SORCER: Computing and Metacomputing Intergrid

Michael Sobolewski

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

Abstract— **This paper investigates Grid computing from the point of view three basic computing platforms. The platform 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 SORCER metacompute Grid based on the service object-oriented paradigm is presented. Finally, we explain how SORCER, with its core services and federated file system, can be used as a traditional compute Grid and an Intergrid—a hybrid of compute and metacompute Grids.**

I. Introduction

The term "Grid computing" originated in the early 1990s as a metaphor for making computer power as easy to access 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 federations [9]. Therefore, the Grid requires a *platform* that describes some sort of framework to allow software to run utilizing virtual federations. These federations are dynamic subsets of departmental Grids, enterprise Grids, and global Grids, which allow programs to run. 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 considered below.

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 platform. A request can be submitted to a *Grid resource broker* to execute a program 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 [23] Grid resource broker scheduler or the Condor-G [5], [39] high-throughput scheduler. Both rely on Globus/GRAM [9] (Grid Resource Allocation and Management) protocol. 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.

A Grid programming language is the abstraction of hierarchically organized networked processors running a Grid computing program—*metaprogram*—that makes decisions about programs such as when and how to run them. Nowadays the same computing module abstraction is applied to a single computer module as it usually is applied to a Grid computing module, even though they are structurally completely different entities. Most Grid modules are still written as monolithic programs using compiled languages such as FORTRAN, C, C++, Java, and scripting languages such as Perl and Python. 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 some *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

This is a DRAFT document and continues to be revised. The latest version can be found at http://sorcer.cs.ttu.edu/publications/papers/sorcer-intergrid.pdf. Please send comments and remarks to sobol@cs.tt.edu.

terms of metainstructions to the *Grid platform* (*operating system*) that manages a dynamic federation of service providers and related resources and enables the metaprogram to interact with the providers according to the metaprogram control strategy.

Thus, 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 cGrid is a virtual federation of processors (roughly CPUs) that execute submitted executable files 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 iGrid to execute both programs and metaprograms as depicted in Fig. 1, where platform layers P1, P2, and P3 are resources, resource management, and programming environment correspondingly.

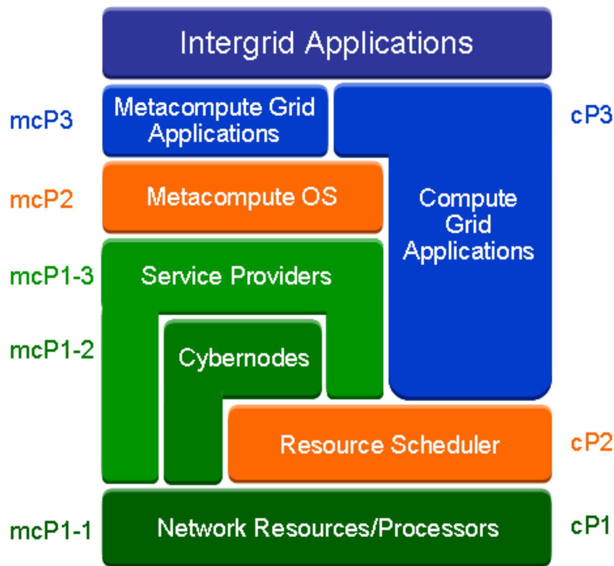

Fig. 1 Three types of Grids: compute grid, metacompute grid, and Intergrid. A cybernode provides a lightweight dynamic virtual processor, turning heterogeneous compute resources into homogeneous services available to the metacomputing OS [22].

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) [8], [26], [29], [30]. 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) [36], [32], [35], [34], [1] builds on top of FIPER to introduce a metacomputing operating system with all basic services necessary, including a federated file system, to support service-oriented programming. It provides an integrated solution for complex metacomputing systems.

The paper is organized as follows. Section II provides a brief description of a service-oriented architecture used in Grid computing with a related discussion of distribution transparency; Section III describes the SORCER metacomputing philosophy and mcGrid; Section IV describes SORCER cGrid, Section V the metacomputing file system, and Section VI SORCER iGrid; Section VII provides concluding remarks.

II. 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 architectures* (SOOA), where providers, requestors, and proxies are network objects, from *service protocol oriented architectures* (SPOA), where a communication protocol is fixed and known beforehand by the 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.

