Michael W. Sobolewski

profession: Professor
Director of SORCR Lab

office: CP 310

phone: (806) 742-1194

mobile: (806) 441-9999

fax: (806) 742-3519

e-mail: sobol@cs.ttu.edu

URL http://sobol.cs.ttu.edu/

s-mail: Computer Science
Texas Tech University
Box 43104
Boston and 8 th St.
Lubbock, TX 79409-3104

Search

Campus

RESEARCH

The computer is the network of service peers.

- SORCER - SOAs and Systems (theses)
- network protocols and security
- distributed and autonomic computing
- knowledge representation
- intelligent network agents
- mobile computing
- information/knowledge sharing
- service-oriented programming
- object oriented programming
- interactive graphics and UIs
- concurrent engineering

TEACHING

The class is the network of discovery peers.

- network security (F02)
- computer networks (S03, F04, F05)
- advanced network programming (S03)
- peer-to-peer computing (F03)
- computer science seminar (F03)
- mobile computing (S04)
- OO Programming in Java (SuII 03, F04, S06)
- communication networks (S05, F06)
- design patterns (F05)

Анонс лекций-презентаций Майкла Соболевского 22–24 мая 2007 г.

И. В. Семушин [1]

Ульяновский государственный университет, Ульяновск, ул. Л. Толстого, 42

1 SORCER: Вычислительный и мета-вычислительный Интергрид

Аннотация. Эта тема исследует сетевые вычисления с точки зрения трех основных вычислительных платформ. Любая такая платформа состоит из виртуальных вычислительных ресурсов, среды программирования, позволяющей разрабатывать сетевые приложения, и сетевой операционной системы для исполнения пользовательских программ и облегчения процесса решения сложных задач пользователя. Обсуждаются три платформы: вычислительная сеть, сеть для мета-вычислений и Интергрид (Интерсеть, организатор межсетевого взаимодействия). Сервис протокол-ориентированные архитектуры противопоставляются сервис объектно ориентированным архитектурам, затем представляется SORCER мета-вычислительная сеть, основанная на сервис объектно ориентированной парадигме. В заключение объясняется, каким образом SORCER с его корневыми сервисами и федеративной файловой системой может быть использован либо в качестве традиционной вычислительной сети, либо как Интерсеть — некоторый гибрид вычислительной и мета-вычислительной сети.

1. SORCER: Computing and Metacomputing Intergrid

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

Email address: innokentiy_v.sem@ulsu.ru (И. В. Семушин).
URL: http://staff.ulsu.ru/semoushin/ (И. В. Семушин).

[1] Визит д-ра Майкла Соболевского проводится в плане выполнения Меморандума о Сотрудничестве между Институтом компьютерных исследований Самарского государственного аэрокосмического университета (СГАУ) им. С. П. Королева, Самара, Россия, Факультетом математики и информационных технологий Ульяновского государственного университета (УлГУ), Ульяновск, Россия и Кафедрой вычислительной техники Техасского технического университета (ТТУ) и Лабораторией сервис-ориентированных вычислительных сред (SORCER), Лаббок, Техас, США.

Передано в НИЧ УлГУ *17 апреля 2007 г.*

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.

---

2 Мета-вычисления с вызовом федеративного метода (FMI)

Аннотация. Сервис провайдеры регистрируют посредников (proxies), в том числе, интеллектуальных посредников, путем внедрения признака подчиненности с использованием двенадцати методов, исследованных в лаборатории SORCER. Выполнение действий верхнего уровня означает динамическую федерацию доступных в текущий момент времени провайдеров в сервис контекстах всех вложенных друг в друга и совокупно протекающих процессов. Сервисы вызываются передачей команд о действиях провайдерам косвенным образом — через объектных посредников, которые являются посредниками доступа, позволяющими сервис провайдерам обеспечить соблюдение стратегии безопасности при предоставлении доступа к сервисам. Когда доступ разрешен, тогда операция, определенная некоторым признаком, вызывается посредством передачи ее точной копии. Вызов федеративного метода позволяет реализовать P2P (peer-to-peer) среду посредством интерфейса сервисов, расширенной модульной организации вызова действий (Exertions) и исполнителей действий (Exerters), а также расширяемости по проектному типу утроенной команды.

2. Metacomputing with Federated Method Invocation

Abstract. 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 are 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. FMI allows for the P2P environment via the Service interface, extensive modularization of Exertions and Exerters, and extensibility from the triple command design pattern.

---

3 Организация посредничества сервисов с помощью внедрения признака подчиненности

Аннотация. Улучшения в технике распределенных вычислений и технологии, которые делают это возможным, привели к значительному усовершенствованию средств промежуточного звена, т. е. средств, находящихся между аппаратным и программным обеспечением, — к улучшению их функциональности и качества, прежде всего, посредством сетевой организации и протоколов. Однако, стиль распределенного программирования остается таким же, как десять, двадцать, даже тридцать лет тому назад. Большинство программ все еще пишется строка за строкой программного кода на языке, подобно программам на Fortran, C, C++, или Java. Эти процедурного типа программы могут рассматриваться как общие ресурсы сети и использоваться сообща по всему миру работниками науки и образования. Однако, для этого нет отвечающих существу дела методологий программирования, которые позволили бы эффективно пользоваться этими процедурными ресурсами как неким потенциально огромным и доступным для всех хранилищем для мета-вычислений, исключая написание программного кода вручную — как раз то, что делалось десятилетия назад. Реализация этого потенциала мета-компьютинга требует значительных усовершенствований в технологии вычислений. Чтобы эффективно работать в больших, распределенных средах, группы параллельного инжиниринга нуждаются в некой сервис ориентированной методологии программирования. Нужны также: общий процесс проектирования, предметно-независимое представление проектов и общие критерии принятия (проектных) решений. Посредничество сервисов с помощью внедрения признака подчиненности может быть использовано для решения проблем, выдвигаемых парадигмой мета-вычислений для комплексной распределенной высокоточной оптимизации инженерных проектов.

3. Service Proxying with Dependency Injection

Abstract. Improvements in distributed computing, and the technologies that enable them, have led to significant improvements in middleware functionality and quality, mainly through networking and protocols. However, the distributed programming style is the same as ten, twenty, even thirty years ago. Most programs are still written line by line of code in languages like Fortran, C, C++, and Java. These procedural programs can be considered as common grid resources and shared by research and education communities worldwide. However, there are no relevant programming methodologies to utilize efficiently these procedural resources as a potentially vast and shared grid repository for metacomputing, except through the manual writing of code — just as it was done decades ago. Realization of the potential of metacomputing requires significant improvements in computing technology. To work effectively in large, distributed environments, concurrent engineering teams need a service-oriented programming methodology along with common design process, discipline-independent representations of designs, and general criteria for decision making. Proxying with dependency injection can be used to address several fundamental challenges posed by the emerging metacomputing paradigm for complex distributed high fidelity engineering design optimization.

4 Jini-платформа: Модель программирования, инфраструктура и Jini ERI

Аннотация. О JINI^{TM} ТЕХНОЛОГИИ. Технология Jini — это открытая программная архитектура, которая делает возможным связывание по сети для построения распределенных систем, в высокой степени приспособленных к изменениям. Эта технология может быть использована для создания технических систем, которые обладают способностью к масштабированию, развитию и гибкому изменению, что обычно и требуется в средах с динамическим временем выполнения. Технология Jini первоначально создавалась корпорацией Sun Microsystems и была передана в Jini CommunitySM в 1999 году. Она находится в свободном доступе и продвигается членами Сообщества Jini через открытый Jini Community Decision Process. Эта лекция может быть полезна в том смысле, что позволит понять, почему и как JINI^{TM} TECHNOLOGY используется в SORCER для управления динамическими федерациями сервисов.

4. Jini Platform: Programing Model, Infrastructure, Jini ERI

Abstract. ABOUT JINI^{TM} TECHNOLOGY. Jini technology is an open software architecture that enables Java dynamic networking for building distributed systems that are highly adaptive to change. It can be used to create technology systems that are scalable, evolvable, and flexible, as typically required in dynamic runtime environments. Jini technology was originally created by Sun Microsystems, and was contributed by Sun to the Jini Community-SM in 1999. It is freely available and is advanced by members of the Jini Community through the open Jini Community Decision Process. This lecture might be useful to understand why and how JINI^{TM} TECHNOLOGY is used in SORCER to manage dynamic federations of services.

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 [8]. 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 [22] Grid resource broker scheduler or the Condor-G [4], [38] high-throughput scheduler. Both rely on Globus/GRAM [8] (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 a series of programs ("jobs") on a computer without human interaction in the era when operating systems were not yet developed.

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 usually applied to a Grid computing module as is applied to single computer module while 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 are 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 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.

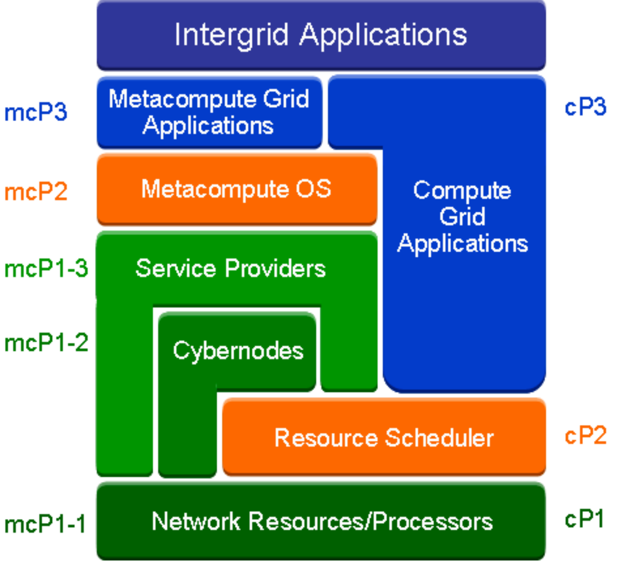


Fig. 1 Three types of Grids: compute grid, metacompute grid, and Intergrid. Platform layers: P1 resources, P2 resource management, P3 programming environment. A cybernode provides a lightweight dynamic virtual processor, turning heterogeneous compute resources into homogeneous services available to the 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) [7], [25], [28], [29]. 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) [35], [31], [34], [33], [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 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 [5]). In SPOA, by contrast, a passive service description is

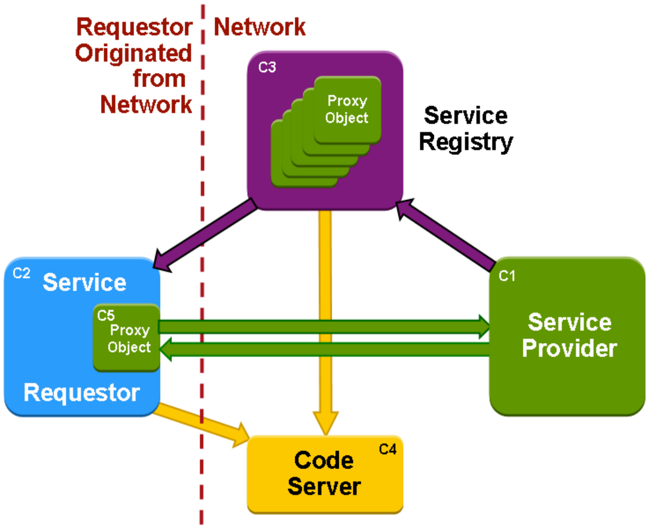


Fig. 2 Service object-oriented architecture

registered (e.g., an XML document in WSDL for Web/Globus services [21], [37], 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/Globus services, IIOP in CORBA [26]). 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 Globus services cannot change the communication protocol between requestors and providers while the SOOA approach is protocol neutral [41]. 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 (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 [14]. 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 Globus 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[6]? We cannot just take an object-oriented program developed without distribution in mind and make it a distributed system, ignoring the unpredictable network behavior. Most RPC systems, except Jini [14], 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/Globus service define distributed objects, but do not have anything common with object-oriented distributed systems.

Exertion-based programming [31], [29] was introduced to handle the metasystemic distribution in SORCER by using indirect remote method invocation with no service provider explicitly specified in a network request called an exertion. Specific 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 [14] in the network and the Jini programming model [5] with explicit leases, distributed events, transactions, and discovery/join protocols. While Jini [15], [16] focuses on service management in a networked environment, SORCR 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 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 [42] 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—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) can be created interactively [32] and allow for a dynamic federation to transparently coordinate their execution within the grid. The exertion federation can be interactively monitored and exertions debugged during execution [34], [20]. 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

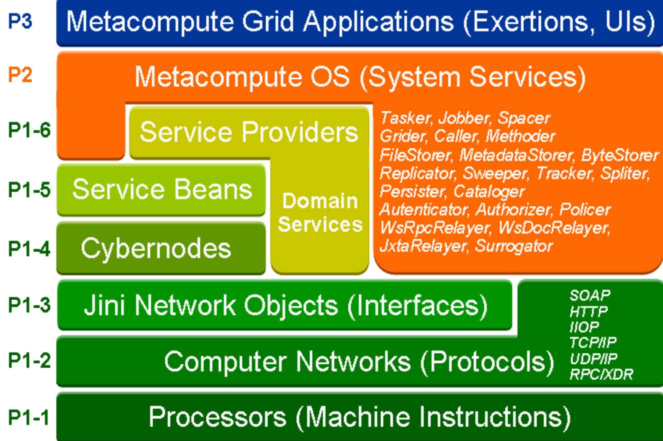


Fig. 3 SORCER layered platform, where P1 resources, P2 resource management, P3 programming environment

allow for reusability, scalability, reliability, and load balancing.

Each SORCER provider (peer) implementing the common Servicer interface, offers services to other peers [31] 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 [33], [1], [2], [39]
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 [9]
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 [32] or programmatically (using SORCER APIs), and its execution can be monitored and debugged [34], [20] in the overlay service network via service user interfaces (Service UI [40]) attached to providers and installed on the fly by service browsers [13]. 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 [31], [12] 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 [23] 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 [31].

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 DROP) depending on the exertion's interface defined in its

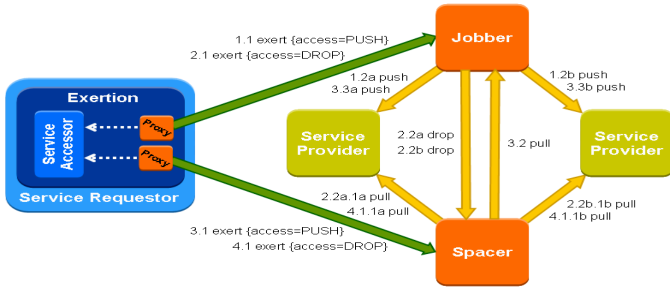


Fig. 4 Push vs. Pull exertion computing

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, SORCER supports a traditional approach to grid computing similar to those found in Condor [4] and Globus [37]. 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 [31]. 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) 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 [18], 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], [2] was designed and developed to provide data access for metaprograms. It complements the file store developed for FIPER [33] 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. SILENUS service 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 [39] 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 [25], [27], [3], [10], [11], [17], [19]. 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 [35] in collaboration with Air Force Research Lab, WPAFB.

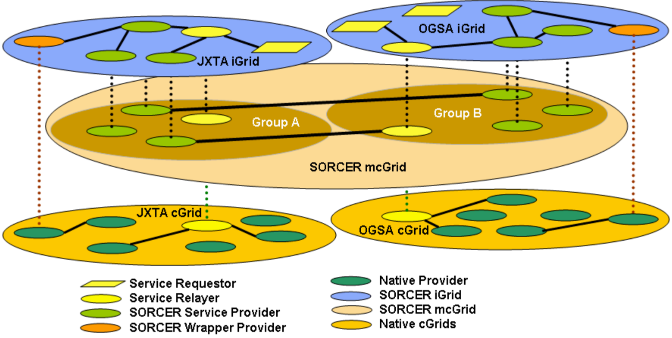


Fig. 5 Integrating and wrapping cGrids with SORCER mcGrids. Two requestors, one in JXTA iGrid, one in OGSA iGrid submits exertion to a corresponding relayer. Two federations are formed that include providers from all the two horizontal layers below the iGrid layer (as indicated by continues and dashed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

Acknowledgment

I would like to thank all my students in the SORCER Research Group [36] 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] Berger, M., Sobolewski, M., SILENUS – A Federated Service-oriented Approach to Distributed File Systems, In Next Generation Concurrent Engineering [30]. pp. 89-96, 2005.

[2] 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.

[3] 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.

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

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

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

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

[8] 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.

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

[10] 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.

[11] Goel, S, Talya S., and Sobolewski, M., Preliminary Design Using Distributed Service-based Computing, In Next Generation Concurrent Engineering [30], pp. 113-120, 2005.

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

[13] 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.

[14] 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.

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

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

[17] 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.

[18] Khurana V., Berger M., Sobolewski M., A Federated Grid Environment with Replication Services. In Next Generation Concurrent Engineering [30], pp. 97-103, 2005.

[19] 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.

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

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

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

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

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

[25] 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.

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

[27] 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

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

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

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

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

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

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

[38] 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.

[39] 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)

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

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

[42] 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.

Metacomputing with Federated Method Invocation

Michael Sobolewski

Texas Tech University
SORCER Research Group
http://sorcer.cs.ttu.edu
sobol@cs.ttu.edu

Abstract

Six generations of RPC systems can be distinguished including Federated Method Invocation (FMI) presented in this paper. Some of them—CORBA, Java RMI, and Web/Globus services—support distributed objects. However, creating object wrappers implementing remote interfaces doesn't have a great deal to do with object-oriented distributed programming. Distributed objects developed that way are usually ill-structured with missing core object-oriented traits: encapsulation, instantiation, inheritance, and network-centric messaging by ignoring the real nature of networking. A distributed system is not just a collection of distributed objects—it's the *network* of objects. In particular, the object wrapping approach does not help to cope with network-centric messaging, invocation latency, object discovery, dynamic object federation, fault detection, recovery, partial failure, etc. The Jini™ architecture does not hide the network; it allows the programmer to deal with the network reality: leases for network resources, distributed events, transactions, and discovery/join protocols to form federations. A service-oriented architecture presented in this paper implements FMI to support metaprogramming. The triple Command pattern implantation uses Jini service management and Rio dynamic provisioning for managing the *network* of FMI objects.

Categories and Subject Descriptors C.2.4 [**Distributed Systems**]: Distributed Applications, D.1.3 [**Concurrent Programming**]: Distributed Programming, D.2.11 [**Software Architectures**]: Domain Specific Architectures, D.2.2 [**Design Tools and Techniques**]: Object-oriented design methods.

General Terms Design, Experimentation, Languages.

Keywords object oriented distributed programming; service oriented architectures; federated service object programming, metacomputing;

1. Introduction

Socket-based communication forces us to design distributed applications using a read/write (input/output) interface, which is not how we generally design non-distributed applications based on procedure call (request/response) communication. In 1983, Birrell and Nelson devised remote procedure call (RPC) [2], a mechanism to allow programs to call procedures on other hosts. So far, six RPC generations can be distinguished:

1. First generation RPCs – Sun RPC (ONC RPC) [24] and DCE RPC, which are language, architecture, and OS independent;
2. Second generation RPCs – CORBA [25] and Microsoft DCOM-ORPC, which add distributed object support;
3. Third generation RPC – Java RMI [21] is conceptually similar to the second generation but supports the semantics of object invocation in different address spaces that are built for Java only. RMI fits cleanly into the language with no need for standardized data representation, external interface definition language, and with behavioral transfer that allows remote objects to perform operations that are determined at runtime;
4. Fourth generation RPCs – Jini Extensible Remote Invocation (Jini ERI) [20] with dynamic proxies, smart proxies, network security, and with dependency injection defining exporters, end points, and security;
5. Fifth generation RPCs – Web/Globus Services RPC [18,35] and the XML movement;
6. Sixth generation RPC – Federated Method Invocation (FMI), presented in this paper, allows for network invocations on multiple federating hosts (virtual metacomputer) in the SORCER environment [33].

All the RPC generations are based on a form of service-oriented architecture (SOA) discussed in Section 2. However, CORBA, RMI, and Web/Globus services are in fact

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

object-oriented wrappers of network interfaces that hide distribution and ignore the real nature of network through classical abstractions of object-oriented programming using existing network technologies. The fact that object-oriented languages are used to create these object wrappers doesn't mean that developed distributed objects have a great deal to do with object-oriented distributed programming. For example, CORBA defines many services, and implementing them using distributed objects does not make them well structured with core object-oriented traits: encapsulation, instantiation, inheritance, and network-centric messaging. Similarly in RMI, marking objects with the Remote interface does not help to cope with network-centric messaging, object discovery, dynamic federation, fault detection, recovery, partial failure, etc.

Building on the object-oriented distributed paradigm is the Federated Service Object-Oriented (FSOO) paradigm exemplified by the Jini architecture [13] in which the network objects come together on the fly to play their predefined roles. In the Service-ORiented Computing EnviRonmet (SORCER) developed at Texas Tech University [33], a service provider is a remote object that accepts network requests—called *exertions*—from service requestors to execute an elementary item of work called a *service task* or a composite item of work called a *service job*. An exertion, either a task or job, can federate on multiple hosts according to its encapsulated data, operations, and control strategy.

An exertion submitted to any provider in SORCER becomes an executing FSOO program that is dynamically bound to all relevant and currently available service providers on the network. The providers that dynamically participate in this invocation are collectively called an *exertion federation*. This federation is also called a *virtual metacomputer* since federating services are located on multiple physical compute nodes held together by the FSOO infrastructure so that, to the individual exertion requestor, it looks and acts like a single computer.

The SORCER environment provides the means to create interactive FSOO programs [29] and execute them using the SORCER runtime infrastructure presented in Section 3. Exertions can be created using interactive user interfaces downloaded on the fly from service providers. Using these interfaces, the user can execute and monitor the execution of exertions within the FSOO metacomputer. The exertions can be persisted for later reuse, allowing the user to quickly create new applications or programs on the fly in terms of existing exertions.

