILENUS supports storing very large files (Turner, 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 enough, 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 to the original form.

6 SORCER IGRID

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. Relayers exhibit native and mcGrid behavior. Some native cGrid providers play SORCER role (SORCER wrappers), thus are available in the iGrid along with mcGrid providers. Also, native cGrid providers via corresponding relayers can access iGrid services (bottom-up in Figure 5).

The iGrid-integrating model is depicted in Fig 5, 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 protocol, location, and implementation neutrality along with the flexibility of EO programming for iGrid computing.

7 CONCLUSIONS

An object-oriented grid is not just a collection of distributed objects; it is the network of unreliable objects that come and go. From an object-oriented point of view, the network of objects is the problem domain of object-oriented distributed programming that requires relevant abstractions in the solution space—a metacompute OS. The SORCER architecture shares the features of grid systems, P2P systems, and provides a platform for procedural programming and service-oriented meta-programming as well. EO programming introduces new network abstractions with federated method invocation in FSOOA. It allows for creation adaptive collaborative applications at the exertion runtime.

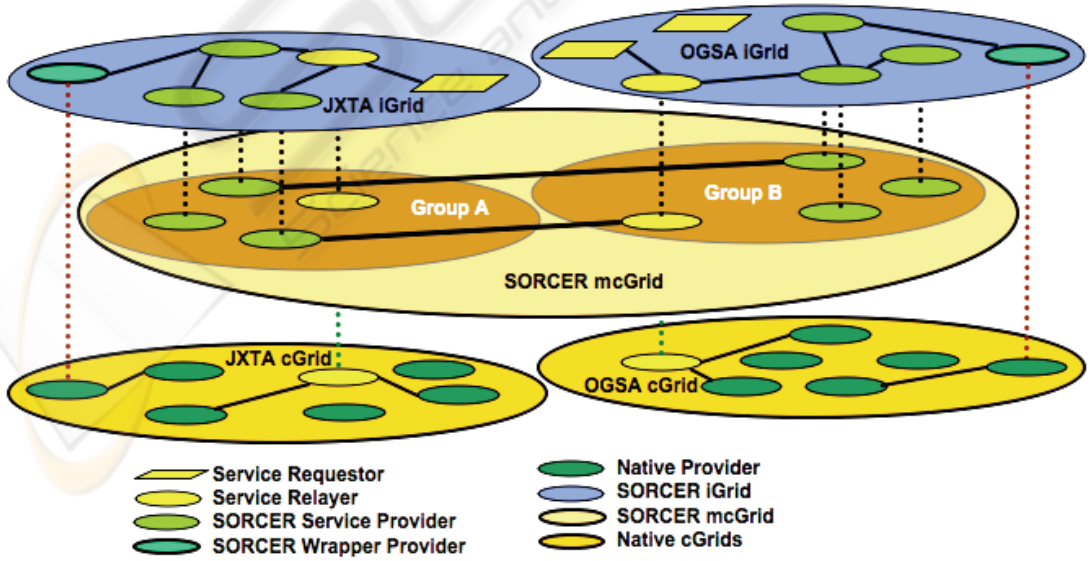

Figure 5: Integrating and wrapping cGrids with SORCER mcGrids. Two requestors, one in JXTA iGrid, one in OGSA iGrid submits exertion to a corresponding relayer. Two federations are formed that include providers from all the two horizontal layers below the iGrid layer (as indicated by continues and dashed links).

Executing a top-level exertion means federating currently available providers in the network that collaboratively process service contexts of all nested exertions. Services are invoked by passing exertions on to one another indirectly via object proxies that act as access proxies allowing for service providers to enforce security policies on access to services. When permission is granted, then the operation defined by a signature is invoked by reflection. SORCER allows for P2P computing via the common `Servicer` interface, extensive modularization of `Exertion`s and `Exerter`s, and extensibility from the Triple Command design pattern. The SORCER federated file system is modularized into a collection of distributed providers with multiple remote façades. Façades supply uniform access points via their smart proxies available dynamically to file requestors. A façade's smart proxy encapsulates inner proxies to federating file system providers that are accessed directly (P2P) by file requestors.

The SORCER iGrid has been successfully tested in multiple concurrent engineering and large-scale distributed applications (Burton, 2002; Kolonay, 2002; Sampath, 2002; Kao, 2003; Lapinski, 2003; Khurana, 2005; Sobolewski, 2006; Berger, 2007; Turner, 2007; Goel, 2005; Goel, 2007; Kolonay, 2007; "SORCER Research", 2008). Due to the large-scale complexity of the evolving iGrid environment, it is still a work in progress and continues to be refined and extended by the SORCER Research Group at Texas Tech University ("SORCER Lab", n.d.) in collaboration with Air Force Research Lab, WPAFB. The SORCER approach is consistent with the object-oriented distributed granularity of many service provider and exertion types, and provider configuration-based dependency injection.