In SOOA (see Fig. 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

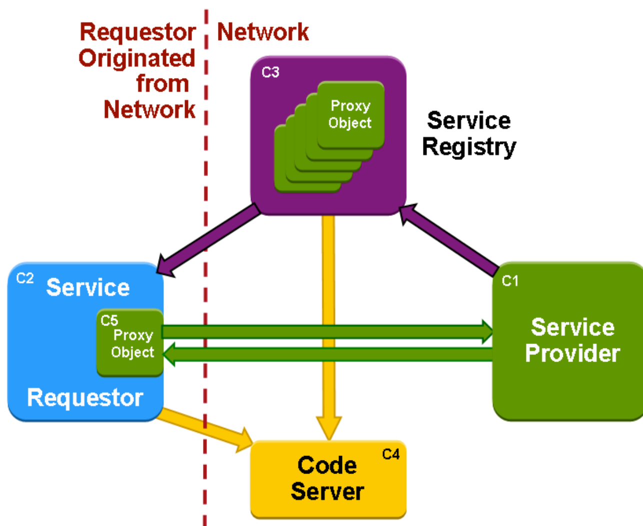

Fig. 2 Service object-oriented architecture

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 (RMI and Jini ERI [6]). In SPOA, by contrast, a passive service description is registered (e.g., an XML document in WSDL for Web/OGSA services [22], [38], 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 [27]). 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 [42]. 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 or how it is implemented. 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. 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 [15]. 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., 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 (WSDL descriptions are shared via files outside of the system); in SOOA, lookup services are mandatory due to the dynamic nature of objects identified by service types. Interactions in SPOA are more like 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); 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's 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 some 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 (interfaces) 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 of network computing[7]? 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 [15], 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 distributed system and cannot be represented completely within a single entity.

The network is dynamic, can't 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 transparency distribution—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's a metasystemic issue in object-oriented

distributed programming. In that context Web/OGSA service define distributed objects, but do not have anything common with object-oriented distributed systems.

Exertion-based programming [32], [30] was introduced to handle the metasystemic distribution in SORCER. It uses indirect remote method invocation with no service provider explicitly specified in a network request called an exertion. Specialized infrastructure objects support exertion-oriented programming. That infrastructure defines SORCER's distributed object modularity, extensibility, and reuse of service-oriented components consistent with the relevant metacomputing granularity and dependency injection—key features of object-oriented distributed programming that are usually missing in SPOA programming environments.

III. SORCER Metacomputing Grid

SORCER is a federated service-to-service (S2S) metacomputing environment that treats service providers as network objects with well-defined semantics of a federated service object-oriented architecture (FSOOA). It is based on Jini semantics of services [15] in the network and the Jini programming model [6] with explicit leases, distributed events, transactions, and discovery/join protocols. While Jini [16], [17] focuses on service management in a networked environment, SORCER is focused on exertion-oriented programming and the execution environment for exertions.

As described in Section II, SOOA consists of three major types of network objects: providers, requestors, and registries. The provider is responsible for deploying the service on the network, publishing its service to one or more registries, and allowing requestors to access its service. 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

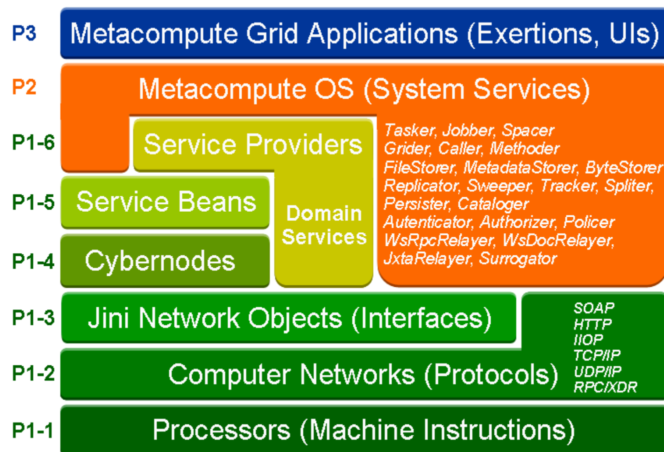

Fig. 3 SORCER layered platform, where P1 resources, P2 resource management, P3 programming environment

protocols to obtain service proxies on the network. SORCER uses Jini discovery/join protocols to implement its FSOOA.