SORCER is based on the evolution of concepts and lessons learned in the FIPER project [5,26,27], a $21.5 million program founded by NIST (National Institute of Standards and Technology). Initial exertion-based programming concepts introduced in FIPER have been practically used in many concurrent engineering applications [29,8,9,16,23]. Academic research on exertion-oriented programming has been established at the SORCER Laboratory, TTU, [33] where twenty SORCER related research studies have been investigated so far [34]. The current version of FMI used in SORCER is described in this paper.

The paper is organized as follows. Section 2 provides a brief description of a service oriented architecture with a related discussion of distribution transparency; Section 3 describes the SORCER methodology; Section 4 presents federated method invocation; Section 5 provides concluding remarks.

2. SOA and Distribution Transparency

Various definitions of a Service-Oriented Architecture (SOA) leave a lot of room for interpretation. In general terms, SOA is a software architecture using loosely coupled software services that integrates them into a distributed computing system by means of service-oriented programming. Service providers in the SOA environment are made available as independent service components that can be accessed without a priori knowledge of their underlying platform or implementation. While the client-server architecture separates a client from a server, SOA introduces a third component, a service registry, as illustrated in Figure 1. 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 ob-

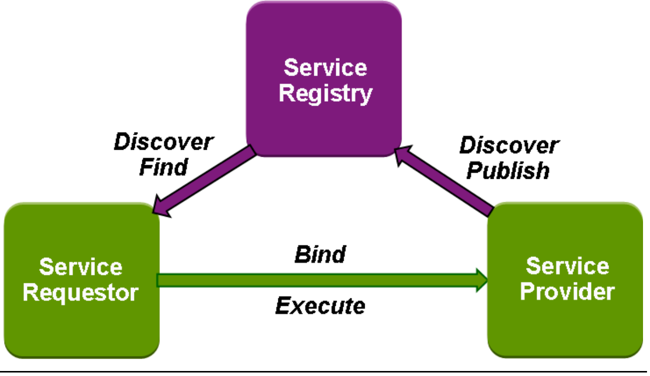


Figure 1. Service oriented architecture.

jects, from *service protocol oriented architectures* (SPOA), where a communication protocol is fixed and known beforehand by the provider and requestor. Based on that protocol and a service description obtained from the service registry, the requestor can bind to the service provider by creating a proxy used for remote communication over the fixed protocol. In SPOA a service is usually identified by a name. If a service provider registers its service description by name, the requestors have to know the name of the service beforehand.

In SOOA, a proxy—an object implementing the same service interfaces as its service provider—is registered with the registries and it is always ready for use by requestors. Thus, in SOOA, the service provider publishes the proxy as the active surrogate object with a codebase annotation, e.g., URLs to the code defining proxy behavior (RMI and Jini ERI). In SPOA, by contrast, a passive service description is registered (e.g., an XML document in WSDL for Web/Globus services, or an interface description in IDL for CORBA); the requestor then has to generate the proxy (a stub forwarding calls to a provider) based on a service description and the fixed communication protocol (e.g., SOAP in Web/Globus 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 Globus services cannot change the communication protocol between requestors and providers while the SOOA approach is protocol neutral [38]. In SOOA, how 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 (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 (see Figure 1 and 2). SORCER uses dynamic discovery protocols to locate available registries (lookup services) as defined in the Jini architecture [12]. Neither the requestor who is looking up a proxy by its interfaces nor the provider registering a proxy needs to know specific locations. In SPOA, however, the requestor and provider usually do need to know the explicit location of the service registry—e.g., the IP address of an ONC/RPC portmapper, a URL for RMI registry, a URL for UDDI registry, an IP address of a COS Name Server—to open a static connection and find or register a service. In deployment of Web and Globus 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.

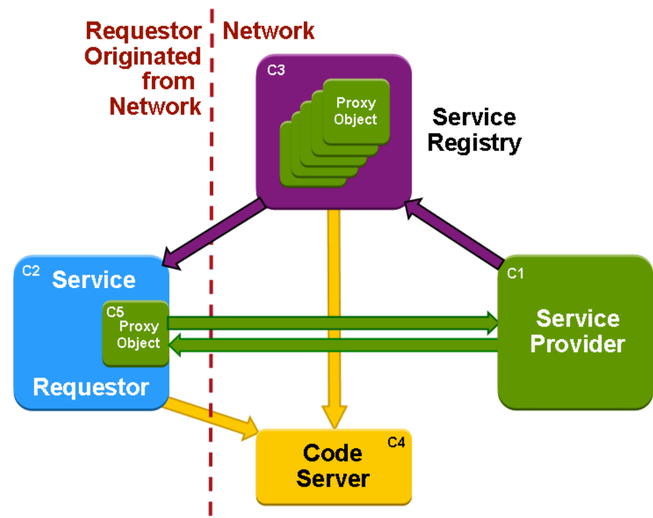


Figure 2. Service object-oriented architecture.

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 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? [4] We cannot just take an object-oriented program developed without distribution in mind and make it a distributed system, ignoring the unpredictable network behav-

ior. Most RPC systems, except Jini [3], 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 not just an object-oriented implementation of a single distributed object; it's a metasystemic issue in object-oriented distributed programming.

Exertion-based programming was introduced [27] to handle the metasystemic distribution in SORCER by using indirect remote method invocation with no service provider explicitly specified in the network request (exertion). Specific infrastructure objects support exertion-oriented programming combined with FMI. 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.

3. Federated Service Object-oriented Computing Environmet: SORCER

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 [12] in the network and Jini programming model with explicit leases, distributed events, transactions, and discovery/join protocols. While Jini focuses on service management in a networked environment, SORCR is focused on exertion-oriented programming and the execution environment for exertions.

As described in Section 2, 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 protocols to obtain service proxies on the network. SORCER uses Jini discovery/join protocols to implement its FSOOA and FMI.

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*. 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 representing the metaprogram parameters [39]. The service context describes the data that tasks and jobs work on. Exertion-oriented programs (metaprograms) can be created interactively [29] and allow for a dynamic federation to transparently coordinate their execution within the grid. Please note that these meta-

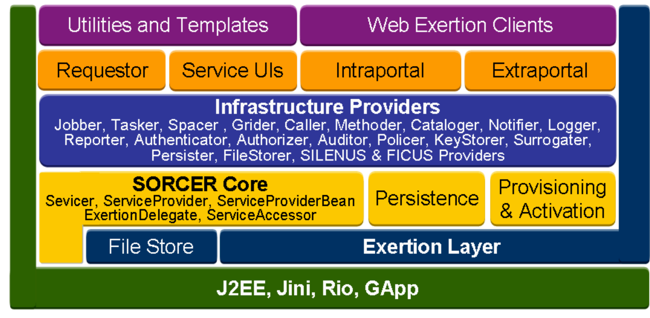


Figure 3. SORCER layered functional architecture.

computing 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.

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 offers services to other peers [19] 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 [26,27]. 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.

An exertion can be created interactively [29] or programmatically (using SORCER APIs) and their execution can be monitored and debugged in the overlay service network [32]. 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. Specialized providers within the federation, or task peers, execute service tasks. Jobs are coordinated by a *rendezvous* or *job peer* called a *Jobber*, one of SORCER infrastructure services [26]. 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 one of available job peers 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. Also, SORCER supports a traditional approach to grid computing similar to those found in Condor [36] and Globus [35]. 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.

Grid-based services in the SORCER environment include *Grider* services collaborating with Jobber services for traditional grid job submission, and *Caller* and *Methoder* services for task execution [15]. Callers execute conventional programs via a system call as described in the service context of a 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. A functional layered SORCER architecture is presented in Figure 3.

4. Federated Method Invocation (FMI)

Each programming language provides a specific computing abstraction. Procedural languages are abstractions of assembly languages. Object-oriented languages abstract elements in the application domain that refer to "objects" as their representation in the corresponding solution space. The object-oriented distributed programming should allow us to describe the distributed problem in terms of the intrinsic unpredictable network problem instead of in terms of distributed objects that hide the notion of the network.

What intrinsic distributed abstractions are defined in SORCER? Well, *service providers* are "objects", but they are specific objects—they are *network objects* with a *network state*, *network behavior*, and *network type(s).* There is still a connection to distributed objects: each service provider looks like a distribute object (compute node) in that it has a network state, network behavior, and network types(s). Service providers act also as *network peers*[19]*;* they are replicated and dynamically provisioned for reliability to compensate for network failures [22]. They can be found dynamically in runtime by type(s) they implement. They can federate for executing a specific network request called an *exertion* and perform hierarchically nested (component) exertions. An exertion encapsulates service data, operations, and control strategy. Once the exertion's invocation is complete, the federation dissolves and the providers disperse to seek other exertions. The exertion can incorporate multiple nested exertions where a precedence relation is defined by a parent-child relationship. The same provider can perform multiple exertions concurrently and any provider that implements the matching service type can be selected for performing the exertion associated with this type. The component exertions may need to share context data of ancestor exertions, and the top-level exertion is complete only if all nested exertions are successful.

With that very concise introduction to the abstractions of exertion-based programming, let's look in detail at how Federated Method Invocation (FMI) is structured and how it works with exertions.

4.1 Service Messaging and Exertions

In object-oriented terminology, a message is the single means of passing control to an object. If the object responds to the message, it has an operation and its

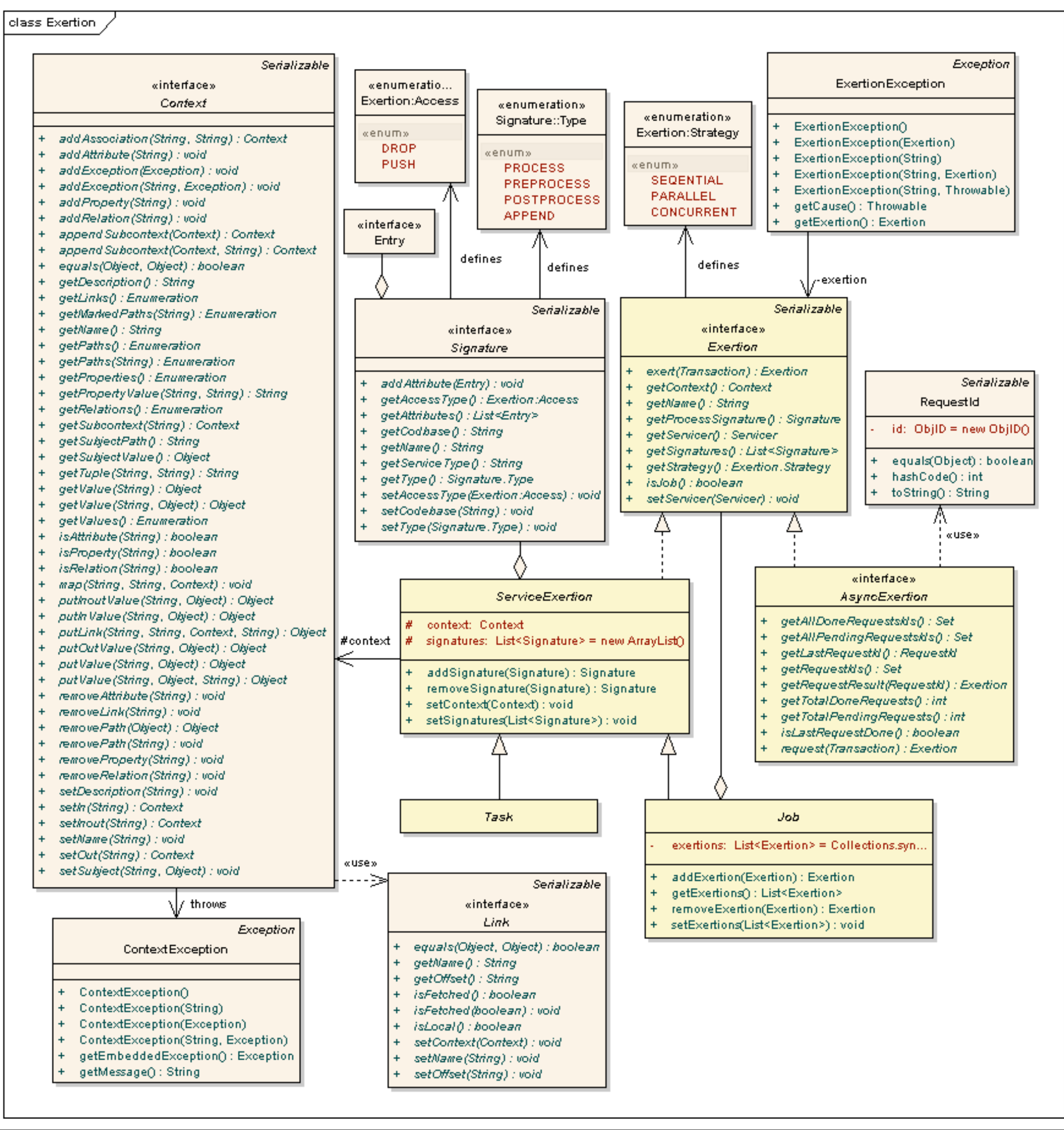


Figure 4. The Exertion interface and related subset of FMI interfaces/classes: the abstract class ServiceExertion with two abstract subclasses: Task and Job along with FMI parameters defined by the Context interface and signatures defined by the Signature interface.

implementation (method) for that message. The equivalent in procedural programming languages to a message is the function call. The message means neither the function as it is nor the signature of the function, but to send the message means roughly to call the function. Because object data is encapsulated and not directly accessible, a message is the only way to send data from one object to another. Each message specifies the name of the receiving object, the name (selector) of operation to be invoked, and any paever, in the unreliable network of objects, the receiving object might not be present or can go away at any time. Thus, we should postpone receiving object identification as late as

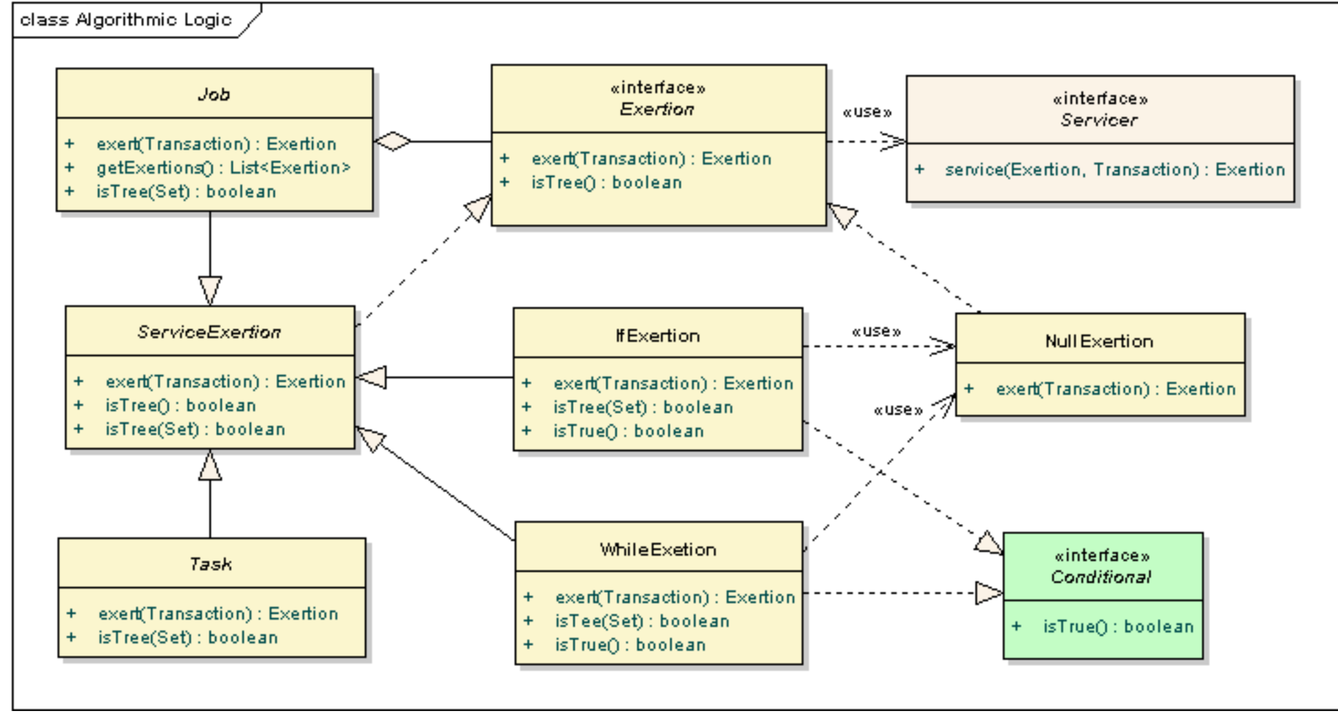


Figure 5. Flow control exertions, conditional IfExertion and looping WhileExertion, used in SORCER.

possible. Grouping related messages per one request for the same data set makes a lot of sense due to network invocation latency and common errors in handling. These observations lead us to service-oriented messages called *exertions* that encapsulate both multiple *service signatures* and data as a s*ervice context*. In other words, an exertion primarily consists of one or more operations and the data upon which the operations should be performed. Two exertion types are distinguished: elementary and composite exertion called *service task* and *service job* respectively (see Figure 4). There are two ways of invoking exertions. In the first case, an Exertion can be invoked by calling Exertion.exert(Transaction). The second way is explained in Subsection 4.6.

4.2 Service Signatures

An exertion initiates the dynamic federation of all needed service providers dynamically—as late as possible—as specified by signatures of top-level and nested exertions. Thus, FMI is defined as exerting an exertion, which is essentially an indirect invocation of network methods specified by the exertion signatures and service context. SORCER service providers and requestors usually communicate via FMI.

A service Signature is defined by:

- signature name
- service type – Java interface name
- selector of the service operation – operation name of the service type (Java interface)
- operation type – Signature.Type: PROCESS (default), PREPROCESS, POSTPROCESS
- service access type – Signature.Access; PUSH (default) direct binding to Jobbers or Taksers, or DROP using Spacer (see Figure 4)
- priority
- execution time flag – if true, the execution time is returned in the service context
- notifyees – list of email addresses to notify upon completed)
- service attributes – requestor's attributes matching provider's registration attributes

An exertion can comprise of a collection of PREPRROCESS and POSTPROCESS signatures, but only one PROCESS signature. The PROCESS signature defines the binding provider for the exertion.

4.3 Exertion Types

A Task is the analog of a statement in conventional programming languages—here an elementary step of the exertion-oriented program. Thus, it is a minimal unit of structuring in exertion-oriented programming. If the provider responds to a Task, it has a method for the task's PROCESS signature. Other signatures associated with the Task provide for preprocessing and postprocessing by the same or federating providers. An APPEND signature provides for the context received from the provider identified by this signature to be appended in runtime to the task's currently processed service context. Appending a service

context allows a requestor to use actual data in runtime not available to the requestor when a task is submitted. A Task is the single means of passing control to a PROCESS provider. Note that a task is a batch of operations that operate on the same service context—a Task shared execution state—and all operations of the Task, as defined by signatures, can be executed by the same provider or a group of federating providers coordinated by the PROCESS provider—the provider identified by the PROCESS signature of the Exertion.

A Job is the analog of a procedure in conventional programming languages—here a federated procedure in an exertion-oriented program. It is a composite of exertions (see Figure 4) that makeup the federated procedure. The following flow control exertion types define algorithmic logic of exertion-oriented programming:

- *Exertion*
  - NullExertion
  - *AsyncExertion*
    - AsyncServiceExertion
  - ServiceExertion
    - ServiceTask
    - ServiceJob
    - IfExertion
    - WhileExertion
    - ForExertion
    - DoExertion
    - ThrowExertion
    - TryExertion
    - BreakExertion
    - ContinueExertion

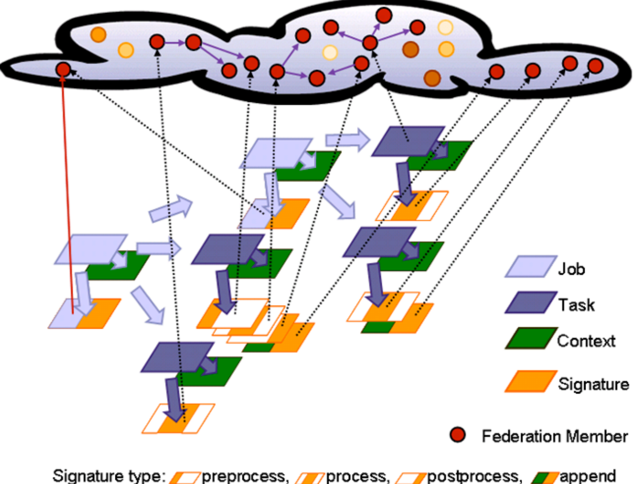


Figure 6. A job federation. The red line (the first from the left) indicates the originating FMI invocation: Exertion.exert(Transaction) or Servicer.service(Exertion, Transaction). The root job with component exertions is depicted below the provider grid (a cloud). Late bindings of all signatures are indicated by dashed lines that define the job's initial federation (metcomputer).

Currently implemented flow control Exertion types in SORCER are depicted in Figure 5.

4.4 Knowledge Representation and Service Context

The implementation of natural language knowledge definition and editing critically depends on the intricacy of translation between natural language constructs and internal knowledge representation structures. This is a function of the chosen knowledge representation method. In the percept formalism, an entity of the world is treated as the image given by perception, and that image is called a percept. A percept conceptualization is the semantic counterpart of the syntactic level of the knowledge description theory called percept calculus [30]. A service context, based on the percept conceptualization, is a data structure that describes service provider ontology along with related data. A service ontology is controlled by provider vocabulary that describes objects and the relations between them in a provider's namespace within a specified service domain of interest. A requestor submitting an exertion to a provider has to comply with that ontology. In the percept conceptualization, attributes and their values are used as atomic conceptual primitives, and complements are used as molecular ones. A complement is an attribute sequence (path) with a value at the last position. An elementary percept property consists of a percept subject and a set of percept complements, and usually corresponds to a simple sentence of natural language.

A service context is a tree-like structure described conceptually in the EBNF conceptual syntax specification as follows:

1. context = [subject ":"] complement { complement }.
2. subject = element.
3. complement = element ";".
4. element= path ["=" value].
5. path = attribute { "/" attribute } [{ "<" association ">" }] [{ "/" attribute }].
6. value = object.
7. attribute = identifier.
8. relation = domain product.
9. association = domain tuple.
10. product = attribute { "|" attribute }.
11. tuple = value { "|" value }.
12. attribute = identifier.
13. domain = identifier.
14. association = identifier.
15. identifier = letter { letter | digit }.