REFERENCES

Altschul, S.F., Gish, W., Miller, W., Myers, E. W. & Lipman, D. J. (1990). Basic Local Alignment Search Tool. *J. Mol. Biol.* 215:403-410.

Berger, M., Sobolewski, M. (2005). SILENUS—A Federated Service-oriented Approach to Distributed File Systems, *Next Generation Concurrent Engineering*. ISPE/Omnipress, ISBN 0-9768246-0-4, pp. 89-96.

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

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, ISBN: 0-13-089408.

Fallacies of Distributed Computing. Retrieved March 5, 2008, from:http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing.

FIPER: Federated Intelligent Product EnviRonmet. Retrieved March 5, 2008, from: http://sorcer.cs.ttu.edu/fiper/fiper.html.

Foster, I., Kesselman, C., Nick, J., S. Tuecke, S. (2002). The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration., Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002. Retrieved March 5, 2008, from: http://www.globus.org/alliance/publications/papers/ogsa.pdf.

Freeman, E., Hupfer, S., & Arnold, K. (1999). *JavaSpaces™ Principles, Patterns, and Practice*. Addison-Wesley, ISBN: 0-201-30955-6.

Goel, S, Talya, S., and Sobolewski, M. (2005). Preliminary Design Using Distributed Service-based Computing, *Next Generation Concurrent Engineering*. ISPE/Omnipress, ISBN 0-9768246-0-4, pp. 113-120.

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

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

Inca X™ Service Browser for Jini Technology. Retrieved March 5, 2008, from: http://www.incax.com/index.htm?http://www.incax.com/service-browser.htm.

JXTA (n.d). Retrieved March 5, 2008, from: https://jxta.dev.java.net/

Jini architecture specification, Version 1.2., 2001. Retrieved March 5, 2008, from: http://www.sun.com/software/jini/specs/jini1.2html/jini-title.html.

Jini.org. Retrieved March 5, 2008, from: http://www.jini.org/.

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.

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., Franklin 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.

McGovern, J., Tyagi, S., Stevens, M. E., Mathew, S. (2003). *Java Web Services Architecture*, Morgan Kaufmann.

Nimrod: Tools for Distributed Parametric Modelling. Retrieved March5,2008,from:http://www.csse.monash.edu.au/~davida/nimrod/nimrodg.htm.

Oram, A. (Editor) (2001). *Peer-to-Peer: Harnessing the Benefits of Disruptive Technology*. O'Reilly.

Package net.jini.jeri. Retrieved March 5, 2008, from: http://java.sun.com/products/jini/2.1/doc/api/net/jini/jeri/package-summary.html

Pitt, E. (2001) *java™.rmi: The Remote Method Invocation Guide*. Addison-Wesley

Project Rio. A Dynamic Service Architecture for Distributed Applications. Retrieved March 5, 2008, from: https://rio.dev.java.net/.

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

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

Sobolewski, M. (2002b). FIPER: The Federated S2S Environment. *JavaOne, Sun's 2002 Worldwide Java Developer Conference*. Retrieved March 5, 2008, from: http://sorcer.cs.ttu.edu/publications/papers/2420.pdf.

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, Scottsdale, AZ. ACTA Press.

Sobolewski, M., Kolonay R. (2006) Federated Grid Computing with Interactive Service-oriented Programming, *International Journal of Concurrent Engineering: Research & Applications*. Vol. 14: pp. 55-66.

Sobolewski, M. (2007). Federated Method Invocation with Exertions, *Proceedings of the International Multiconference on Computer Science and Information Technology*, Springer Verlag, ISSN 1896-7094, Vol. 2: pp. 765 – 778. Retrieved March 5, 2008, from: http://www.proceedings2007.imcsit.org/pliks/96.pdf.

Sobolewski, M. (2008). Service-oriented Programming, SORCER Technical Report SL-TR-13. Retrieved March 5, 2008,from:http://sorcer.cs.ttu.edu/publications/papers/2008/SL-TR-13.pdf.

Soorianarayanan, S., Sobolewski, M. (2004). Monitoring Federated Services in CE, *Concurrent Engineering: The Worldwide Engineering Grid*, Tsinghua Press and Springer Verlag, ISBN 7-302-08802-0, pp. 89-95.

SORCER Lab. RetrievedMarch5, 2008, from: http://sorcer.cs.ttu.edu/.

SORCER Research Topics. Retrieved March 5, 2008, from: http://sorcer.cs.ttu.edu/theses/.

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 Fran Berman, Anthony J.G. Hey, and Geoffrey Fox, editors, *Grid Computing: Making The Global Infrastructure a Reality*. 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. (Eds.). Springer Verlag, ISBN: 978-1-84628-975-0, pp. 421-430.

The Service UI Project. Retrieved March 5, 2008, from: http://www.artima.com/jini/serviceui/index.html.

Waldo, J. (n.d.). The End of Protocols. Retrieved March 5, 2008,from: http://java.sun.com/developer/technicalArticles/jini/protocols.html.

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.