In SOOA, a service provider is an object that accepts remote messages from service requestors to execute an item of work. These messages are called *service exertions*. 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, a kind of network procedure executed by a large-scale federation. The executing exertion is a service-oriented program that is dynamically bound to all needed and currently available service providers on the network. This collection of providers identified in runtime is called an *exertion federation*. This federation is also called an *exertion space*. While this sounds similar to the object-oriented paradigm, it really isn't. In the object-oriented paradigm, the object space is a program itself; here the exertion space is the *execution environment* for the exertion that is a service-oriented distributed program. 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 service providers are hosted by the network of computers.

The overlay network of service providers is called the *service provider grid* and an exertion federation is called a *virtual metacomputer*. The *metainstruction set* of the metacomputer consists of all operations offered by all service providers in the grid. Thus, a service-oriented program is composed of metainstructions with its own service-oriented control strategy and service context [43] representing the metaprogram parameters. 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 attributes. Three types of signatures are distinguished: PROCESS, PREPROCESS, and POSTPROCESS. 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 describes the data that tasks and jobs work on. Exertion-oriented programs (metaprograms) allow for a dynamic federation to transparently coordinate their execution within the grid. Please note that these metacomputing concepts are defined differently in classical grid computing where a job is just an executing process for a submitted executable code with no federation being formed for the executable.

In a federated service environment, the system is not made up of just a single service, but the cooperation of many services. A service exertion may consist of hierarchically nested exertions that require different service types. A service can be broken down into small component services instead of being one monolithic all-in-one service. These smaller component services—treated as virtual metacomputer

instructions—can then be distributed among different hosts to allow for reusability, scalability, reliability, and load balancing.

Each SORCER provider (peer) implementing the common Servicer interface, offers services to other peers [32] on the object-oriented overlay network. These services are exposed indirectly by methods in well-known public remote interfaces and considered as elementary (tasks) or compound (jobs) statements of the FSOOA. Requestors do not need to know the exact location of a provider beforehand; they can find it dynamically by discovering service registries (lookup services) and then looking up a needed service implementing required service types.

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

a) Taskers – process service tasks
b) Jobbers – process service jobs
c) Contexters – provide service contexts for APPEND Signatures
d) FileStorers – provide access to federated file system providers [34], [1], [3], [40]
e) Catalogers – Servicer registries
f) Persisters – persist service contexts, tasks, and jobs to be reused for interactive exertion-based programming
g) Spacers – manage exertion spaces shared across Servicers for space-based computing [10]
h) Relayers – gateway providers; transform exertions to native representation, for example integration with Web services and JXTA
i) Autenticators, Authorizers, Policers, KeyStorers – provide support for service-oriented security
j) Auditors, Reporters, Loggers – support for accountability, reporting and logging
k) Griders, Callers, Methoders – support conventional grid computing (in cGrids)
l) Generic ServiceTasker and ServiceJobber 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 [33] or programmatically (using SORCER APIs), and its execution can be monitored and debugged [35], [21] in the overlay service network via service user interfaces (Service UI [41]) attached to providers and installed on the fly by service browsers [14]. 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* called Taskers, execute service tasks. Jobs are coordinated by *rendezvous peers*: a *Jobber or Spacer*, two of the SORCER core services (see Fig. 3 for details), of the SORCER platform. However, a job can be sent to any service provider (peer). A peer that is not a Jobber type is responsible for forwarding the job to an available job peer in the SORCER grid and returning results to the requestor. Thus implicitly, any peer can handle any job or task. Once the job execution is complete, the federation dissolves and the providers disperse to seek other exertions to join.

An Exertion is invoked by calling on its exert method. The SORCER API defines the following three related operations:

1. Exertion.exert(Transaction):Exertion - join the federation
2. Servicer.service(Exertion, Transaction):Exertion – request a service in the federation initiated by the receiver
3. Exerter.exert(Exertion, Transaction):Exertion – execute the component exertion by the target provider in the federation

This Triple Command pattern [32], [13] 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 [24] via the Servicer interface, extensive modularization of Exertions and Exerters, and extensibility from the triple design pattern so requestors can submit any service-oriented programs (exertions) they want with or without transactional semantics. The triple Command Pattern is used as follows:

1. An exertion can be invoked by calling Exertion.exert(Transaction). The Exertion.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 (which can also be a smart proxy) the Servicer.service(Exertion, Transaction) 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(Exertion, Transaction).
4. Exerter.exert method calls exert on either of ServiceTasker or ServiceJobber (depending on the type of the exertion: either Task or Job) that by reflection calls the method specified in the PROCES signature (interface and selector) of the exertion. All application domain methods of any application interface (custom Tasker interfaces) have the same signature: a single Context type parameter and a Context type return vale. Thus a custom interface looks like an RMI interface with the above simplification on the common signature for all interface methods.