A relation with a single attribute is called a *property* and is denoted as attribute | attribute.

To illustrate the idea of context, lets consider the following context example (graphically depicted in Figure 7):

```
laboratory/name = SORCER: university=TTU;
university/department/name=CS;
```

```
university/department/room/number=20B;
university/department/room/phone/number=806-742-
university/department/room/phone/ext=237;
director <person | Mike | W | Sobolewski>
         /email=sobol@cs.ttu.edu;

person | firstname | initial | lastname.
```

A context leaf node, or *data node* is where the actual data resides. The service context—all context paths—denotes an application domain namespace, and a *context model* [39] is its context with data nodes appended to its context paths. A context path is a hierarchical name for a data item in a leaf node. Note that Context can be represented as an XML document—what has been done in SORCER for interoperability—but the power of object Contexts comes from the fact that any Java object can be naturally used as a data node. In particular exertions themselves can be used as data nodes and then executed and controlled by providers to run complex iterative programs, e.g., nonlinear multidisciplinary optimization [16].

4.5 Service-to-Service (S2S) Computing

Tasks are usually executed by providers of the Tasker type (task peer). A Job contains a service context called *control context* that describes the control strategy for the Job. Dedicated service providers of the Jobber type (job peer also called rendezvous peer), interpret and execute a job's control context in terms of the job's nested exertions accordingly. A Jobber manages a shared context (shared execution state) for the job federation and provides a substitution for input context parameter mappings. A Jobber creates a federation of required service providers (Taskers and Jobbers) in runtime. A SORCER peer (Servicer) that is unable to execute an Exertion for any reason forwards the Exertion to any available Servicer matching the exertion's PROCESS signature and returns the resulting exertion back to its requestor. In Figure 6, a job federation is illustrated with late bindings for all signatures in all component exertions.

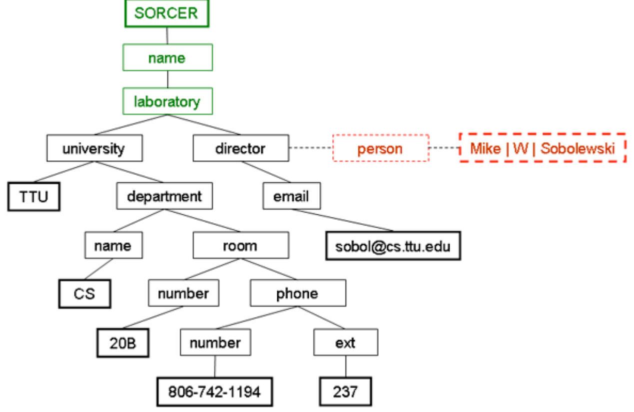


Figure 7. Example of a service context.

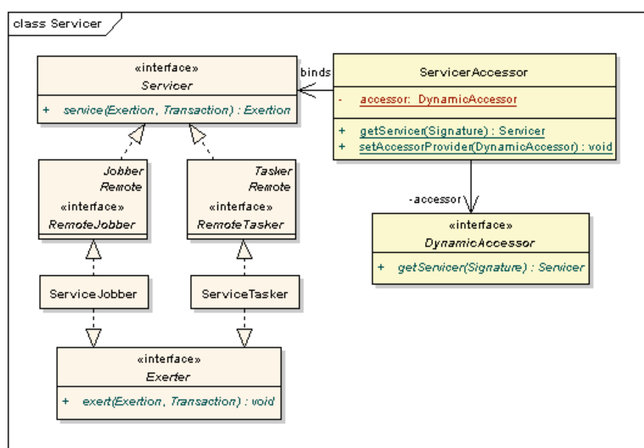


Figure 8. FMI Servicers: Tasker and Jobber with the name service provider interface—DynamicAccessor.

All SORCER service providers are service peers as they implement the top-level Servicer interface (see Figure 8). As a result, each Servicer can initiate a federation created in response to Servicer.service(Exertion, Transaction). Servicers come together to form a federation participating in execution of the same exertion. When the exertion is complete, Servicers leave the federation and seek a new exertion to join. Note that the same exertion can form a different federation for each execution due to the dynamic nature of looking up Servicers by their implemented custom interfaces. The hierarchy of SORCER Servicer types is defined as follows (see Figure 8, interfaces names in italic below):

- *Servicer* (defines S2S)
  - *Tasker*
  - *Jobber*
  - *Provider* extends *Remote* and *Monitorable*
    - *AdministrableProvider*
      - ServiceProvider (implements discovery, join, and monitoring)
        - ServiceTasker (implements *Tasker* and *Exerter*)
        - ServiceJobber (implements *Jobber* and *Exerter*))
- *Exerter* (not *Remote*)

ServiceAccessor uses DynamicAccessor as a naming service provider for FMI. The naming service provider furnishes a means to dynamically locate service providers on the network. The SORCER ProviderAccessor implements DynamicAccessor using the SORCER Cataloger service with the Jini Discovery and Lookup Services.

Despite the fact that every Servicer can accept any exertion, Servicers have well defined roles in SORCER S2S exertion-oriented programming (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 [31,1]
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 [7]
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
l) Generic ServiceTasker and ServiceJobber implementations are used to configure domain specific providers via dependency injection—configuration files for smart proxying and inserting business objects called service beans.

4.6 FMI Triple Command Pattern

Polymorphism lets us encapsulate a request—in FMI an exertion—then establish the signature of operation to call and vary the effect of calling the underlying operation by varying its implementation. The Command design pattern [10] establishes an operation signature as an interface and defines various implementations of the interface. In FMI, the following three operations are defined:

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

The Triple Command pattern defines various implementations of these interfaces: Exertion, Servicer, and Exerter. This approach allows for the P2P environment [8] 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. Note that both ServiceTasker and ServiceJobber are Servicers and Exerters (see Figure 8 for details); more precisely their proxies are remote objects of the Servicer type only while the provider itself (local object) is both of Servicer and Exerter type.

FMI 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 (see Figure 8 for classes involved).
2. If the matching provider is found, then on its access proxy (that 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 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.

In the FMI approach, a requestor can create any Exertion, composed from any hierarchically nested Exertions, with any service provider supplied anthology. The context anthologies along with object proxies and their object attributes are network-centric; they are part of the provider's registration so can they be accessed via Cataloger or lookup services by any requestor on the network, e.g., service browsers [11] or custom service UI user agents [37] providing interactive exertion-oriented programming. In SORCER, using these zero-install service UIs, the user can define data for downloaded ontology and create a task/job to be executed on the virtual metacomputer.

Individual Providers, in particular Taskers and Jobbers, implement their own exert(Exertion, Transaction) methods according to their service semantics, in SORCER implemented by ServiceTasker and ServiceJobber respectively. SORCER specific domain providers either subclass ServiceTasker or ServiceJobber, or by dependency injection (using Jini configuration methodology) configure either one with one of 12 proxying methods developed in SORCER. In general, many different types of taskers and jobbers can be used in SORCER at the same time (currently one ServiceTasker and one ServiceJobber implementation exists) and exertions via their signatures will make appropriate choices as to what virtual metacomputer to run.

Invoking an exertion, let's say `ext`, is similar to invoking an executable program `ext.exe` at the command prompt. If we use the Tenex C shell (`tcsh`), invoking the program is equivalent to: `tcsh ext.exe`, i.e., passing the executable `ext.exe` to `tcsh`. Similarly, to invoke a metaprogram using FMI, in this case the exertion `ext`, we call `ext.exert(null)` if

no transactional semantics is required. Thus, the exertion is the metaprogram and the network shell at the same time, which might first come as a surprise, but close evaluation of this fact shows it to be consistent with the meaning of object-oriented federated programming. Here, the *virtual metacomputer* is a federation that does not exist when the exertion is created. Thus, the notion of the *virtual metacomputer* is enclosed in the exertion exemplified by FMI.

The observation concluding that the exertion is the metaprogram and the network shell at the same time brings us back to the distribution transparency issue discussed in Section 2. It might appear that Exertion objects are network wrappers as they hide network intrinsic unpredictable behavior. However, Exertions are not distributed objects, as do not implement any remote interfaces; they are local objects. Servicers are distributed objects and there are many types of Servicers addressing different aspects of networking. The network intrinsic unpredictable network behavior is addressed by the SORCER object-oriented distributed infrastructure: Taskers, Jobbers, Catalogers, Spacers, FileStorers, Authenticators, Authorizers, Policers, etc. The Servicer-based infrastructure facilitates exertion-oriented programming and metaprograms execution using presented FMI and allows for constructing reliable object oriented distributed systems from unreliable distribute components - Servicers.

5. Conclusions

A distributed system 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 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 are 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. FMI allows for the P2P environment via the Service interface, extensive modularization of Exertions and Exerters, and extensibility from the triple command design pattern. The presented FMI has been successfully tested in multiple concurrent engineering, large-scale distributed applications.

Acknowledgments

I would like to thank all my students in the SORCER Research Group [34] for their motivation, innovation, and excitement they generate when working with exertion-based programming. Without their research efforts, it would not be possible to integrate so many different views of distributed programming and to validate so many diverse and controversial opinions on distributed objects and object-oriented distributed computing.

References

[1] Berger, M., and Sobolewski, M., SILENUS – A Federated Service-oriented Approach to Distributed File Systems, In Next Generation Concurrent Engineering [28]. pp. 89-96 (2005)

[2] Birrell, A. D. & Nelson, B. J., Implementing Remote Procedure Calls, XEROX CSL-83-7, October 1983.

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

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

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

[6] Foster I., Kesselman C., Tuecke S., The Anatomy of the J. Supercomputer Applications, 15(3) (2001)

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

[8] 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)

[9] Goel, S, Talya S., and Sobolewski, M., Preliminary Design Using Distributed Service-based Computing, Proceeding of the 12th Conference on Concurrent Engineering: Research and Applications, ISPE, Inc., pp. 113-120 (2005)

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

[11] 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.

[12] Jini architecture specification, Version 2.1. Available at: http://www.sun.com/software/jini/specs/jini1.2html/jini-title.html. Accessed on: March 15, 2007 (2001)

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

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

[15] Khurana V., Berger M., Sobolewski M., A Federated Grid Environment with Replication Services. In Next Generation Concurrent Engineering [28].

[16] 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)

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

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

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

[20] Package net.jini.jeri. Available at: https://java.sun.com/products/jini/2.1/doc/api/net/jini/jeri/package-summary.html. Accessed on: March 15, 2007.

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

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

[23] 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)

[24] RPC: Remote Procedure Call Protocol Specification Version 2. Available at: http://www.ietf.org/rfc/rfc1831.txt (1995). Accessed on: March 15, 2007.

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

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

[27] 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.

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

[29] 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)

[30] Sobolewski, M., Percept Conceptualizations and Their Knowledge Representation Schemes, Ras Z.W. and Zemankova M. (Eds.) Methodologies for Intelligent Systems, Lecture Notes in AI 542, Berlin: Springe-Verlag, pp. 236-245 (1991)

[31] 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)

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

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

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

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

[36] Thain D., Tannenbaum T., Livny M.. Condor and the Grid. In Fran Berman, Anthony J.G. Hey, and Geo rey Fox, editors, Grid Computing: Making The Global Infrastructure a Reality. John Wiley (2003)

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

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

[39] 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)

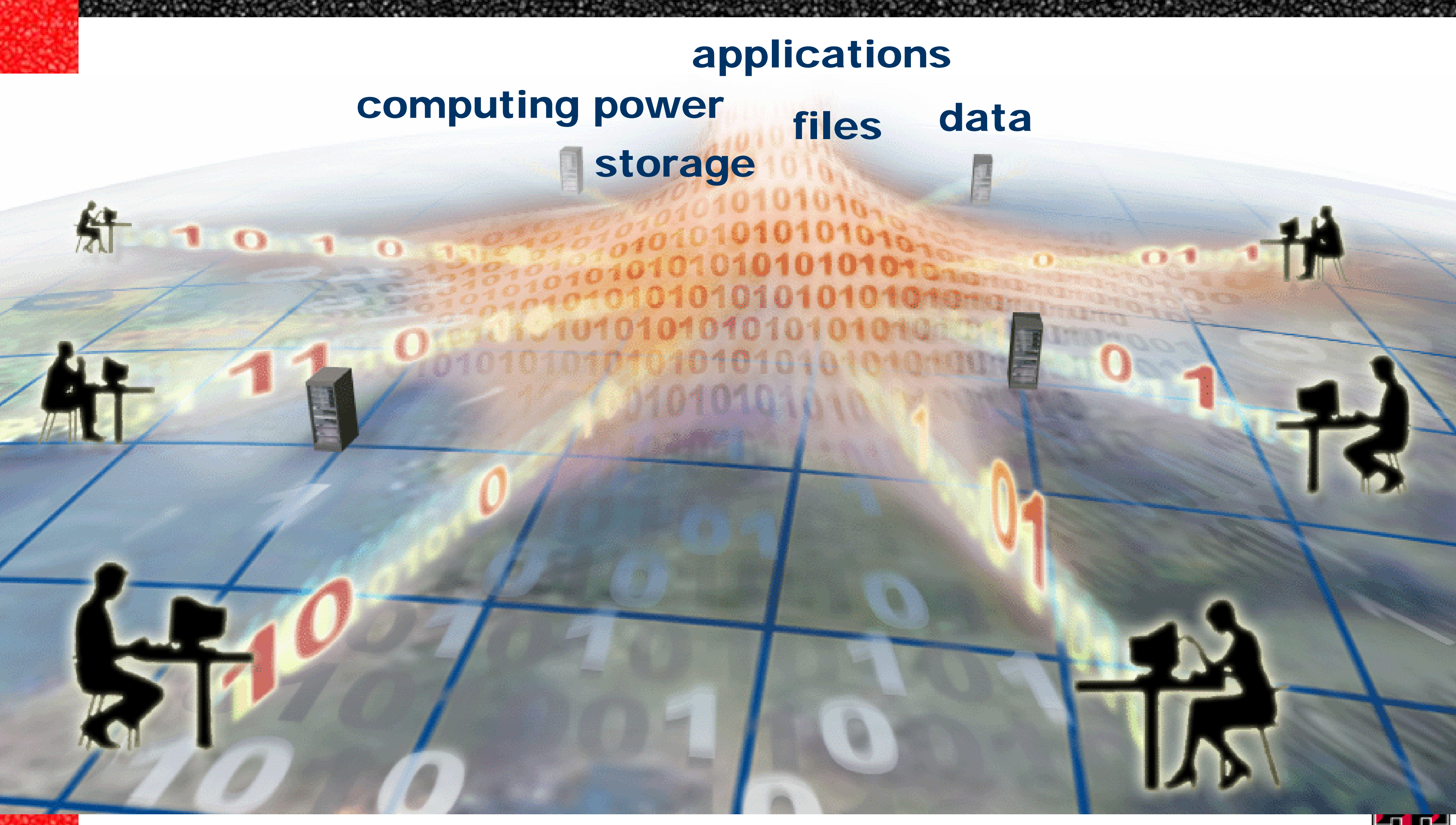
TEXAS TECH UNIVERSITY

Service Proxying with Dependency Injection

SORCER Seminar
Sept 28, 2006

Michael Sobolewski
Computer Science, TTU
sobol@cs.ttu.edu
http://sobol.cs.ttu.edu

From the Internet ….



…. to Metacomputing

SOA = SPOA + SOOA

Service Registry

Discovery
Find

Discovery
Publish

Service Requestor

Bind

Execute

Service Provider

SOOA - Network Centricity

Requestor Originated from Network

Network

C2
Mobile Proxy Object

Service Registry

C0
Service
C4
Proxy Object
C5
Requestor

C1
Service Provider

C3
Code Server

Proxy Types

1. Static proxy – created explicitly before it is used (rmic)
2. Dynamic proxy – no need to create it statically in advance (RMI vs. Jini ERI)
3. Remote proxy – shields the requestor object from the fact that the underlying object is remote
4. Access (protection) proxy – enforces a security policy on access to a service or data-providing object (Sevicer)
5. Façades – a façade grants access to multiple underlying objects (Servicer + AdminProxy + server)
6. Virtual proxy – performs lazy initialization of expensive back-ends (Service UI – UIDesrcriptor)
7. Smart proxy - grants access to local (fat) and remote resources
8. Bootstrap proxy – trusted proxy, created from local codebase (proxy verification)
9. Hybrid proxy – combination of the above types

Intergrid

SORCER.jxta

SORCER.ogsa

SORCER.grid

SORCER.grid

SORCER.core

JXTA

OGSA

Service Requestor
Service Broker
SORCER Service Provider
SORCER Wrapper Provider

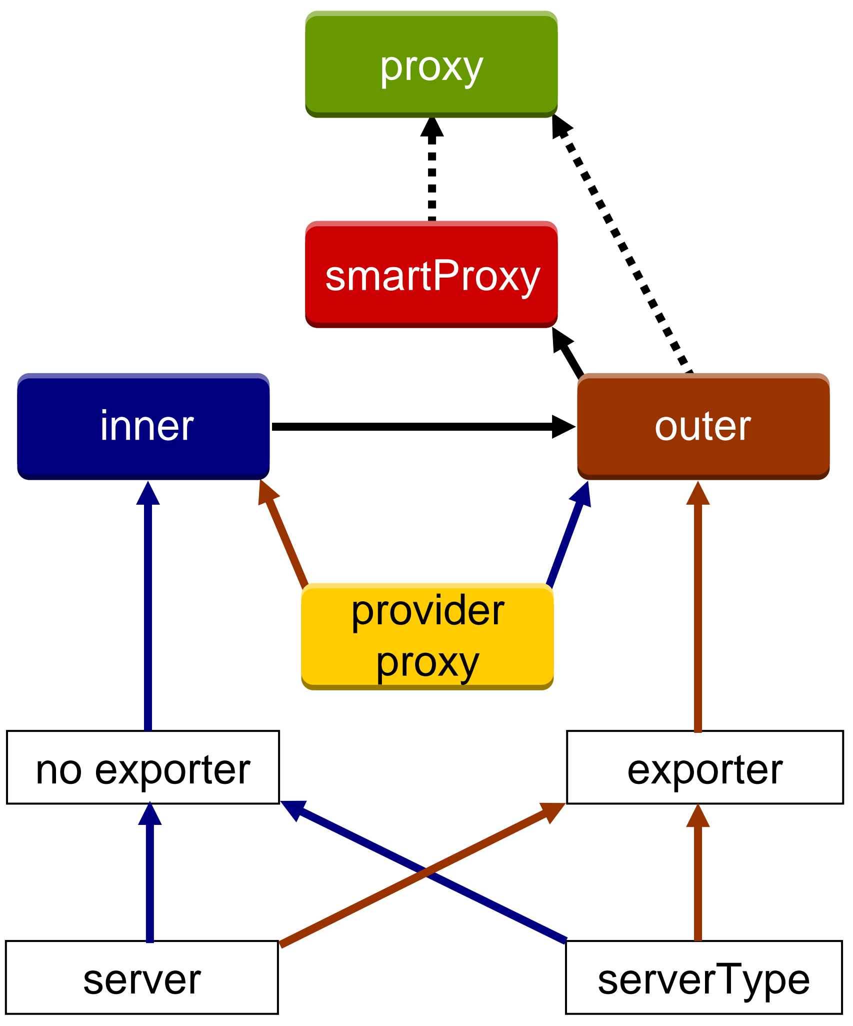
Native Provider
Technology wrapper-grids: Jini, JXTA, CORBA, Web Services, Grid Services, .NET
SORCER.grid
Technology native grids

The Runtime Environment

- SORCER (Eclipse) workspace
- Jini services (usually available on the network)
  - Lookup service (reggie)
  - JavaSpace (outrigger)
  - Transaction Manager (mahalo)
  - Event Mailbox (mercury)
  - Lease Renewal (norm)
  - Lookup Discovery (fiddler)
- Webster (HTTP class server) - iGrid/bin/webster/bin
- Service browser (Inca X) - iGrid/bin/incax/bin
- SORCER services - iGrid/bin/sorcer/bin
- Custom services - iGrid/bin/<serviceName>/bin

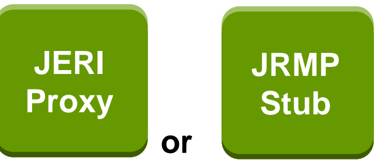
Proxying with Dependency Injection

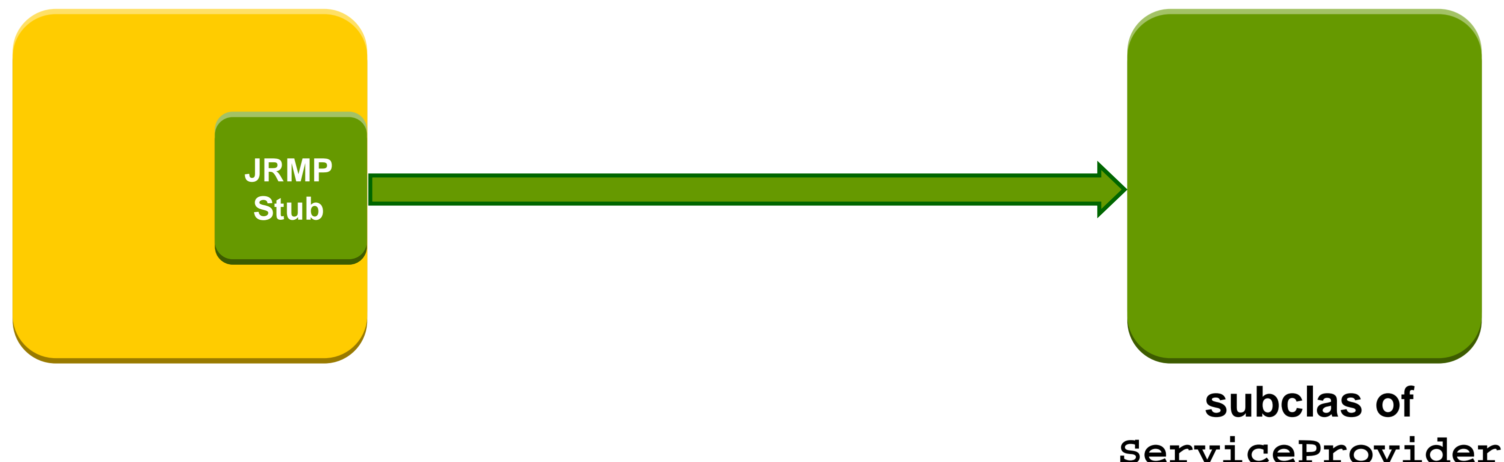
- smartProxy
- server
- serverType
- serverExporter
- beans



Twelve cases studied

Providers Implementing Remote Interfaces





subclas of `ServiceProvider`

Provider = Server

Direct calls can be forbidden with indirect service calls

`Servicer#service(Exertion):Exertion`

`ArithmeticImpl` Implements `ArithmeticRemote`

JERI Proxy or JRMP Stub

jeri and *jrmp*

Adder

Subtractor

Multiplier

Divider

Remote

Arithmetic

ArithmeticRemote

ArithmeticProviderImpl_Stub

ServiceProvider

uses

Arythmometer

ArithmeticProviderImpl

2

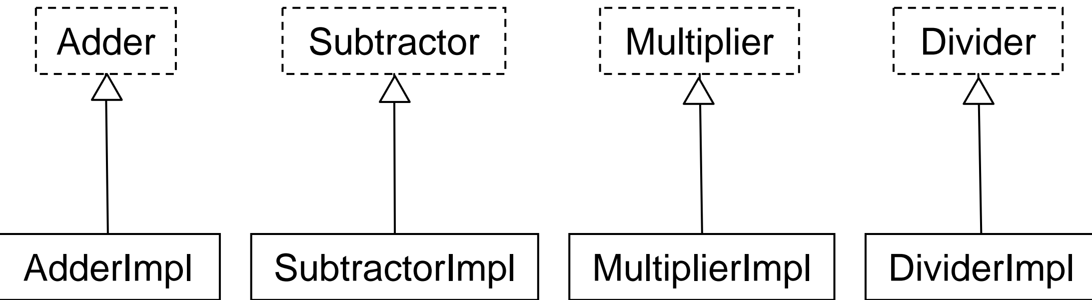
Providers with Service Beans

Provider = `ServiceProvider` + `beans`;
bean - implements service methods in its exposed (not `Remote`) interfaces.
Beans are not servers, they are not exported.

Service Beans

JERI Proxy

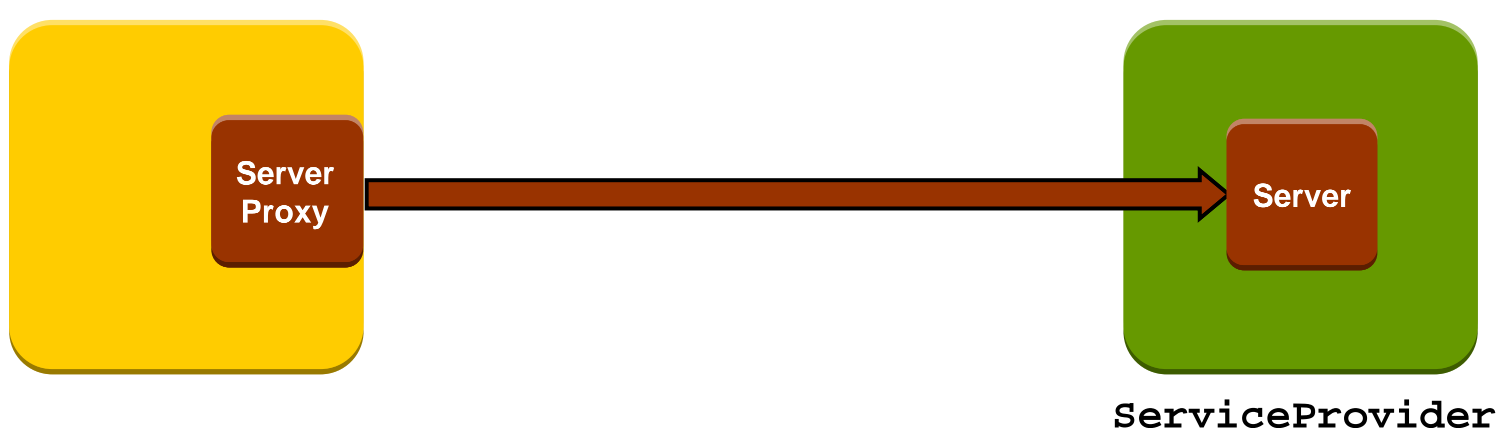
beans




```
beans = new String[] {
          "sorcer.arithmetic.AdderImpl",
          "sorcer.arithmetic.SubtractorImpl",
          "sorcer.arithmetic.MultiplierImpl",
          "sorcer.arithmetic.DividerImpl" };
```

3

Providers with Exported Servers



Provider = `ServiceProvider` + `server`;
`server` and `serverExporter` entries defined, and
`server` is not `Partner` type

RemoteArithmometer Implements ArithmeticRemote

Server Proxy

server

Adder
Subtractor
Multiplier
Divider
Arithmetic
Remote
Arythmometer
ArithmeticRemote
RemoteArythmometer
RemoteArythmometer_Stub

```
// RMI object
server = new RemoteArithmometer();
// exported with
serverExporter = new JrmpExporter(0);
```

ServiceProvider

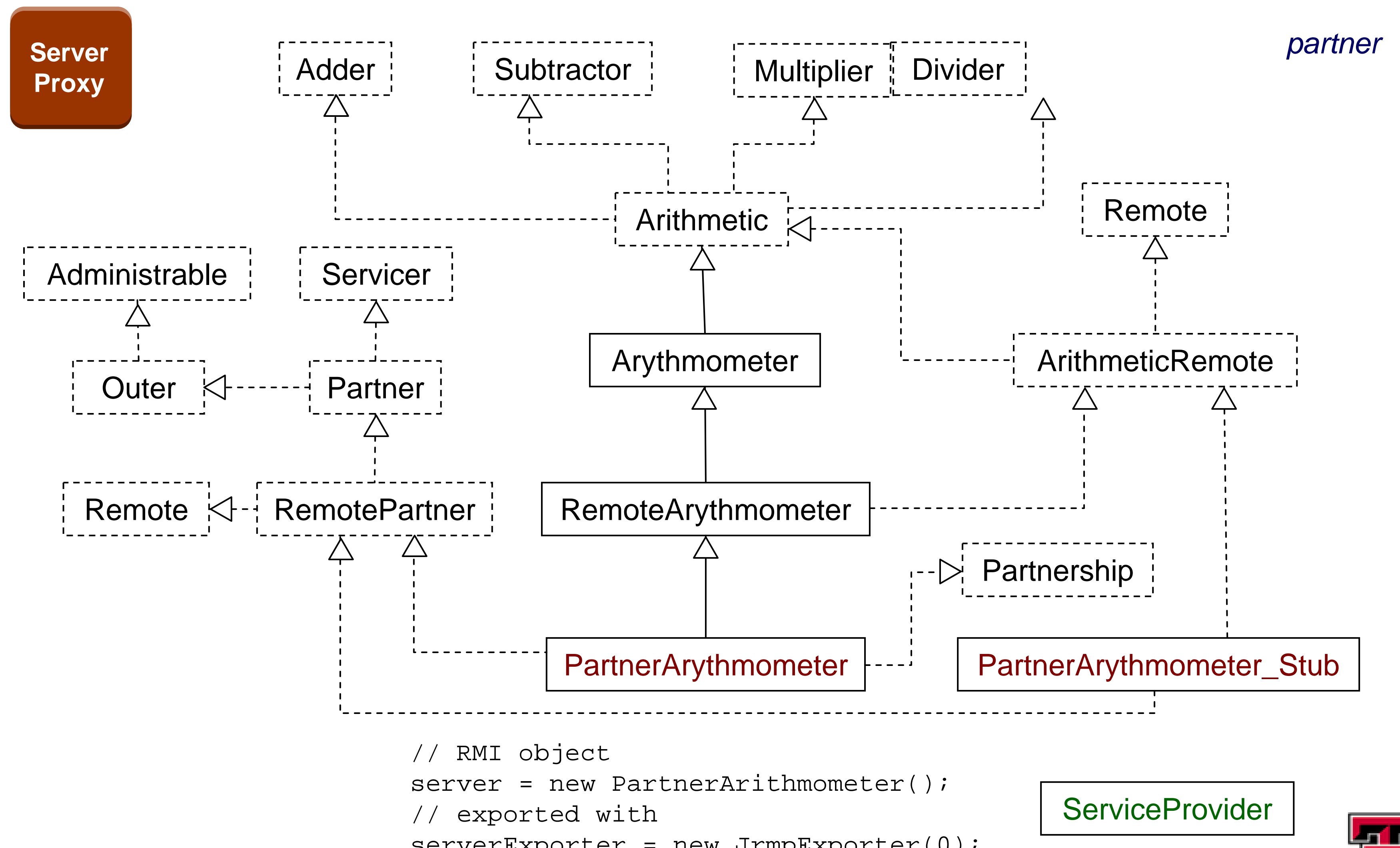
4

Providers with Exported Partnership Servers

Server Proxy

Provider = `ServiceProvider` + `server`; `server` and `serverExporter` entries defined, and server implements `Partner`

RemoteArithmometer Implements ArithmeticRemote and Partnership




```
// RMI object
server = new PartnerArithmometer();
// exported with
serverExporter = new JrmpExporter(0);
```

ServiceProvider

5

Providers with not Exported Servers

Prv Proxy

Server

Server Proxy

Registering Provider (Partner)

`ServiceProvider`

Provider = `ServiceProvider` + `server`;
`server` entry defined; no `serverExporter` entries defined, and `server` is not `Partner` type

RemoteArithmometer Implements ArithmeticRemote, not Exported

Prv Proxy


```
// RMI service type
serverType = "sorcer.arithmetic.RemoteArithmometer";
// exported with
//serverExporter = new JrmpExporter(0);
```

6

Smart Proxies – Fat Proxy

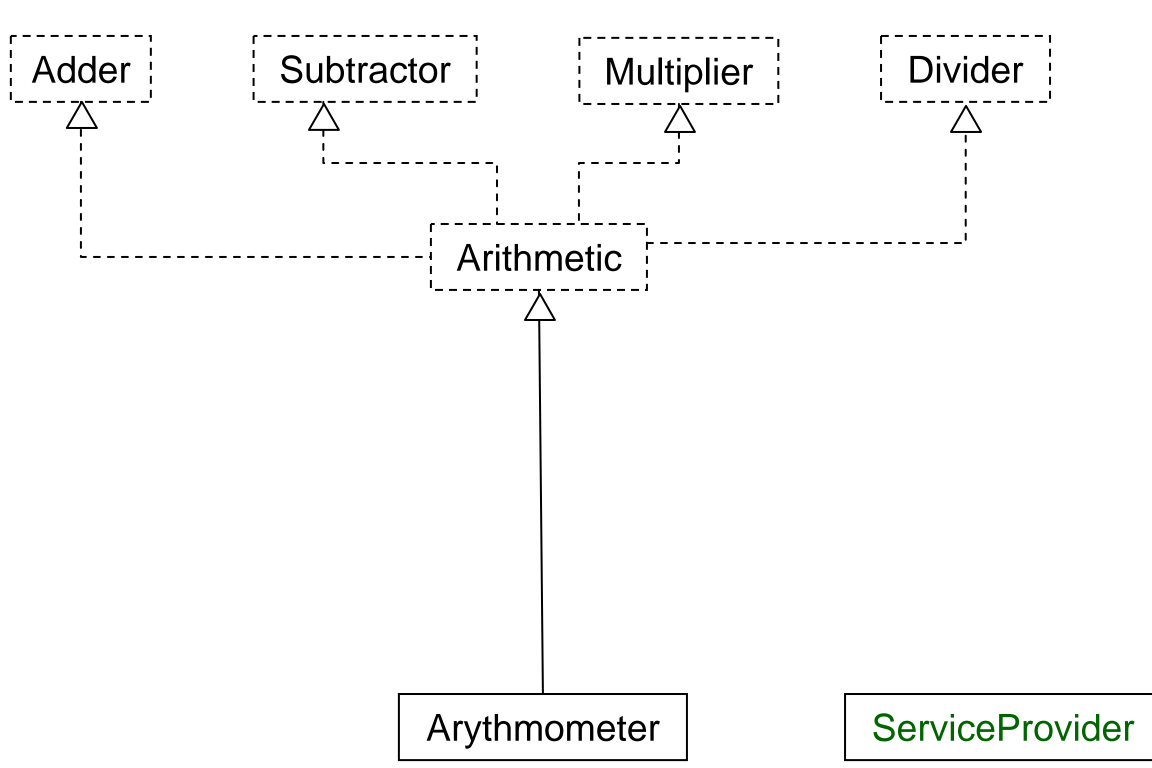
`ServiceProvider`

Provider = `ServiceProvider` **+** `smartProxy`**;**
requestor invokes local calls only;
`smartProxy` **is not** `Partnership type`**, and**
`ServiceProvider` **maintains the proxy registration.**

Arithmometer Implements Arithmetic (no Remote)

Smart Proxy

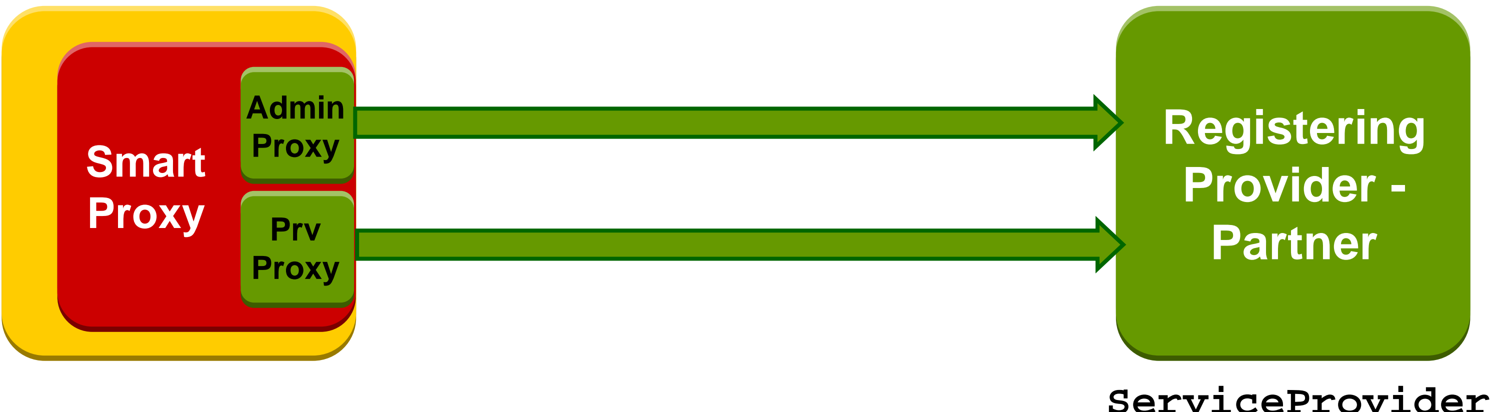
fat




```
smartProxy = new Arithmometer();
```

Outer Smart Proxies

Semismart Proxies



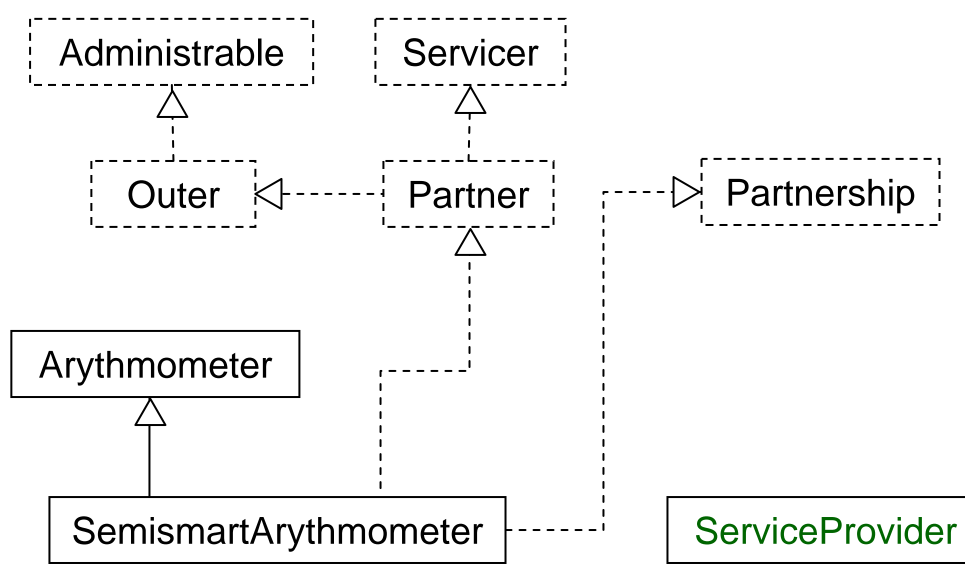
`ServiceProvider`

Provider = `ServiceProvider` + `smartProxy`
Requestor invokes local calls only;
`smartProxy` implements `Partnership`, and
`ServiceProvider` maintains the proxy registration.

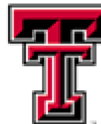
SemismartArithmometer Implements Outer

Smart Proxy

semismart



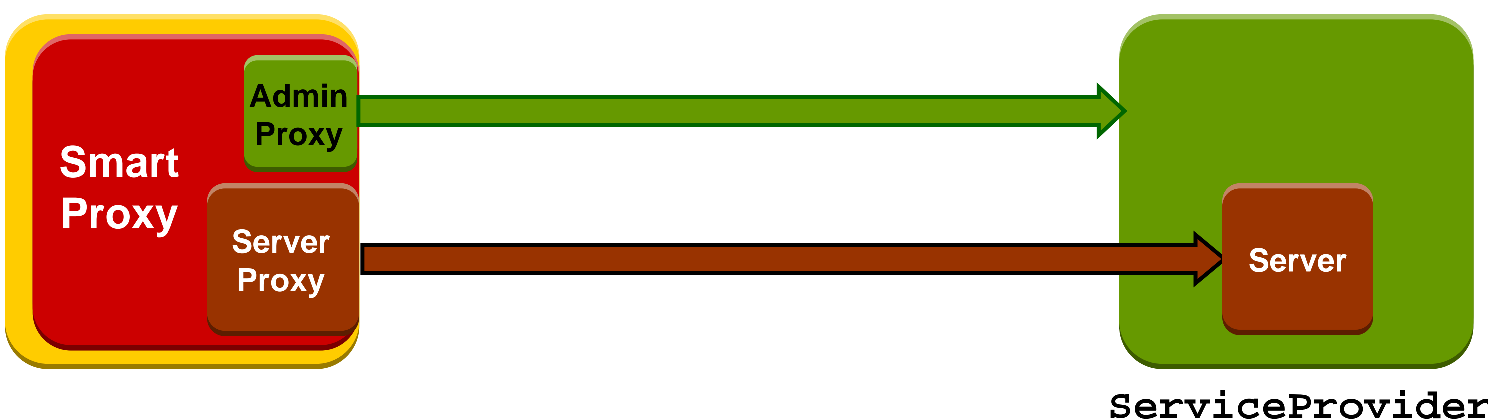

```
smartProxy = new SemismartArithmometer();
```



8

Smart Proxies with Exported Servers

Smart Proxy



ServiceProvider

Provider = ServiceProvider + smartProxy + server;
server and serverExporter entries defined, and server is not Partnership type;
smartProxy implements Partnership, and ServiceProvider maintains the smartProxy registration.

SmartArithmometer Implements Averager

Smart Proxy

smart

Arythmometer

Partner

Averager

Remote

AveragerImpl

SemismartArythmometer

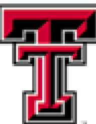
AveragerImpl_Stub

uses

SmartArythmometer

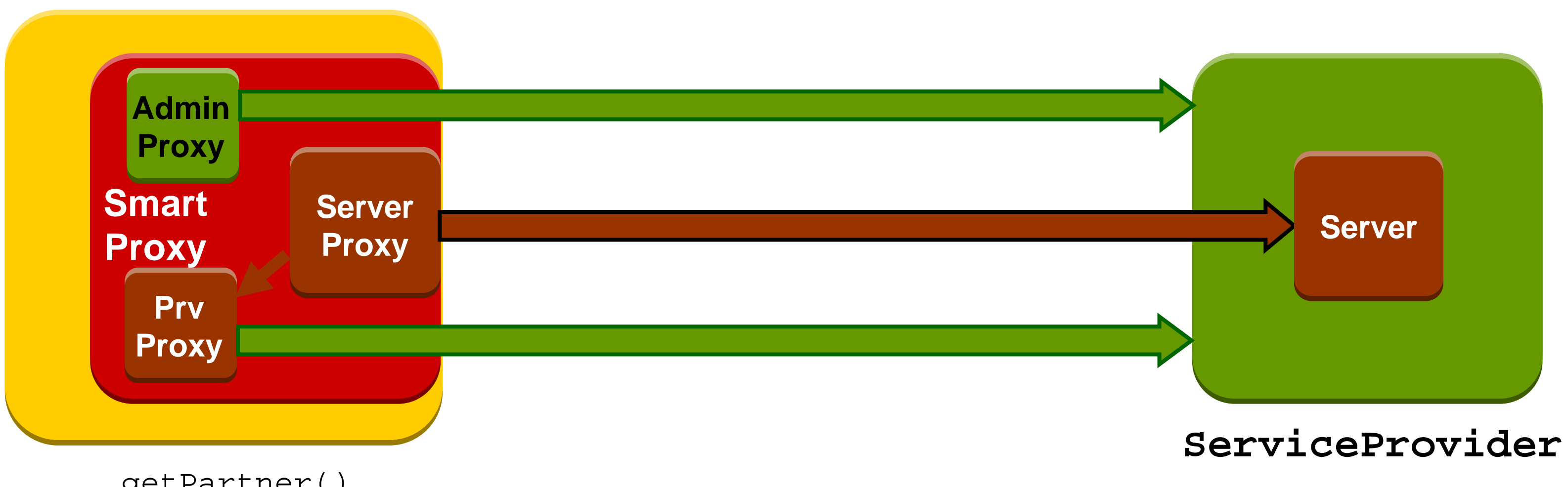
ServiceProvider

```
smartProxy = new SmartArithmometer();
server = new AveragerImpl();
serverExporter = new JrmpExporter(0);
```



Smart Proxies with Partnership Servers

Smart Proxy



Provider = `ServiceProvider` + `smartProxy` + `server;`
requestor invokes local calls only;
`server` and `serverExporter` entries defined, and
`server` implements `Partnership`
`smartProxy` implements `Outer`, and
`ServiceProvider` maintains the `smartProxy` registration.