The fundamentals of exertion-oriented programming and SORCER federated method invocation are described in [32].

In Fig. 4 four use cases are presented to illustrate push vs. pull exertion-oriented computing. We assume that an exertion is a job with two component exertions executed in parallel (a and b). The Job exertion can be submitted directly to either Jobber (use cases: 1. access is PUSH, and 2. access is DROP) or Spacer (use cases: 3. access is PUSH, and 4. access is

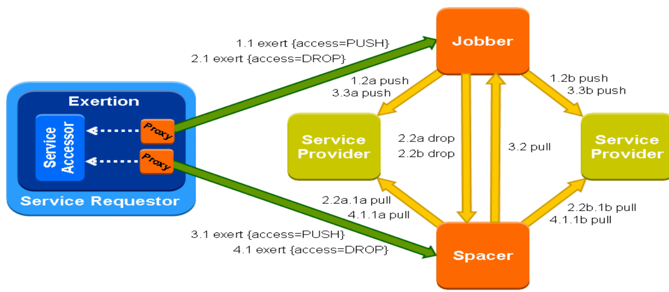

Fig. 4 Push vs. Pull exertion computing

DROP) depending on the exertion's interface defined in its PROCES signature. In cases 1 and 2 the signature is Jobber and in cases 3 and 4 the signature is Spacer. The exertion's ServicerAccessor delivers the right service proxy, either for a Jobber or Spacer. Depending on the access type of the parent exertion, all the component exertions are pushed to relevant providers according to their signatures (case 1 and 3), or dropped into the exertion space by the Jobber (case 2) and Spacer (case 4). In the cases 2 and 4, the component exertions are pulled from the exertion space by providers matching their signatures as soon as they are available to do any processing (case 2 and 4). Thus Spacers provide efficient load balancing for processing the exertion space.

IV. SORCER Computing Grid

Also, to use legacy applications, SORCER supports a traditional approach to grid computing similar to those found in Condor [5] and Globus [38]. Here, instead of exertions being executed by services providing business logic for requested exertions, the business logic comes from the service requestor's executable programs that seek compute resources on the network.

The cGrid-based services in the SORCER environment include Grider collaborating Jobber for compute grid job submission, and *Caller* and Methoder services for task execution [32]. Callers execute conventional programs via a system call as described in the Caller's service context of the submitted task. Methoders download required Java code (task method) from requestors to process any submitted context accordingly with the downloaded code. In either case, the business logic comes from requestors; it is conventional executable code invoked by Callers with the standard Caller's service context or mobile Java code executed by Methoders with any service context provided by the requestor.

The SORCER cGrid with Methoders was used to deploy an algorithm called Basic Local Alignment Search Tool (BLAST) [1] to compare newly discovered, unknown DNA and protein sequences against a large database with more than 3 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's Livestock Issues 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 [19], 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 17 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 UIs attached to a Grider and the SORCER federated file system.

V. SORCER Federated File System

The SILENUS federated file system [1], [3] was designed and developed to provide data access for metaprograms. It complements the file store developed for FIPER [34] 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 grid requestors try to access large files at the same time, the server will be overloaded. In a P2P architecture, every host 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 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 facade 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 facade 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 need to coordinate communication between these two services. The facade service itself is split into a provider component, called the coordinator, and a smart proxy component that contains

needed inner proxies provided dynamically by the coordinator. These inner proxies facilitate 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 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 very well by utilizing disconnected operation and data synchronization mechanisms. It provides a number of user agents including a zero-install file browser (service UI) attached to the SILENUS Facade. This file browser with file upload and download functions is combined with an HTML editor and multiple viewers for documents in HTML, RTF, and PDF formats. Also a simpler version of SILENUS file browser is available for smart MIDP phones.

SILENUS supports storing very large files [40] 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.