SmartArithmometer Implements Averager, Averager Implements Partnership

Smart Proxy

smart-partner

Arythmometer

Outer

Averager

Remote

AveragerImpl

Partner

SemismartArythmometer

PartnerAveragerImpl

PartnerAveragerImpl_Stub

SmartArythmometer

uses

ServiceProvider

```
smartProxy = new SmartArithmometer();
server = new PartnerAveragerImpl();
serverExporter = new JrmpExporter(0);
```

10

Smart Proxies with not Exported Servers

Smart Proxy

Server

Smart Proxy

Prv Proxy

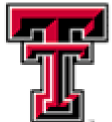
Server Proxy

Admin Proxy

Registering Provider

`ServiceProvider`

Provider = `ServiceProvider` + `smartProxy` + `server`;
`server` entry defined; no `serverExporter` entries defined, and `server` is `Partnership` type;
`smartProxy` implements `Outer`, and
`ServiceProvider` maintains the `smartProxy` registration.

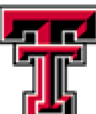


RemoteArithmometer Implements ArithmeticRemote, not Exported

Smart Proxy

Averager
Partner
AveragerImpl
PartnerAveragerImpl
Arythmometer
Outer
PartnerAveragerImpl_Stub
uses
SemismartArythmometer
ServiceProvider
uses

```
smartProxy = new SemismartArithmometer();
// RMI service type
serverType = "sorcer.arithmetic.RemoteArithmometer";
// exported with
//serverExporter = new JrmpExporter(0);
```



Proxying with Taskers

tasker

Tasker Proxy

Tasker Proxy

Tasker

Tasker Task **(`ArithmeticTask`)**
Tasker Method **(`ArithmeticMethod`)**

`ServiceProvider implements Tasker`

Tasker executes inserted *Tasker Method*

Proxying with Callers

caller

Caller Proxy

Caller Proxy

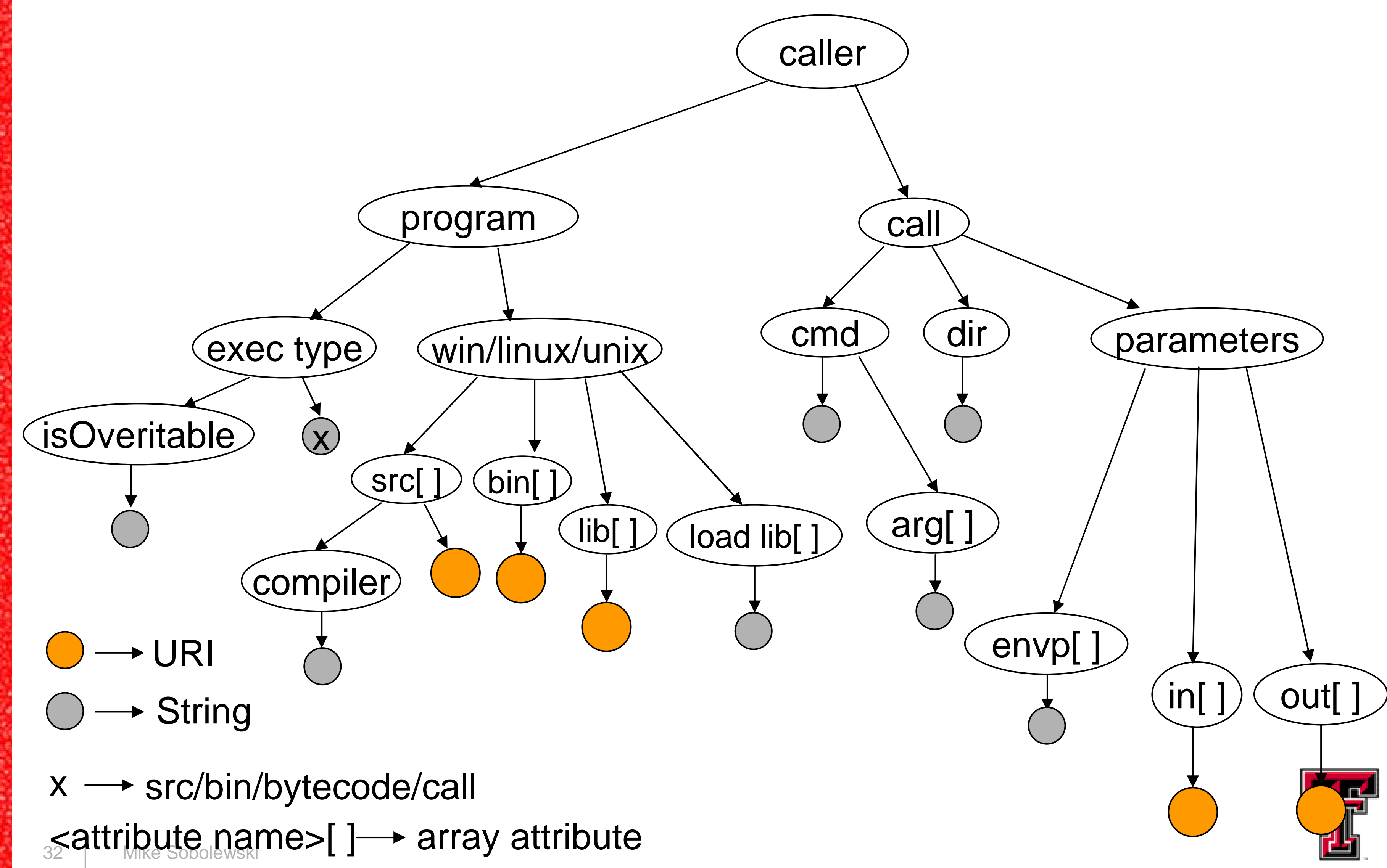
Caller

Caller Task with **`CallerContext`**

`ServiceProvider implements Caller`

Callers make a context-based system call

Caller Service Context



SORCER Research Domain

- Service-Oriented Programming
- Service-Oriented Computing Environment
- Service-Oriented Programming Development Tools
- Service-Federated Assurance and Security
- Self-Aware Service Federations
- Autonomic Service Federations
- Service Federated Intergrids
- *Metacomputing Service-Oriented MAO*
- SORCER Theses

Q&A

Grid interactive service-oriented programming environment

R.M. Kolonay
Air Force Research Laboratory, WPAFB OH

M. Sobolewski
Texas Tech University, Lubbock TX

ABSTRACT: Improvements in distributed computing, and the technologies that enable them, have led to significant improvements in middleware functionality and quality, mainly through networking and protocols. However, the distributed programming style has changed little over the years. Most programs are still written line per line of code in languages like C, C++, and Java. These conventional programs that can provide grid operations and grid data can be considered as common grid resources and shared by research and education communities worldwide. However, there are no relevant programming methodologies to utilize efficiently these shared service providers as a potentially vast grid repository, except through the manual writing of code. Realization of the potential of grid computing requires significant improvements in grid programming methodologies. The Grid Interactive Service-Oriented (GISO) methodology is presented that provides a programming environment with development tools that permit interactive (point-and-click), true grid programming, thus permitting the different elements of programming to be stored, reused, aggregated, and executed with a level of concurrency and grid-level control strategy not achievable in the conventional programming languages.

1 INTRODUCTION

From the beginning of networked computing, the desire has existed to develop protocols and methods that facilitate the ability of people and automatic processes across different computers to share information and knowledge across heterogeneous systems. As ARPANET (Postel and Sunshine 1981) began through the involvement of the NSF (Postel & Reynolds 1987, Lynch & Rose 1992) to evolve into the Internet for general use, the steady stream of ideas became a flood of techniques to submit, control, and schedule jobs across distributed systems (Lee 1992). The latest in these ideas is the *grid* (Foster 2002, Kesselman et al. 2002, Tuecke et al. 2002, Foster et al. 2001), to be used by a wide variety of different users in a non-hierarchical manner to provide access to powerful aggregates of resources (Foster & Kesselman 1999), Grimshaw, & Wulf 1997). Grids, in the ideal, are intended to be accessed for computation, data storage and distribution, and visualization and display, among other applications without consideration for the specific nature of the hardware and underlying operating systems on the resources on which these jobs are carried out (Smarr 1997, NRC 1993).

The reality at present, however, 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.

The need for further improvements in grid computing is clear, and requires significant further improvements in grid programming technology. By inspection of the above paradigm, it is clear that incremental improvements in the scripts and submission techniques will not suffice. A new *grid interactive service-oriented (GISO)* integrated development environment (IDE) that is based on evolution of the concepts and lessons learned in the FIPER project (Sobolewski 2002, Lapinski & Sobolewski 2002), Röhl et al. 2000), a $21.5 million program founded by the United States National Institute of Standards and Technology (NIST), is presented. It provides an environment that will permit true interactive click-and-drag grid programming through the manipulation of graphical elements that represent object-oriented grid resources, thus permitting the different elements of grid program to store, reuse, aggregate, and execute with a level of concur-

rency and grid-level control strategy not achievable in the conventional programming languages.

The presented GISO programming approach is characterized as follows:

1. Service-oriented grid programming is achieved by applying the object-oriented concepts directly to the grid as a repository of network objects (method and context providers)
2. Service-oriented execution infrastructure enabling dynamic federations of grid providers to execute service-oriented programs
3. Provisioning and deploying grid objects with an autonomic behavior, enabling grid objects to be instantiated and managed on compute resources available through the grid using an adaptive quality of service model
4. An open, web-based environment in which existing proprietary applications and analytical packages are integrated through Java-based wrappers that handle grid processes and data distributed across different locations.

2 GISO CONCEPTUAL FRAMEWORK

Building on the object-oriented paradigm the *service-oriented* paradigm, in which the objects are distributed, or more precisely they are *network objects* and play some predefined roles. A *service provider* is an object that accepts messages from *service requestors* to execute an item of work – a *task*. The task object is a service request – a kind of elementary grid instruction executed by a service provider. A service *jobber* is a specialized service provider that executes a *job* – a compound request in terms of tasks and other jobs. The job object is a *service-oriented program* that is dynamically bound to all relevant and currently available service providers on the grid. This collection of grid providers dynamically identified by a jobber is called a *job federation*. This federation is also called a *job 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 job space is the *execution environment* for the job itself and the job is a service-oriented program. This changes the game completely. In the former case the object space is a *virtual computer*, but in the latter case the job space is the *virtual network*. This virtual network or *grid federation* is the jobs' execution environment and the *job object* is a service-oriented program. In other words, we apply the object-oriented concepts directly to the grid in the service-oriented manner.

The GISO framework is built on the top of the FIPER Technology (Kolonay et al. 2002) middleware. The GISO environment provides the means to create interactive service-oriented programs and execute them without writing a line of source code. Jobs and tasks are created using web-based user interfaces. Also via web-based interfaces the user can execute and monitor the execution of jobs or tasks. The jobs and tasks are persisted for later reuse. This feature allows the user quickly to create new applications or programs on the fly in terms of existing tasks and jobs.

In all, GISO development tools provide (see Figure 1) accessibility through web-centric architecture; self-manageability using federated grids, scalability via network centricity, and adaptability with the power of mobile code inserted for execution through service providers.

Figure 1. GISO layered architecture

In this paper the focus is on the GISO programming and developed tools identified in Figure 1.

3 GISO PROGRAMMING AND DEVELOPMENT TOOLS

The peer-to-peer (P2P) service-oriented framework presented targets multiparty grid transactions. A collection of all registered service providers (active and inactive) is called a service grid. A nested transaction is composed of a federation of providers that come together for completing a transaction. A transaction consists of a set of tasks with specific precedence relationships. When performing a nested transaction, be it either a banking transaction or an engineering analysis, there are three basic components that can be identified. These are; the *process* or series of steps that must be executed to complete the transaction, a specification of the *action* to be taken in each step of the process, and the *information/data* associated with each step in the process (both input and output). Within FIPER the program objects that represent the components of a nested transaction are FiperExertions (FiperJob and FiperTask), FiperMethod, and FiperContext. The basic work unit within the FIPER programming environment is an exertion. Each exertion contains a FiperMethod and a FiperContext object. The Fiper-

Method specifies what *action* that is to be taken in a given step in the process. The FiperContext contains all the *data* the FiperMetod operates on or generates. The FiperContext also holds attributes for the data much like MIME types that identify the application(s) the data
is associated with, its format (text, binary etc.), and other user defined modifiers. A FiperJob defines the *process*. It consists of one or more exertions, the execution strategy for the process (sequential, parallel, looping and conditionals), and the mapping/relationship of data between exertions. The hierarchy of these classes is shown in Figure 2. It is worth nothing that recursion of FiperJobs is supported. That is any of the FiperTasks within a Fiper Job can be a FiperJob itself.

The relationship between the FIPER program objects and the general description of a nested transaction is as follows; a FiperJob represents the process, the *FiperMethod represents the action*, and a FiperContext represents the data/information. The FiperTask acts as a container holding the FiperMethod and FiperContext creating the basic unit of work that is passed between various service providers.

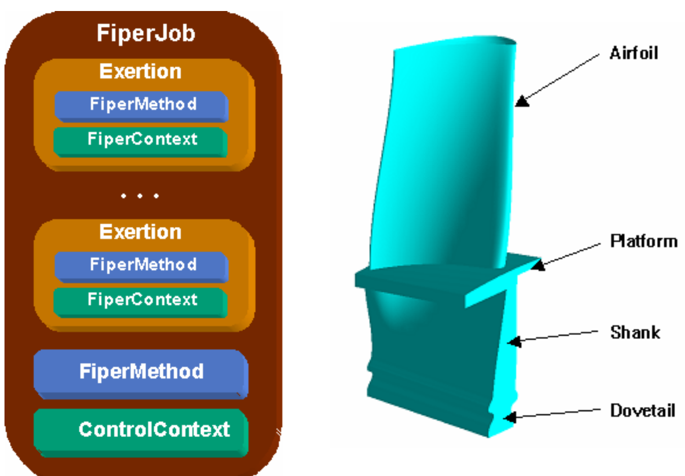
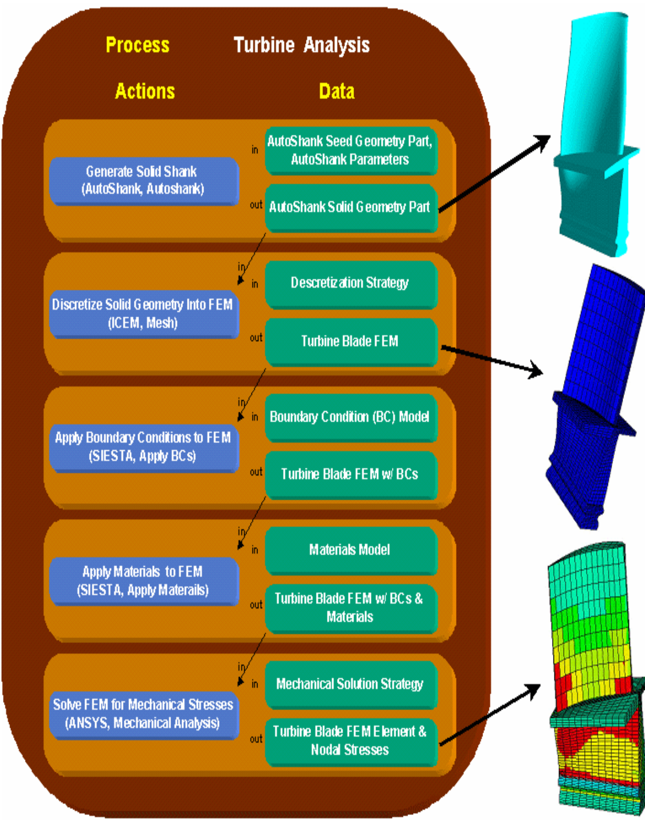


Figure 2. Program Object Hierarchy

Figure 3. Turbine Blade Geometry

As an example of a nested transaction in the FIPER Environment consider the following engineering application, the mechanical analysis of a gas turbine component. The component, a turbine blade is shown in Figure 3. The *process* of performing a mechanical analysis consists of the following *actions*; generate solid geometry, descretize the geometry into a finite element model (FEM), apply boundary conditions to FEM, apply materials to FEM, and solve the FEM for structural stresses. The necessary input data for each action and the resulting output data are shown in Figure 4. Also depicted in Figure 4 is the association between the three components of a nested transaction and the FIPER program objects.

To create the necessary program objects (FiperContext, FiperMethod, FiperTask, and FiperJob) for a nested transaction in the FIPER environment a collection of web browser user agents has been developed. It is not necessary to use these user agents for the development and execution of a FiperJob. Any standalone application can perform programmatically the same steps to create the necessary ob-



jects and act as a service requestor to submit the

Figure 4. Process for the Mechanical Analysis of a Turbine Blade

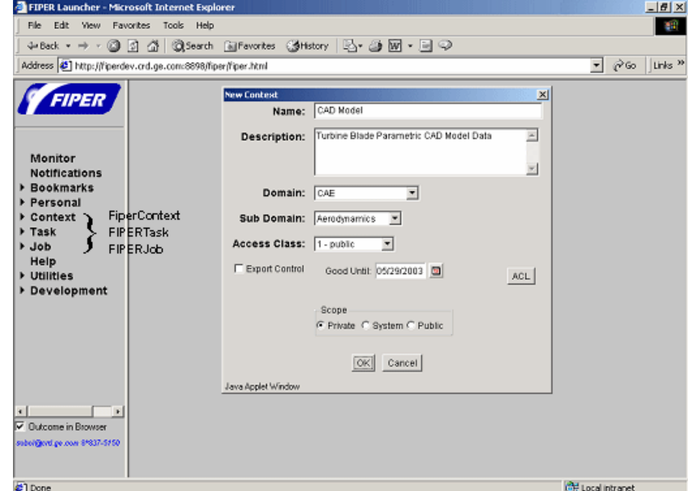


Figure 5. FIPER Launcher and New Context Dialogue

FiperJob for execution. The following sections illustrate the usage of the web user agents to create and execute the necessary FIPER program objects to perform the mechanical analysis of the turbine blade. Figure 5 shows the Fiper launcher page once logged into the Fiper environment. Here it can be

seen that there are separate selections for the above described program objects, FiperContext, FiperTask, and FiperJob. The FiperMethod object is created within the FiperTask menu selection.

3.1 *Context Editor*

The Context Editor allows the end-user to specify the data or references to the data along with attributes associated with the data. When creating a new context the end-user is presented with the dialog that requires the following fields. The Name and Description fields are user defined, the Domain and Subdomain are selected from a drop down menu. The Access field is a company internal access classification and the Export Control box indicates if the data is export controlled. The ACL button produces an Access Control List (ACL) dialogue that allows the end-user to assign read, write, and execute permissions on this program object based on userid or role. Once the end-user completes the New Context Dialogue and selects OK the Context Editor then appears. Figure 6 shows the Context Editor along with the context for the first action or task in the Turbine Mechanical Analysis Job represented in

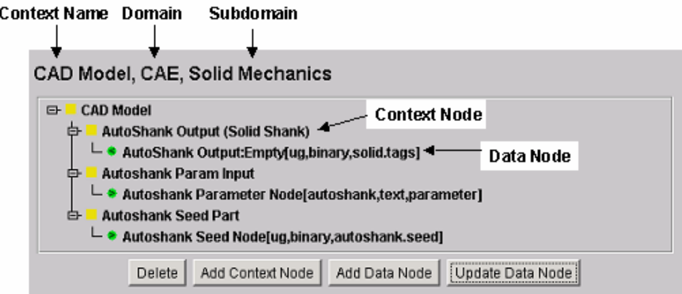


Figure 4.
Figure 6 also illustrates that the FiperContext is a tree structure with Context Nodes and Data Nodes. The Data Nodes are further identified as either input ">" or output "<". The editor allows the end-user the ability to create, edit, or delete Context Nodes and Data Nodes in the FiperContext.

3.2 *FiperTask Editor*

From the Fiper launcher in Figure 5 the end-user se-
Figure 6. FiperContext Editor

lects Task, New, and completes the New Task Dialog to gain access to the Task Editor shown .

Recalling that the FiperTask is the fundamental building block or work unit in the FIPER Environment which contains the *action* and *data* for a nested transaction (reference Figure 4), the Methods field represents the *action* and the Context field represents the *data*. To view/edit more detail on theses fields the end user selects “Update Content” which produces an editor (see Figure 7). Figure 7 shows the definition of the FiperMethod and the Context that is used for the selected task, Generate Solid Shank. The fields Interface, Command, Provider, and Method Type define the Method. The Interface and the Provider are used as the attributes to locate a service within the environment with the current implementation. The context for this task is the CAD Model Context presented in Figure 6. Once all the actions/FiperTasks have been defined for a given process/FiperJob the FiperJob itself can then be constructed.

3.3 *FiperJob Editor*

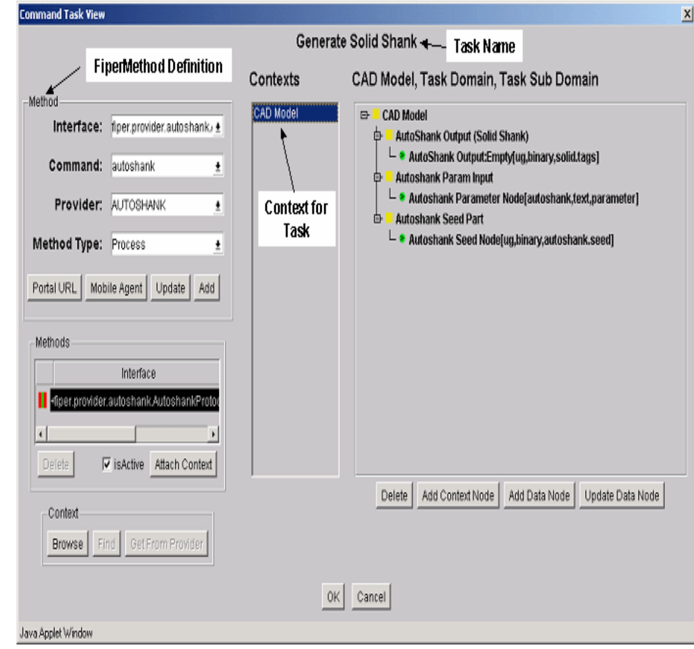
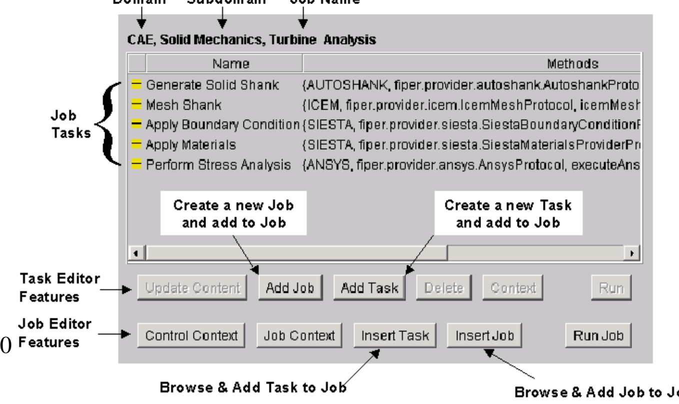


Figure 7. FiperTask, FiperMethod and FiperContext Editor

Figure 8 illustrates the creation of the FiperJob represented in Figure 4. It contains all the tasks, Generate Solid Shank, Mesh Shank, Apply Boundary Conditions, Apply Materials, and Perform Stress Analysis.
Figure 8. FiperJob Editor

The Job Editor lists all FiperTasks associated with the job along with the FiperTask’s Name and FiperMethod Attribute information (Provider Name and requested provider’s type - interface). The Task and Job Editor features allow the end user to add additional FiperTasks or FiperJobs by either browsing existing program objects or creating new



objects on the fly. The Job Editor features also enable the specification of the Control Context and the JobContext. The ControlContext specifies the flow and method of execution of the FiperJob. The final step before a FiperJob can be executed is to define the flow of data between tasks in the job. This is done using the JobContext dialog, which can be invoked from the Job Editor features on the Job Editor Dialog in Figure 8.

The FiperJob Context dialog for the Turbine Analysis Job is shown in Figure 9. Here the Job is shown with each task and the context for each task in a hierarchical tree structure. The data flow from one task to the other is defined by dragging one Fiper DataNode onto another Fiper DataNode. In Figure 16 this has occurred by dragging the AutoShank Output Solid Shank Node contained in Task0 onto the Solid Shank unnamed Fiper DataNode in Task1.

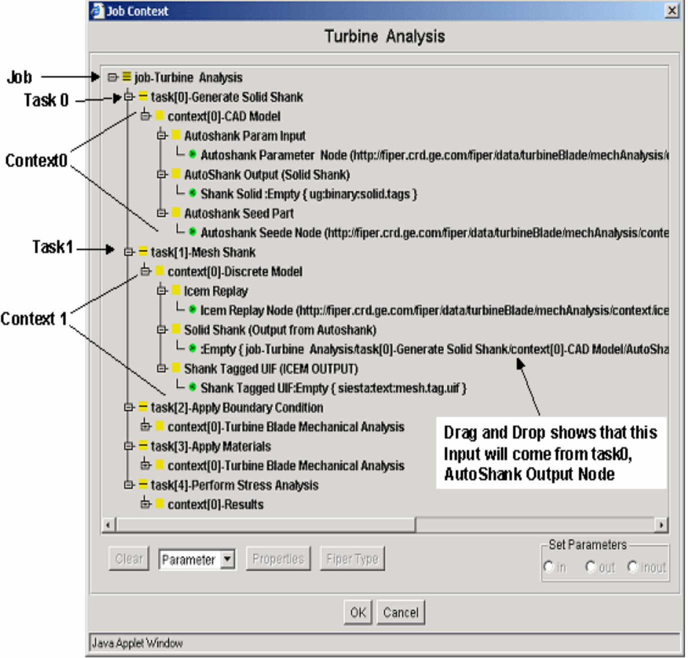


Figure 9. Fiper JobContext Dialog

Once the data flow has been defined in the JobContext the FiperJob is now ready for execution. To submit the job to the Fiper Environment the Run Job button is selected in the Job Editor (Figure 8). A typical engineering analysis or design job could take anywhere from a few hours up to several days or even weeks. With jobs running this long it is critical that the end-user have access to the status of the job and control over the job as it executes. This is the function of the Job Monitor.

3.4 *FiperJob Monitor*

The most critical capability that GISO programming will need from an end-users perspective is the ability to interact with the process/FiperJob once it has been submitted to the environment. Using a GISO IDE will require a cultural change within the end-user community. Today's state of practice is that typical designers and analysts execute single standalone applications either on their desktop or submit the runs to a major shared resource (MSR) computing environment. In either case the end-user is executing applications individually and if a failure occurs they know at least which application the failure occurred within. Also, when running locally or in a MSR the end-user usually has some or all control over the running application and can closely monitor the progress of the execution by monitoring log files and or output files from the application. In the GISO IDE the end-user is now combining many application to perform a nested transaction and submitting the execution of the nested transaction to the network, which could easily take days or weeks to complete. In the GISO IDE the end-user may have no idea where the execution is taking place and worse will have no feedback as to the state of progress of the process. In the GISO IDE the end-user surrenders all control to the environment, a precarious proposition for a designer who is accustomed to having complete control of the applications they are running. With these facts in mind a few essential functionalities are identified for GISO programming that are necessary for end-user to accept such a working environment. The end-user must be able to monitor the progress of the process and obtain intermediate results from a given task. The end-user must be able to control the process once it is submitted to the environment by stopping, suspending, or terminating the process. For a suspended GISO program the end-user must be able to edit not only the data within the process but also the process itself by adding or deleting tasks. After any edits to the data or process the end-user must be able to resume the process from any task within the process not necessarily the task the process was suspended at. If the process fails the end-user must obtain meaningful information that specifies where the failure occurred and what action needs to be taken to correct the problem. This last requirement puts a significant burden on the service provider developers to properly trap exceptions and translate them into meaningful information for the end-user.

In the FIPER Environment the monitoring/client process interaction is done using the Job Monitor. Figure 10 shows the Turbine Analysis Job running in the Job Monitor. The Job Monitor can be viewed as an "interactive debugger for program objects or services on the network". The Job Monitor shows the progress of the process (green complete, green/yellow running, red failed, yellow suspended). It also displays intermediate information from a task (by viewing the job context) if the provider returns such information. The client is also able to stop,

suspend, step and resume a running job. In addition, for a given suspended or completed job, the client has access to a drop down menu that allows full edit capability of the data in the job or the job/process itself. Data can be changed, tasks can be edited/added/deleted and the job resumed from any task.

Figure 10. FiperJob Monitor

4 CONCLUDING REMARKS

In the GISO approach object-oriented concepts are applied to the network and grid-oriented programs. A job is a service-oriented program executed in a federated service-oriented environment across multiple virtual organizations. Jobs are created using friendly, interactive web-based graphical interfaces. Jini connection technology from Sun Microsystems enables federated, platform independent, real world grids. It allows us to create GISO programs that process a whole aircraft engine as a virtual object-oriented product control structure that can be manipulated by multidisciplinary teams as network-centric, active, evolving product. New shared programs and engineering applications can be assembled as needed on the fly by integrating new capabilities into existing workflows, systems, devices and applications. The presented web-centric GISO IDE reduces the costs of solving business problems as well as of establishing and maintaining online business relationships. Services are provided by shared low cost, easy to develop service providers and are integrated into the core business of an enterprise. An experimental version of presented approach was successfully deployed at GE Aircraft Engines.

REFERENCES

Foster, I. & Kesselman, C. eds.1999. "The Grid: Blueprint for a New Computing Infrastructure," Morgan Kaufmann Publishers, San Francisco CA.

Foster, I., C. Kesselman, C., Tuecke, S. 2001. The Anatomy of the Grid: Enabling Scalable Virtual Organizations.. International J. Supercomputer Applications, 15(3), 2001. Defines Grid computing and the associated research field, proposes a Grid architecture, and discusses the relationships between Grid technologies and other contemporary technologies.

Foster, I. & Kesseman, C. 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. (extended version of Grid Services for Distributed System Integration).

Grimshaw, A. S. & Wulf W. A. 1997. "The Legion vision of a worldwide virtual computer", Communications of the ACM, 40(1), 39-45.

Hafner, K. & Lyon, M. 1996. "Where Wizards Stay Up Late," (a history of Internet development), Simon and Schuster, New York.

Kolonay, R.M., Sobolewski M., Tappeta, R., Paradis, M., Burton, S. 2002. *Network-Centric MAO Environment*, The Society for Modeling and Simulation International, 2002 Westrn Multiconference, San Antonio, Texas, Jan 27-31.

Lapinski M. & Sobolewski, M. 2002. "Managing Notifications in a Federated S2S Environment," International Journal of Concurrent Engineering: Research & Applications, December.

Lee, J. ed. 1992 "Time-Sharing and Interactive Computing at MIT," IEEE Annals of the History of Computing 14:1

Lynch, D.L. & Rose, M.T. 1992. "Internet System handbook," Addison-Wesley, reading, MA.

National Research Council 1993. Reports relevant to early grid research include the following: "National Collaboratories: Applying Information Technology for Scientific Research," National Academy Press, Washington D.C.

Postel, J., Sunshine, C. & Cohen, D. 1981. "The ARPA Internet Protocol," Computer Networks 5:261-271.

Postel, J. & Reynolds, J. 1987. "Request for Comments Reference Guide (RFC1000)," Internet Engineering Task Force.

Smarr L.1997. "Computational infrastructure: Toward the 21st century," Special issue on plans for a National Technology Grid, Communications of the ACM 40, 11

Sobolewski, M. 2002. FIPER: The Federated S2S Environment, JavaOne, Sun's 2002 Worldwide Java Developer Conference, San Francisco, 2002.

Röhl P. J., Kolonay, R.M., Irani, R.K., Sobolewski, M., Kao, K. 200. "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.

Tuecke S., Czajkowski, Foster, I., Frey, J., Graham, S., Kesselman, C. 2002. Grid Service Specification. Open Grid Service Infrastructure WG, Global Grid Forum, Draft 2.

A FEDERATED INTELLIGENT PRODUCT ENVIRONMENT

Peter J. Röhl[*], Raymond M. Kolonay[†], Rohinton K. Irani, Michael Sobolewski, Kevin Kao
General Electric Corporate Research and Development
Schenectady, NY 12301

Michael W. Bailey[†]
GE Aircraft Engines
Cincinnati, OH 45215

Abstract

The concept of a federation of distributed devices on a network which enter the federation through a process of "discover" and "join", by which they register with a service request broker and publish the services which they perform is applied to engineering software tools. A highly flexible computer architecture is developed, leveraging emerging web technologies like Sun Microsystems' Jini™, RMI, JavaSpaces, in which engineering software tools like CAD, CAE, PDM, optimization, cost modeling, etc. act as distributed service providers and service requestors. The individual services communicate via so-called context models, which are abstractions of the master model data of a particular product. A human user interacts with the framework through a thin client like a web browser from anywhere in the world, where proper security measures to prevent unauthorized access to proprietary data is maintained. The paradigm of the CAD Master Model is extended with the introduction of the Intelligent Master Model (IMM), which, in addition to the *what*, captures the *why* and *how* of a design through the use of knowledge-based engineering tools. An initial example, the mechanical analysis of a turbine engine blade, is implemented.

Introduction

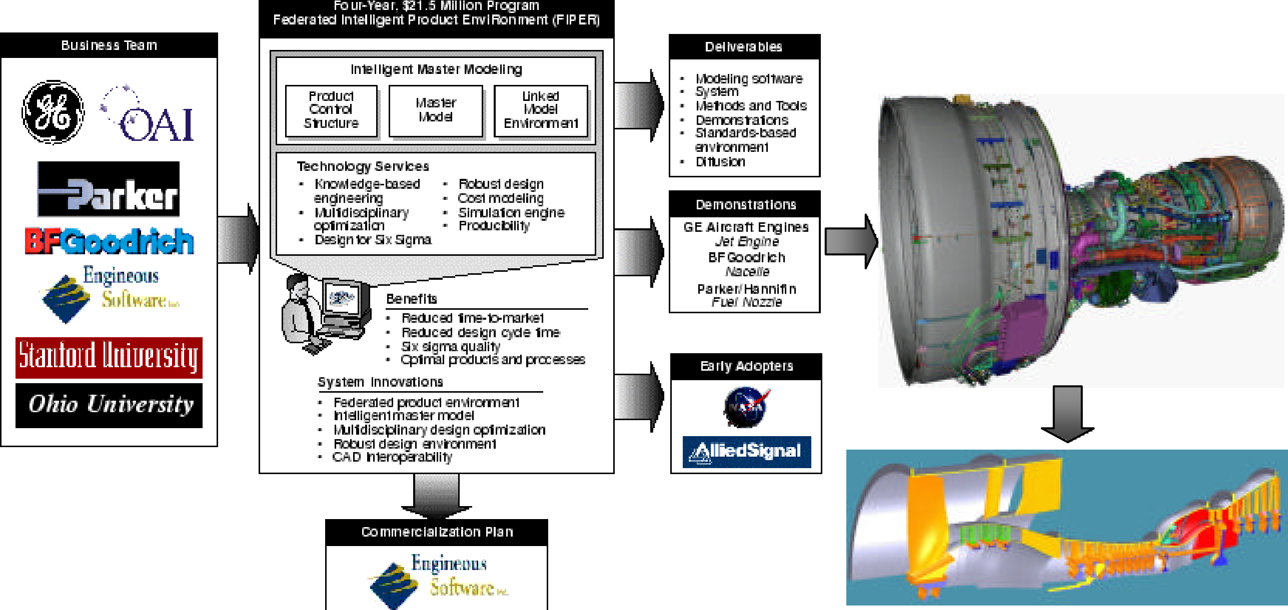
Turbine engine development is a highly coupled multidisciplinary process. In a market with ever increasing demands in terms of life cycle cost, environmental aspects (noise, emissions, and fuel consumption), and performance, the availability of accurate analytical tools during the design process is a given and ceases to be a discriminator between the various competitors[1,2]. It is, therefore, the application of these tools and their automated interaction in a robust computational environment, which may decide over success or failure of a specific project through reduction of design cycle time and avoidance of costly rework because of availability of high-fidelity information earlier in the design process. At the same time, especially in a multi-national company, design increasingly takes place at spatially distributed locations, potentially all over the world, where all participants in the design process need constant real-time access to all relevant up-to-date product information. In light of these challenges, GE has teamed with Engineous Software, BFGoodrich, Parker Hannifin, Ohio Aerospace Institute, and Ohio and Stanford Universites in a four-year effort to develop a "Federated Intelligent Product EnviRonment" (FIPER) under the sponsorship of the National Institute for Standards and Technology-Advanced Technology Program (NIST-ATP™), see Figure 1. FIPER strives to "drastically reduce design cycle time, and time-to-market by intelligently automating elements of the design process in a linked, associative environment, thereby providing true concurrency between design and manufacturing. This will enable distributed design of robust and optimized products within an advanced integrated web-based environment"[3].

The Intelligent Master Model

FIPER draws extensively on GE Aircraft Engines' Common Geometry Strategy, the Linked Model Environment (LME) and top-down Product Control Structure (PCS)[4] (Figure 2) using Unigraphics[5] (UG) WAVE functionality, but tries to extend these efforts with the capture of designer's knowledge in Knowledge Based Engineering (KBE) systems to create the "Intelligent Master Model" (IMM). In the following paragraphs, we will give a brief overview over the

* Member AIAA
† Senior Member, AIAA

Copyright © 2000 General Electric Company.
Published by American Institute of Aeronautics and Astronautics, Inc., with permission.



Common Geometry Strategy and the terms introduced above.

Figure 1: The FIPER Project

Figure 2: Product Control Structure

GEAE Common Geometry Strategy

The GEAE Common Geometry initiative started four years ago as a logical extension to Productivity Tools which had been under development since the early

nineties. It was realized that merely automating what was essentially a serial process had limitations and a fundamental paradigm shift was required. Bottom-up design may be optimal from a part perspective but does not necessarily lead to optimal system design. As initially conceived, the GEAE Common Geometry strategy objective was to make a single geometric representation common to all product creation activities from product concept through preliminary and detail design to manufacturing and services. However, to fully exploit the concept, knowledge has to be fused with feature-based parametric CAD (Figure 3), an environment linking CAD to engineering analysis, the LME (Figure 4) and a PCS to render it an Intelligent Master Model. This permits a top-down approach to design which permits system level requirements to flow down to drive the design. The IMM is a major enhancement to the master model concept, elevating the functionality of today's CAD systems to a new level.

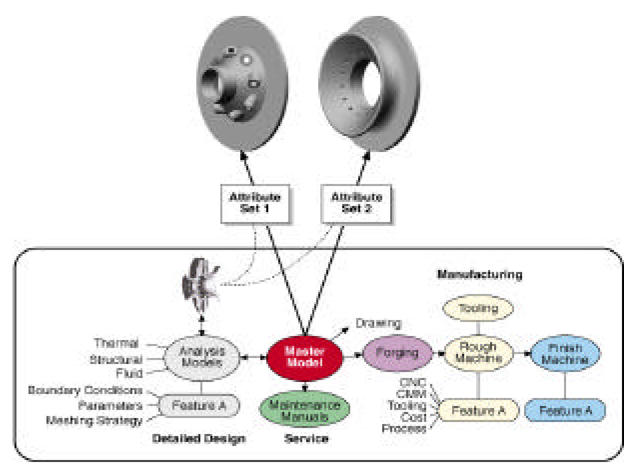


Figure 3: The Master Model Supports Feature-Based Modeling for Design and Manufacturing

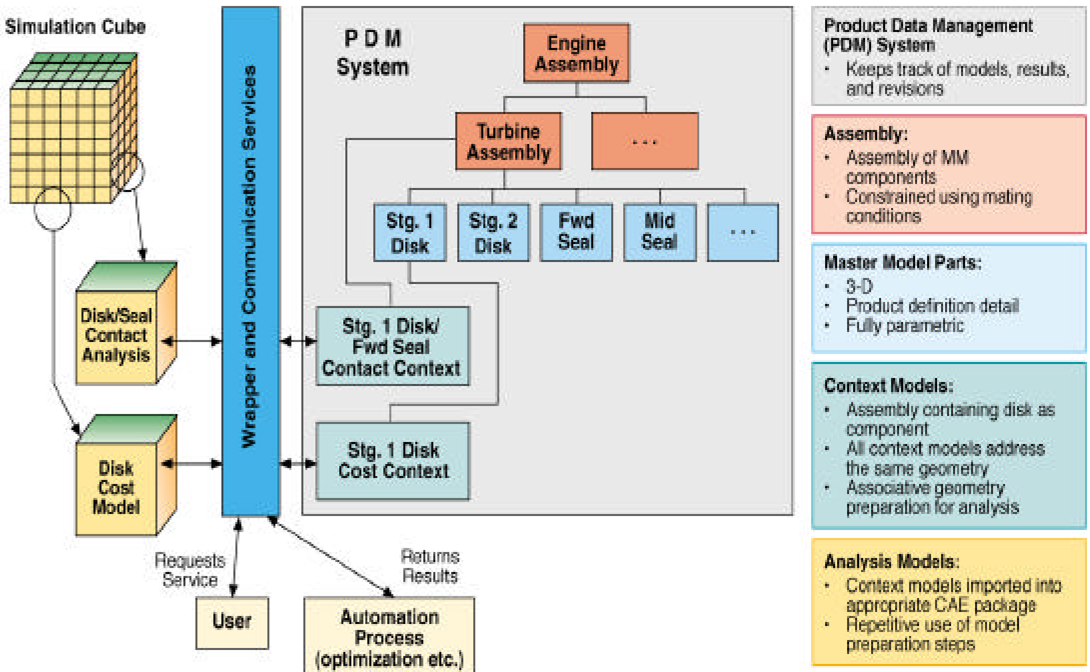


Figure 4: Linked Model Environment

The PCS allows top-down control of the design. It enables the lead engineer to lay out the overall system configuration and control changes in a top-down fashion. It facilitates what-if analysis at the conceptual, preliminary, and detailed design levels by allowing the designer to make parametric changes in the overall system layout and space allocation to evaluate one configuration versus another. Common Geometry refers to the notion that all disciplines involved in the design and manufacturing process have access to and use the same (evolving) geometric representation of the product. Realizing that different disciplinary engineering design and analysis tools require geometry at different levels of detail, the concept of a "context model" was introduced. The context model represents a disciplinary context-specific, yet fully associative, "view" of the master model geometry. Feature suppression is extensively used in context models. For example, a bolt

hole, which is important for the stress analyst, may not be required for a thermal analysis and therefore be suppressed in the thermal context model. Another context or “view” of the bolt hole are the manufacturing processes and cost to produce it. These context models are then linked to the respective disciplinary analysis tools, e.g. FEA, CFD, cost, producibility, etc, in the LME , see Figure 4.

<u>System Level Layout with Integrated Design (SOLID)</u>

The pilot projects to evaluate IMM functionality were described in Reference 4. To demonstrate the top-down design approach, a compressor was built using the feature based parametric CAD and WAVE functionality in Unigraphics. The following year this was extended to a full core engine comprising a compressor, combustor and high pressure turbine. The resulting model was called SOLID (System Level Layout with Integrated Design). Several lessons were learned during the pilot which were incorporated not only in the SOLID core but in Unigraphics enhanced functionality. The SOLID core is shown in Figure 5.