VI. 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, 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 own relayers can access iGrid services (bottom-up).

The iGrid-integrating framework 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 SORCER mcGrid's SOOA with its inherent protocol, location, and provider implementation neutrality, along with architectural qualities—flexibility, scalability, and adaptability for Intergrid computing.

VII. Conclusions

A Grid is not just a collection of distributed objects; it's the network of objects. 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. The SORCER architecture shares the features of grid systems, P2P systems and provides a platform for procedural programming and service-oriented metaprogramming. Exertion-based programming introduces new network abstractions with federated method invocation in SOOA. Service providers register proxies, including smart proxies, via dependency injection using twelve methods investigated in SORCER. Executing a top-level exertion means a dynamic federation of currently available providers in the network collaboratively process service contexts of all nested exertions. Services are invoked by passing exertions on to providers indirectly via object proxies that act as access proxies allowing for service providers to enforce a security policy on access to services. When permission is granted, then the operation defined by a signature is invoked by reflection. SORCER allows for the P2P computing via the common Service interface, extensive modularization of Exertions and Exerters, 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 providers accessed directly (P2P) by file requestors.

The SORCER iGrid has been successfully tested in multiple concurrent engineering, large-scale distributed applications [26], [28], [4], [11], [12], [18], [20]. 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 [36] in collaboration with Air Force Research Lab, WPAFB.

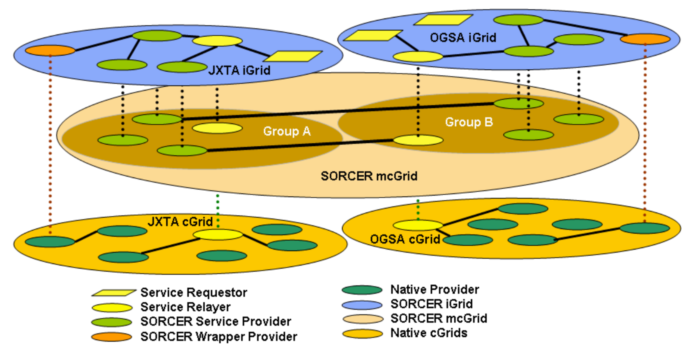

Fig. 5 Integrating and wrapping cGrids with SORCER mcGrids. Two requestors, one in JXTA iGrid, one in OGSA iGrid submits exertion to a corresponding relayer. Two federations are formed that include providers from all the two horizontal layers below the iGrid layer (as indicated by continues and dashed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

Acknowledgment

I would like to thank all my students in the SORCER Research Group [37] for their motivation, innovation, and excitement they generate when working on the iGrid development. Without their research efforts, it would not be possible to integrate so many different views of Grid

computing and to validate so many diverse and controversial opinions on distributed objects and iGrid computing.

REFERENCES

[1] Altschul, S.F., Gish, W., Miller, W., Myers, E.W. & Lipman, D.J., Basic Local Alignment Search Tool." J. Mol. Biol. 215:403-410, 1990.

[2] Berger, M., Sobolewski, M., SILENUS – A Federated Service-oriented Approach to Distributed File Systems, In Next Generation Concurrent Engineering [31]. pp. 89-96, 2005.

[3] Berger, M., Sobolewski, M., Lessons Learned from the SILENUS Federated File System, Proceeding of the 14th Conference on Concurrent Engineering, São José dos Campos, Brazil, Springer Verlag, 2007.

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

[5] Condor: High Throughput Computing, Available at: http://www.cs.wisc.edu/condor/condor_globus.html. Accessed on: March 15, 2007.

[6] Edwards W.K., Core Jini, 2nd ed., Prentice Hall, ISBN: 0-13-089408, 2000.

[7] Fallcies of Distributed Computing. Available at: http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing. Accessed on: March 15, 2007.

[8] FIPER: Federated Intelligent Product EnviRonmet. Available at: http://sorcer.cs.ttu.edu/fiper/fiper.html. Accessed on: March 15, 2007.

[9] Foster I., Kesselman C., Nick J., S. Tuecke S., 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. Available at: http://www.globus.org/alliance/publications/papers/ogsa.pdf. Accessed on: March 15, 2007.

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

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

[12] Goel, S, Talya S., and Sobolewski, M., Preliminary Design Using Distributed Service-based Computing, In Next Generation Concurrent Engineering [31], pp. 113-120, 2005.