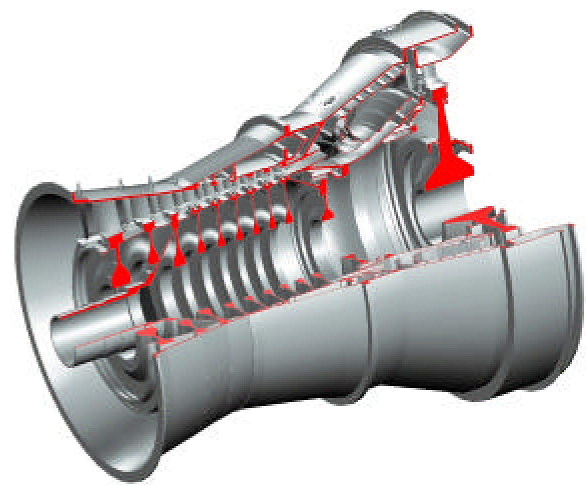


Figure 5: Complete Engine Core Model

<u>Knowledge Based Engineering</u>

KBE is a technology that allows an engineer to create a product model based on rules that capture the methodology used to design, configure, and assemble products. KBE facilitates the capture of the intent behind the product design by representing the *why* and *how,* in addition to the *what* of a design, see Figure 6. The knowledge captured could include everything from high-level, non-geometric engineering rules, manufacturing constraints, dependencies and relationships to parametric geometry definition. The geometric description is only one view of the information associated with the total product model. Links can also be established to standard parts catalogs, material databases, analysis tools, empirical knowledge, and design handbooks. Effectively, one can house, and ultimately archive, corporate design practices as well as design and manufacturing engineers’ expertise which can then be used by non-experts in a consistent manner to produce correct-first-time designs. Once the product model has been created, it can be used to rapidly create a new instance of the design when the product specifications change. In addition, various outputs, including analysis context models, etc. would be automatically created.

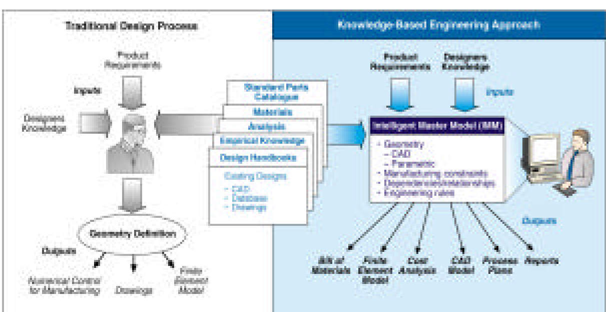


Figure 6: Knowledge-Based Engineering Extends the Master Model Concept

In the FIPER environment KBE is being used to intelligently modify the PCS, drive changes to parameters that define cross-sections and features and thereby intelligently scale a complete aircraft engine or components of the engine. To accomplish this, the approach being used within the Unigraphics system is to imbed the KBE language Intent[TM,6] to drive generative and variational design. While variational design creates a new design by intelligently scaling an existing design, generative design creates a new design based on a set of rules without the use of existing geometry. Rules management is also being addressed by incorporating them in a Product Data Management (PDM) system so that they are well documented, categorized and easily searchable.

Another use of KBE that is being pursued is for the formulation and execution of the MultiDisciplinary Optimization (MDO) problem. Here knowledge will be used to guide the decomposition of the overall optimization problem into smaller, more manageable, sub-problems, and to integrate the solutions of the sub-problems into an overall system level design.

The initial approach to KBE was the encapsulation of rules about the product in the form of the XESS spreadsheet functionality contained within Unigraphics. These spreadsheets are linked to the geometry such that design rules and practices are parameterized to

drive geometry. In addition external analysis codes such as those used for engine disk design can be executed. Thus an increase in airflow through the compressor would initiate an aerodynamic resizing of blades and vanes, resulting in a blade and platform resizing combined with disk redesign. Upon initiation of the UG/WAVE update, the whole compressor would rubber band to accommodate the increased airflow.

It was realized there that were limitations to the utilization of spreadsheets to capture the knowledge required to accomplish intelligent scaling of the engine core. GEAE evaluated several KBE packages to find the desired functionality. In the meantime Unigraphics Solutions entered into an agreement with Heide Corporation to integrate their Intent™ software into Unigraphics as "UG Knowledge Fusion". The reason for this is that for complex products the number of rules gets large very quickly and consequently difficult to manage. There are two types of KBE rules, Generative and Checking. Generative rules would for example change the number of stages in the compressor from 9 to 7 stages whereas the Checking rules would check that the disk bore stress and burst margin conform to design practices and run the appropriate codes to validate this requirement.

FIPER Architecture

Fundamental to the FIPER project is its web-based distributed software architecture. FIPER federates processes, tools, methods, documents, or knowledge bases and data into a dynamic, distributed Intelligent Master Model with its underlying services. Some services are generic (for example optimization algorithms, or knowledge-based systems), and thus, are not associated with a particular IMM context but are globally available within FIPER. Members of a federation agree on basic notions of administration, identification, and policy. The resulting federation provides the simplicity of access, ease of administration and support for sharing services provided by a large monolithic system, while retaining the flexibility, and control provided by a plug-and-play environment.

FIPER supports three centricities and deploys three neutralities. FIPER's three centricities are network centricity, service centricity, and web centricity. FIPER is composed of various service providers; any of these can come and go and the system can respond to changes in its environment in a reliable way (network centricity). The services connected to FIPER discover each other and cooperate in a distributed environment (service centricity). Users can request to use multiple services and check the status of their submissions in different locations through HTTP portal with thin web clients (web centricity).

The three neutralities FIPER deploys are location neutrality, protocol neutrality, and implementation neutrality, Figure 7. Services need not be co-located; they are discovered and joined, which simplifies management of the entire software environment (location neutrality). In addition, the way clients communicate with a service provider is not essential. A service proxy can use any protocol, for example, Remote Method Invocation (RMI), IIOP or even a plain socket communication. Clients are not aware of what protocols are used and where the implementations reside (protocol neutrality). Furthermore, the clients who use the FIPER services do not need to know what languages are used and how a service is implemented (implementation neutrality). In all, FIPER provides accessibility through web centric architecture, self-manageability using federated services, scalability via network centricity, and adaptability with the power of plugging-and-playing capability.

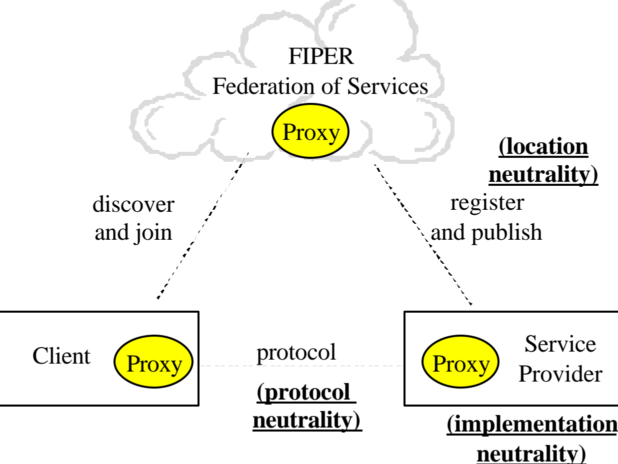


Figure 7: FIPER's Three Neutralities

FIPER's federated architecture is based on Java and Sun's emerging Jini™ software system (Figure 8). The overall goal is to turn the network into a flexible, easily administered tool on which resources can be found by humans or computational clients. The Jini™ system consists of:

1. A set of components that provides the infrastructure for federating services in a distributed environment
2. A programming model that supports the production of reliable distributed environment
3. The functionality to register services and resolve service requests

Java and the emerging Jini™ technology are at the heart of this technology. Services are found and resolved through a "lookup" service (Figure 8). New services are added to the look-up service by a process called discovery and join. When plugged into the environment, the service first uses a discovery protocol to locate an appropriate lookup service and then joins, or registers, with the lookup service. Services can communicate with any other generic service in the entire federated product space. In the case of FIPER, this is achieved by an IMM context, user, or service posting a need which is resolved by a lookup service. The lookup service connects the requesting entity to an entity that has the functionality to supply the service. Figure 8 illustrates this in a given space with four services; CAD, KBE, Optimization and Robust Design, and the Simulation Engine. Each service provider must be Java wrapped in order to join the federation, but it can have its own framework of execution. A service could be based on RMI, CORBA, Java Native Interface (JNI), Microsoft COM/DCOM, or even simple socket connection.

Clients define and submit their jobs via web browsers. A FIPER service manager then dispatches each job into tasks. These tasks can be executed sequentially, in parallel, or combination of both in the FIPER environment, depending on their input/output data dependency. If a parallel strategy is chosen, tasks are dropped into spaces (by using JavaSpaces, for example) for distributed computation. Each service provider agent, if present, picks up appropriate tasks and generates results back to the spaces. On the other hand, FIPER provides a service catalog for direct task execution. The catalog discovers all FIPER services and maintains a list of currently active ones. Appropriate registered service providers will then be selected to perform tasks. Finally, a service manager collects all the outputs and informs the FIPER notification manager about the outcome. The results are presented to the clients when they request.

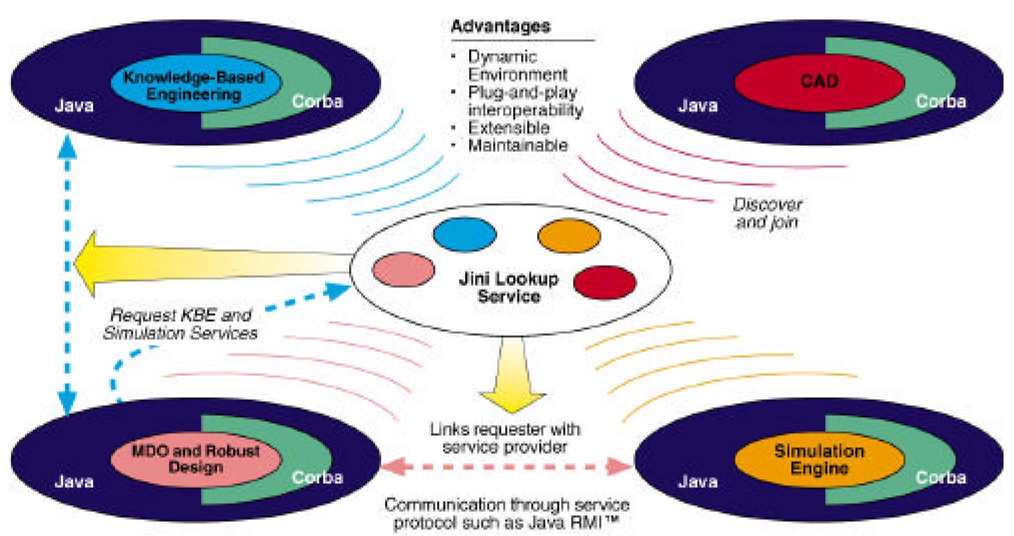


Figure 8: Web-Based FIPER Architecture

This environment promotes concurrency and ensures that current and consistent information is employed throughout the distributed system. The dynamic nature of this approach allows services to be added (for example, support for an additional CAD system) or withdrawn from a federation at any time. The federated environment enables transparent communication between the globally distributed IMM contexts and services, thus providing the means to solve distributed complex tasks such as intelligent scaling of entire systems (e.g., an aircraft engine) and MDO problems.

Engineering Services

The basic premise of FIPER is that everything is on the network and everything on the network is viewed as a service. With this in mind FIPER can contain any "service" needed to support a product throughout its life cycle. For example, services for customer requirements, design, manufacture, sales, distribution, maintenance, and disposal can all be supported by FIPER. For the purpose of the NIST project FIPER will focus on the services necessary for the design and manufacture of a product. Specifically the domains of Design for Six Sigma (DFSS)/MDO, CAD/KBE, Engineering Analysis & Sensitivities, Pre/Post processing, and Data Repositories will be addressed.

This is illustrated in Figure 9. For an initial application the services required for the mechanical analysis of an aircraft turbine component will be developed. These services consist of associative parametric solid geometry modeling, meshing, boundary/initial condition application, and analysis solution. Although all of these services could potentially be provided by one monolithic system, this is rarely the case in today's design environment. Tools for these different services are selected based on many varying criteria ranging from "best in class" to corporate mandate. Figure 10 shows the relationship of these services. Although a very simple case, it can be used to demonstrate the FIPER "service" paradigm in a distributed heterogeneous computing environment.

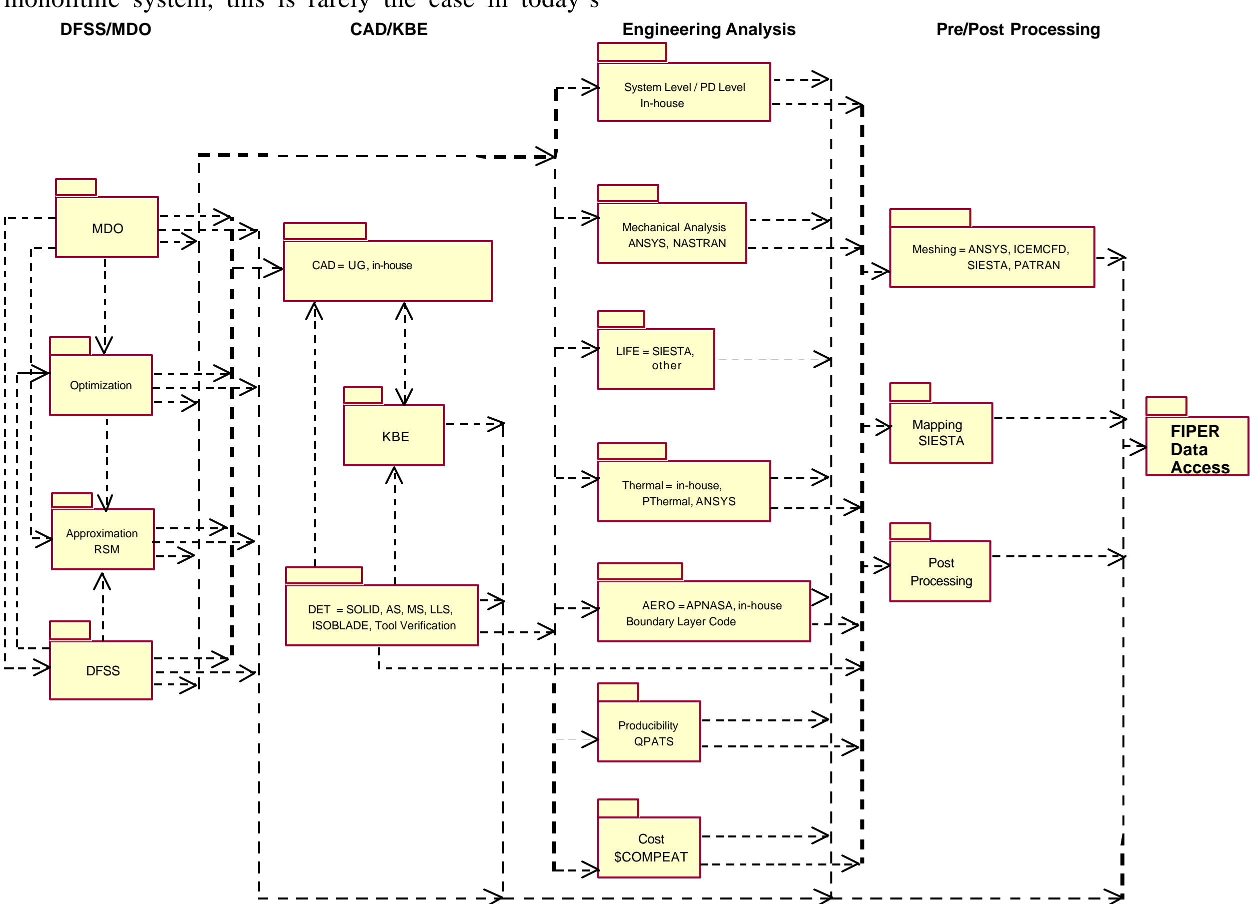


Figure 9: Services Package Diagram

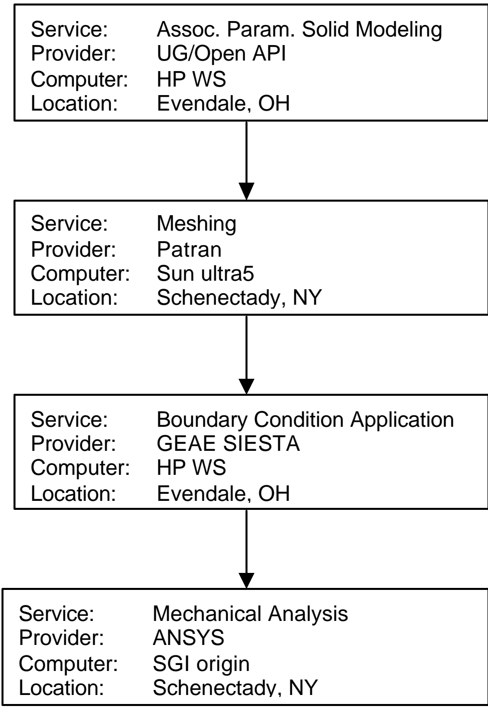


Figure 10: Analysis Flow Chart

Parametric Solid Geometry Service

This service generates the necessary solid geometry that will be meshed and analyzed. In this example the provider for this is a Unigraphics User Function (UFUNC) program that requires an initial seed part and a parametric data fle as input. With these inputs the program constructs a three-dimensional solid of the component and associates attributes or "tags" with various geometric entities (surfaces, edges, etc.). These tags will be identifiable and used by other services in the system such as meshing and boundary condition application. The UG UFUNC program is "wrapped" as a FIPER service and deployed on a Unix workstation in a remote location.

Meshing Service

The meshing service discretizes a given component. As input, it requires a geometric entity and some information describing a strategy for meshing the supplied geometry. The meshing strategy contains information such as mesh seeding parameters and element types. These attributes are mapped to the geometry via the associated geometric tags. For the present case MSC PATRAN[7] is the service provider. A wrapper is written for PATRAN that takes the meshing strategy information and generates PATRAN PCL. With the PCL the wrapper invokes PATRAN which in turn produces a mesh for the given geometry. The service exports the meshed geometry in the form of a PATRAN neutral file. As shown in Figure 10, the meshing service resides on a local Unix work station. It is worthwhile to note that all the geometric tags that were created in the solid geometry service are transferred onto the descritized geometry. Thus, the tags can continue to be used to identify particular attributes of the model. These will be available to other services.

Boundary Condition Application

The boundary condition service applies a set of boundary conditions to a given meshed geometry or group of geometries. It requires as input a descritized geometry (PATRAN neutral file is one acceptable format) and information describing the boundary conditions to be applied (specified displacements, temperatures, etc.). Here, a GEAE in-house application called SIESTA is the service provider. SIESTA is wrapped as a FIPER service and published at a remote Unix workstation. The wrapper accepts as input a PATRAN neutral file and generic boundary condition information. The wrapper produces SIESTA native commands that apply the specified boundary conditions to the meshed model. As in the case of the meshing the geometric tags are utilized to associate boundary conditions to particular geometric features. The output from this service is in a form suitable for a particular engineering analysis application such as ANSYS®[,8]

Analysis Solution

Once the model has been meshed, boundary conditions applied, and materials selected, the model is ready for solution. This service takes the specified input and invokes the appropriate solver on the model. In the current study ANSYS® is wrapped as a FIPER service. The wrapper takes as input an ANSYS® input file and simply issues a system call which executes ANSYS®. The results of the service are returned in the form of VRML (Virtual Reality Modeling Language) files that summarize the results. The ANSYS® service is located on a local high end compute server.

Use Cases

In order to determine the required functionality of FIPER, a set of use cases was developed. The use cases were divided into three major categories: System level analysis/design, sub-system analysis/design, and component analysis/design. FIPER should be flexible enough to handle requirements in all these regions. The use cases are represented by use case diagrams and sequence diagrams. Standard terminology as specified in the Unified Modeling Language (UML)[9] is used. A use case diagram and a sequence diagram for the

mechanical analysis of a turbine component is shown in Figures 11 and 12.

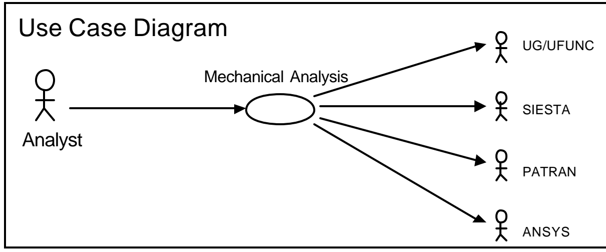


Figure 11: Use Case Diagram

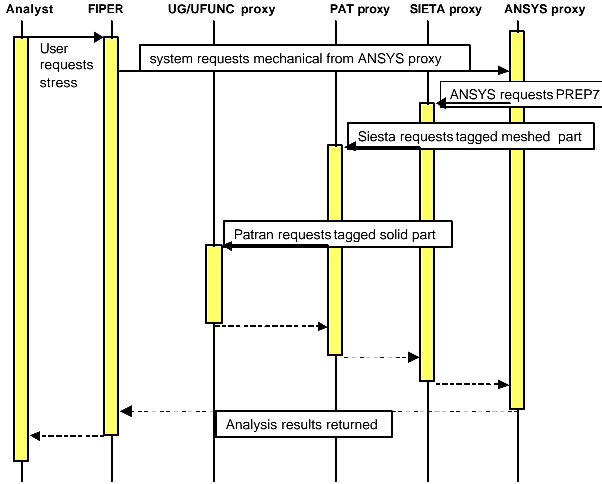


Figure 12: Sequence Diagram for Turbine Mechanical Analysis

The system level use cases focus on the "intelligent scaling" of a given system, for example an entire gas turbine engine. Intelligent scaling refers to the resizing of the system based on the use of a KBE system. The KBE contains rules ranging from standard design practices, simple empirical equations, to the invocation of high-end engineering analysis codes. Once the system level resizing is complete, FIPER should support the ability to "zoom" in on a given component or subsystem and perform a detailed analysis to verify the results produced by the intelligent scaling. The sub-system use cases address a collection of components and employ preliminary level analysis applications along with some detailed analysis level applications. These use cases also include the use of formal optimization techniques to aid in performing robust and optimal design.