[13] Grand M., Patterns in Java, Volume 1, Wiley, ISBN: 0-471-25841-5, 1999.

[14] Inca X™ Service Browser for Jini Technology. Available at: http://www.incax.com/index.htm?http://www.incax.com/service-browser.htm. Accessed on: March 15, 2007.

[15] Jini architecture specification, Version 1.2., 2001. Available at: http://www.sun.com/software/jini/specs/jini1.2html/jini-title.html. Accessed on: March 15, 2007.

[16] Jini, Wikipedia. Available at: http://en.wikipedia.org/wiki/Jini. Accessed on: March 15, 2007.

[17] Jini.org, Available at: http://www.jini.org/. Accessed on: March 15, 2007.

[18] Kao K.J., Seeley C.E., Yin Su, Kolonay R.M., Rus T., Paradis M.J., 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 USA, Sept. 2003.

[19] Khurana V., Berger M., Sobolewski M., A Federated Grid Environment with Replication Services. In Next Generation Concurrent Engineering [31], pp. 97-103, 2005.

[20] 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, 2002.

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

[22] McGovern J., Tyagi S., Stevens M.E., Mathew S., Java Web Services Architecture, Morgan Kaufmann, 2003.

[23] Nimrod: Tools for Distributed Parametric Modelling. Availabel at: http://www.csse.monash.edu.au/~davida/nimrod/nimrodg.htm. Accessed on: March 15, 2007.

[24] Oram Andy, Editor, Peer-to-Peer: Harnessing the Benefits of Disruptive Technology, O'Reilly (2001)

[25] Project Rio, A Dynamic Service Architecture for Distributed Applications. Available at: https://rio.dev.java.net/. Accessed on: March 15, 2007.

[26] Röhl, P.J., Kolonay, R.M., Irani, R.K., Sobolewski, M., Kao, K. A Federated Intelligent Product Environment, AIAA-2000-4902, 8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Long Beach, CA, September 6-8, 2000.

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

[28] Sampath R., Kolonay R.M, Kuhne C.M., "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, Sept. 2002

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

[30] Sobolewski M., FIPER: The Federated S2S Environment, JavaOne, Sun's 2002 Worldwide Java Developer Conference, 2002. Available at: http://sorcer.cs.ttu.edu/publications/papers/2420.pdf. Accessed on: March 15, 2007.

[31] Sobolewski M., Ghodous P. (Eds), Next Generation Concurrent Engineering. Proceeding of the 12th Conference on Concurrent Engineering: Research and Applications, ISPE/Omnipress (2005)

[32] Sobolewski M., Metacomputing with Federated Method Invocation, Technical Report SL-TR-11, March 2007. Available at: http://sorcer.cs.ttu.edu/publications/papers/FMI.pdf. Accessed on: April 5, 2007.

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

[34] 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, Nov 17-19, Scottsdale, AZ. ACTA Press, 2003.

[35] Soorianarayanan, S., Sobolewski, M., Monitoring Federated Services in CE, Concurrent Engineering: The Worldwide Engineering Grid, Tsinghua Press and Springer Verlag, pp. 89-95, 2004.

[36] SORCER Research Group. Available at: http://sorcer.cs.ttu.edu/. Accessed on: March 15, 2007.

[37] SORCER Research Topics. Available at: http://sorcer.cs.ttu.edu/theses/. Accessed on: March 15, 2007.

[38] Sotomayor B., Childers L., Globus® Toolkit 4: Programming Java Services, Morgan Kaufmann (2005)

[39] Thain D., Tannenbaum T., Livny M.. Condor and the Grid. In Berman F., Hey A.J.G., Fox G., editors, Grid Computing: Making The Global Infrastructure a Reality. John Wiley , 2003.

[40] Turner A., Sobolewski M., FICUS - A Federated Service-Oriented File Transfer Framework, Proceeding of the 14th Conference on Concurrent Engineering, São José dos Campos, Brazil, Springer Verlag (2007)

[41] The Service UI Project. Available at: http://www.artima.com/jini/serviceui/index.html. Accessed on: March 15, 2007.

[42] Waldo J., The End of Protocols, Available at: http://java.sun.com/developer/technicalArticles/jini/protocols.html. Accessed on: March 15, 2007.

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