The last class of use cases, component level analysis/design, addresses the requirements for performing MDO/robust design with high fidelity analysis codes such as FEM and CFD.

Outlook

The material presented in this paper represents the first six months of work into a four-year research contract. While a lot of effort has been put into developing the architecture, defining the Intelligent Master Model, and setting up a suite of use cases representing typical problems encountered in turbine engine development, a number of technical risks still remains. Web technology is developing rapidly, but so far the engineering community has leveraged very little of these emerging tools. Sun's Jini™ technology was intended for distributed hardware devices, not software, yet conceptually there is no reason why it should not be applicable in this type of environment.

The example presented, the turbine blade mechanical analysis, is the first use case - and FIPER demonstration that - is currently being implemented. Over the next three years, these use cases will be expanded to cover the range from component to subsystem and system level design, analysis, and optimization, up to the intelligent scaling of a complete turbine engine core.

If the project is successful, it will constitute a complete paradigm shift in the use of engineering software. The authors anticipate to keep the scientific community informed about the progress of the work through subsequent publications and conference presentations.

Acknowledgments

This research is jointly funded through the National Institute for Standards and Technology-Advanced Technology Program (NIST-ATP™) and the General Electric Company. The authors would like to acknowledge this support, as well as the valuable input from the whole FIPER team.

References

[1] Röhl, P.J.; He, B.; Finnigan, P.: *A Collaborative Optimization Environment for Turbine Engine Development*, AIAA 98-4734, Proceedings, 7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, St. Louis, MO, September 1998

[2] He, B.; Röhl, P.J. et al.: *CAD and CAE Integration with Application to the Forging Shape Optimization of Turbine Disks.* Proceedings, 39th AIAA/ ASME/ASCE/AHS/ASC Structural Dynamics, and Materials Conference, Long Beach, CA, April 1998

[3] Federated Intelligent Product Environment, Technical Proposal, Ohio Aerospace Institute, General Electric, BFGoodrich, Parker Hannifin,

Engineous Software, Ohio University, Stanford University, April, 1999

[4] Bailey, M.W.; Irani, R.K.; et al.: *Integrated Multidisciplinary Design*, Presented at the XIV ISABE conference, Florence, Italy, September, 1999

[5] Unigraphics V15 User Documentation, Unigraphics Solutions, Cypress, CA, 1999

[6] Intent User Manual, Heide Corporation, Medfield, MA, 2000

[7] PATRAN V8.0 User Documentation, MacNeal-Schwendler Corporation, Costa Mesa, CA, 1999

[8] ANSYS V5.5 User Documentation, ANSYS Inc., 1999

[9] The Unified Modeling Language User Guide, G. Booch, J. Rumbaugh, I. Jacobson, Addison Wesley Longman Inc., 1999

SORCER: Federated Grid Computing

June 16, 2004

http://sorcer.cs.ttu.edu

Michael Sobolewski
sobol@cs.ttu.com

Ravi Malladi
Abhijit Rai

Presentation Agenda

- What is SORCER?
- Evolution of Computing
- The Service-Driven Network
- Three Centricities
- Code Mobility
- SORCER Architecture Qualities
- Discovery, Join, and Lookup
- From a Grid to Intergrid
- Two Use Cases
  - Surrogate Services for Mobility
  - Federated Grid Dispatching

What is SORCER?

- Service-ORiented EnviRonment (SORCER)
- CS TTU Lab
- Next Generation FIPER
  - Service Centric
  - Network Centric
  - Mobile Code Centric
- Service-Oriented Programming Philosophy
- Federated Grid Computing

SORCER 1.5 Years Old

- 15 P2P Projects
- 11 Mobile Computing Projects
- 7 Theses (1 defended, 6 proposed)
- 16 Publications
- Development
  - FIPER Enhancements (GE GRC, GE AE)
  - S-BLAST (USDA-ARS)
  - SORCER Proth (HPCC, Mathematics)
  - IT TTU Doc Manager (IT TTU)
  - CE2004 Doc Manager and conference website (ISPE), http://www.ce2004.org

SORCER Lab Net

perle2:
129.118.29.67

cs.ttu.edu

macsobol:
129.118.18.170

sobol:
129.118.18.172

amber:
129.118.29.74

coral:
129.118.29.73

jade:
129.118.29.7

pipal:
129.118.18.134

.18.205

adelie:
129.118.18.183

cpxw049:
129.118.18.230

cs.ttu.edu

opal:
129.118.18.203

topaz:
129.118.18.204

GS108, CP 310

yucca2.cs.ttu.edu
129.118.29.100

Window Server 2003

neem2: 129.118.29.99

yucca.cs.ttu.edu
129.118.18.88

Red Hat 7.3

neem: 129.118.18.18

beryl:
129.118.29.71

yew2: 129.118.29.98

Solaris 8

yew: 129.118.18.41

cpxw048:
129.118.18.229

Mac OSX

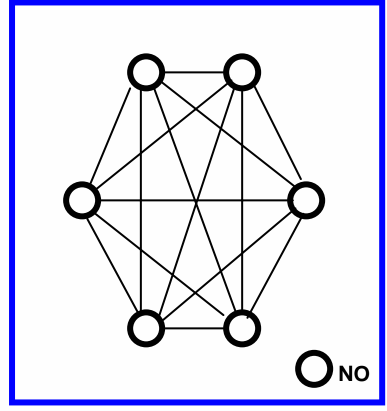
poppy: 129.118.29.110

Pioneer PDP-503CMX 50 in

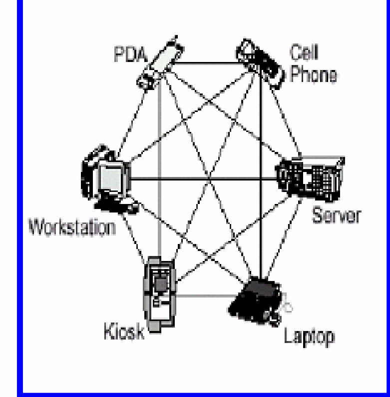
poppy2: 129.118.18.9

Evolution of Computing

From servers …

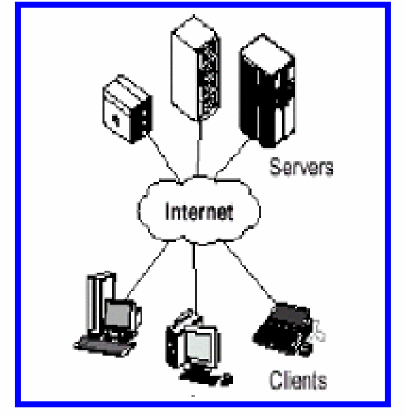


Service-to-Service



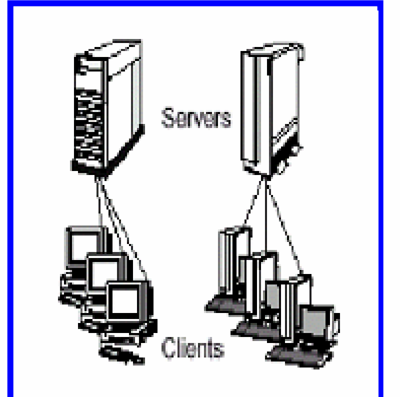
Peer-to-Peer

… to network objects



Web-based computing

- virtual overlay network
- interactive SOP
- federated SOC
- secure
- self-healing
- autonomic
- heterogeneous



Client-server silos

Waves of Network Computing

Waves of Network Computing

Computers 10^{8}

Embedded Computers 10^{11}

Things - 10^{14}

Desktops

Phones
TVs
Cars
Games

Thermostats
Switches
Packages
Clothes

Clients

Functions

Transfers
Transactions
Content
Telemetry
Control

IP v4 — IP Layer → IP v6

Protocols

Organization

FTP
SMTP
X
Telnet
RPC/XDR
HTTP
RMI/IIOP
LDAP
SOAP
UDDI
Identity
Jini
JXTA

Client/Server
N-tier
Web Applications
Web Services
Polyarchical
Fractal

Network is Not Enough

We need to confront Deutsch's Eight Fallacies of the Network

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

End-to-End Computing

Devices & Appliances Tier

Client Desktop Tier

Edge Tier

App Server Tier

Data Center

Java is the glue that enables end-to-end computing

Java™ Technology

Leads the Way…

New application development by language (Soundview TG, 2001)

Percent (%)

100
75
50
25
0
-25
-50
-75

Java technology continues to eclipse other development environments

Java
VB
C/C++
PB
Small Talk
Cobol
PL/I

Source: Soundview Technology Group

Devices vs. Profiles

workstation
server
communicator
POS
pager
NC
PDA
PC, laptop
screen-phone
set-top box, net TV
smartphone
cell phone
card

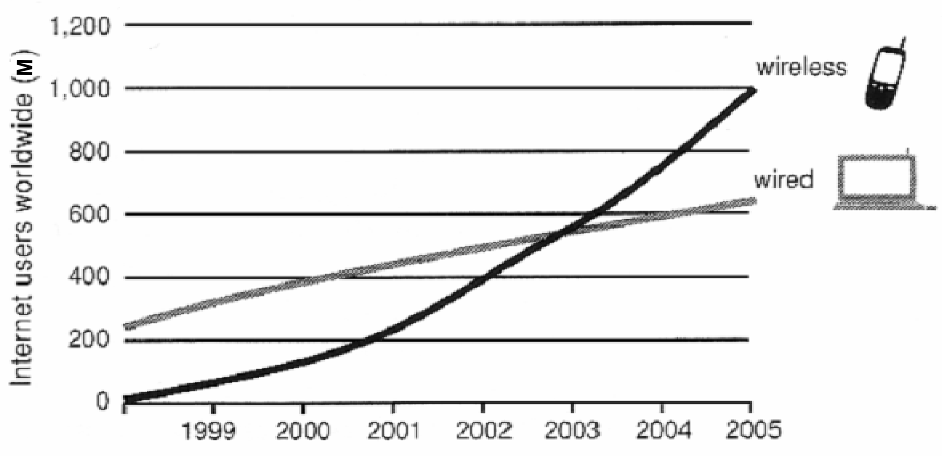
Java 2 Enterprise Edition
Java 2 Standard Edition
Java 2 Micro Edition
Java Card
Java Platform

HotSpot | Classic VM | K VM | Card VM

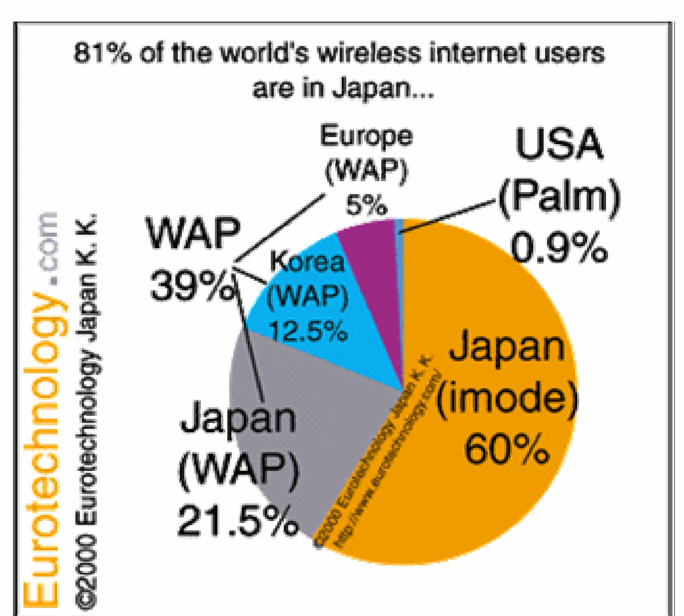
Memory: 10MB ↔ 1MB 500kB ↔ 10kB

64 bit | 32 bit | 16 bit | 8 bit

From Wired to Wireless



Wireless devices overtake the Internet (2001 Motorola Inc.)



[12/2003]

Global Mobile Users	1.3 billion
Analogue Users	34m
US Mobile users	140m
Global GSM users	870m
Global CDMA Users	164m
Global TDMA users	120m
Total European users	320m
#1 Mobile Country	China (200m)
#1 GSM Country	China (195m)
#1 SMS Country	Philippines

SORCER Vision

Service

Service

Service

Federated S2S environment to ...

- Service — Build new services
- Service — Convert legacy apps to dynamic SORCER services (J2EE™ technology)
- Assemble SORCER services together (RMI, Jini, Rio, JXTA, WS technologies)
- Create modern clients accessing services

The computer is the service grid that exposes services to clients AWAT

Pervasive SORCER.grid

The Edge

Service Interface

SORCER.grid

Service Provider

Peers/Groups

Surrogates

Mobile Devices …

Knowledge Systems

Soft Computing

Data Persistence and Mining

Image Processing/Analysis
Computer Vision
Multimedia

Distributed Computing

Security

Parallel/Space Computing

Large Scale Systems

Mobile Computing

SORCER Paradigm

- Clients Request Services from the Network
  - DOE Services
  - Analysis Services
  - Optimization Services
- Clients may not care where or who supplies the services

Geometry
Ansys Solve
Mesh
Materials
BC

SORCER Service: An entity that publishes (by attributes) functional capabilities on the network. (Mesh, Thermal Analysis, Print, etc.)

The Network is a Virtual Computer that Exposes Services to Clients AWAT

Nozzle Combustor CAD/IO B2B

(UG)

(ProE)

Parker Aerospace

1. Update combustor PCS

2. Request for nozzle validation

3. Check for nozzle insertion

4. Perform modal analysis

5. Perform CFD blow analysis

(Blow Analysis)

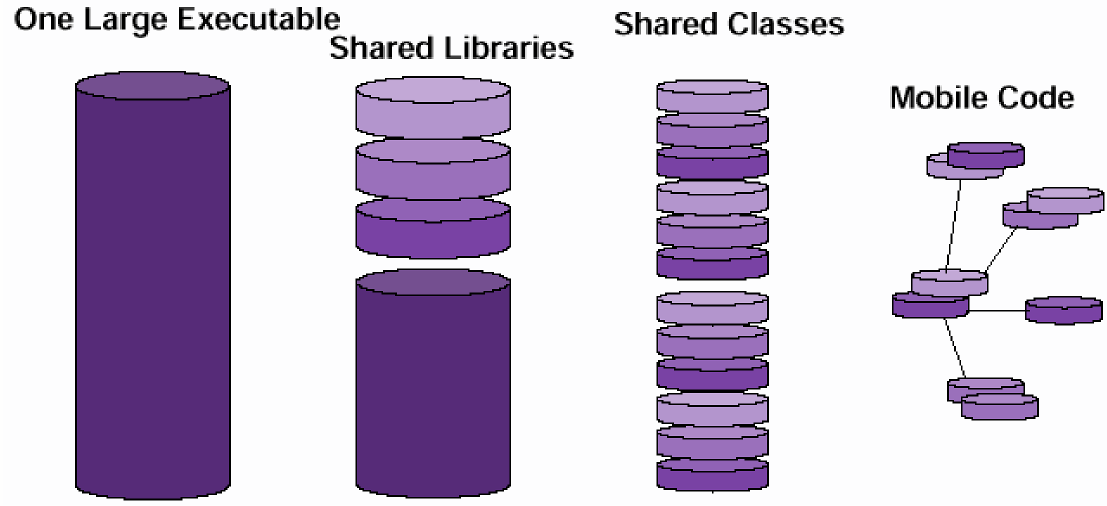
SORCER's 3Cs

- Service Centricity - Federation of services
- Network Centricity - N-1, 1-1, 1-N, N-C (Services discover each other)
- Web Centricity - HTTP Portal with thin web clients

Applying OO techniques to the network

Shrinking Programs

- OLE - One Large Executable
- Shared Libraries
- Share classes
- Mobile Code



Program units becoming smaller and mobile

Change Over Time

- Needs of the system evolve faster than the system
- Many decisions implemented in runtime
- It’s less about ***knowing*** and more about ***not knowing***

Access to a network
with reconnection to the network

Interface vs. Implementation

- Interfaces are forever
- Implementation is for now, can change
- Mobile code allow multiple implementations

Requestors need to know a service interface; what not how

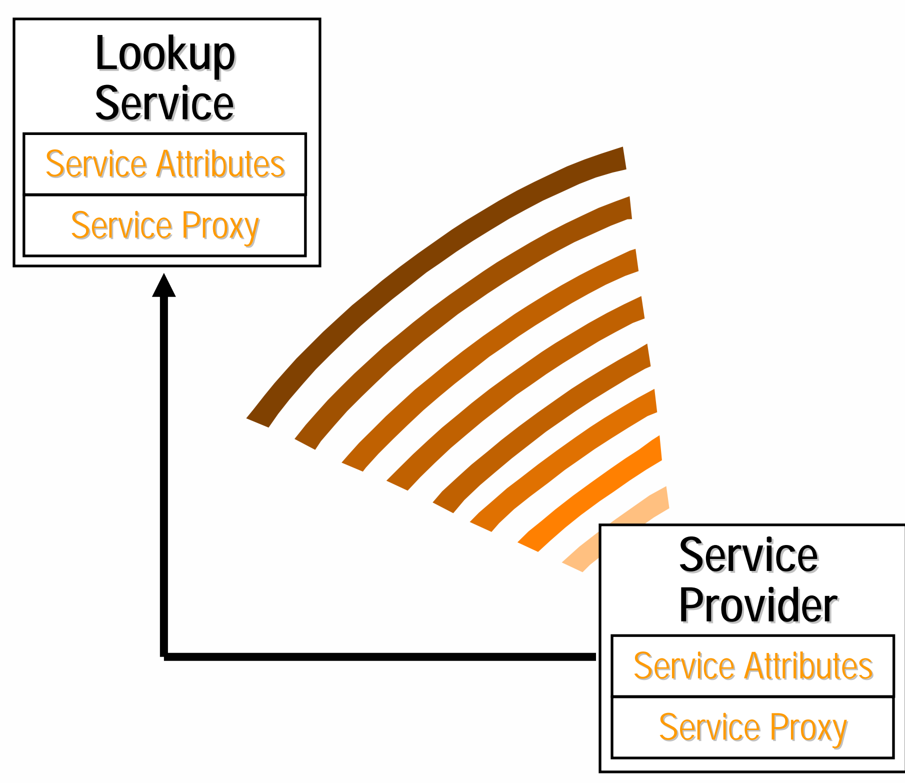
Architecture Qualities

- Accessibility - Web centricity
- Manageability - Federated services
- Scalability, Reliability - Network Centricity
- Adaptability - Mobile Code

Services appear as network objects identified by type

Real Distributed Objects

Provider Discovery and Join



A Service Provider Seeks a Lookup Service

A Service Provider Registers with Lookup Service

Requestor Discovery and Lookup

TEXAS TECH UNIVERSITY
Computer Science

Discovery , Lookup & Communicate

Lookup Service
Service Attributes
Service Proxy

Client
Service Attributes
Service Proxy

Service Provider

A Client Seeks a Lookup Service &
a Service with the Specified Attributes
Client Receives a Copy of the Service Proxy
Client Interacts Directly with Service Provider

Service-to-Service (S2S)

Network objects

What does it mean to be a service?

A service is an act of requesting a `service(Exertion)` operation from a service provider.

Service Requestor

`service(exertion)`

Service Provider

If accepted
then
`exertion.exert();`
else
forward to a relevant service provider

ServiceInterface *1* ,…, ServiceInterface *i*

signature *1* ,…, signature *k*

operation *1* , …, operation *l*

Impl1

Clusters, Federations, Exertions

Exertions
SO Programs

Self-organized
service providers
allocated to best
resource

Dynamic QoS
provisioning grid

Compute
resource
management grid

Compute
resource

Vertical iGrid Grids

Exertions – iGrid.space

SS Beans – iGrid.field

Service Providers – iGrid.grid

SORCER.grid

SORCER.core

Cybernodes – iGrid.mesh

Computing Devices – iGrid.net

iGrid.grid – service providers including services from technology (horizontal) grids
SORCER.core – SORCER infrastructure service providers
SORCER.grid – SORCER domain specific service providers

SORCER Functional Architecture

Utilities and Templates

Web CME User Agents

Requestor

Service UIs

Intraportal

Extraportal

Infrastructure Service Providers (SORCER.core)
Jobber, Spacer, Cataloger, Provisioner, Persister, Tasker, Caller,
FileStorer, Notifier, Reporter, Monitor, Profiler, Securer, Auditor

Federated Service Provider
Service Provider, Exertion Dispatchers, Proxy,
Service Bean, Data Accessor, ProviderAccessor

Persistence Layer

Static Bootstrapping
Service Joiner

File Store

CME Layer

J2EE, Jini, Rio, GApp

Mobile Devices Support

Java 2 Enterprise Edition

Java 2 Standard Edition

Personal Profile

RMI Profile

Foundation Profile

TV Profile

Auto Profile

Handheld Profile

MID Profile

J2ME CDC

J2ME CLDC

Smart Card Profile

HotSpot

CVM

KVM

Card VM

Surrogate Services

SORCER.grid

Jini Capable Machine

Surrogate Host

Surrogate Host Resources

Other Exported Service

Exported Service

Inter-connect Specific Code

Private Protocol

HTTP/HTTPS

Provide Service

Mr. X

Request

Deploy Calendar Service

Interaction Using Private Protocol

SORCER Calendar Service Created

Surrogate

Request

Deploy Calendar Service

Client

Service UI

Mr X service

Get me Calendar of Mr. X

SORCER.grid

Jobber

Service Oriented Program

SORCER service

Deployed SORCER SUROGATE Service

Grid Dispatcher UI

Job Dispatcher

Set Application Parameters

Program Name: SORCER - Proth

Job Size:

Notify:

Arguments | Attributes | Executables

Run | Clear

Input File : Browse | Insert

Input List	Output List

- Choose the Application to run (For example Proth)
- Specify the Job Size for the jobs
- Set the Arguments, Attributes and Executables for the application

Arguments UI

Specify Arguments and Input files

Arguments | Inputs | Outputs

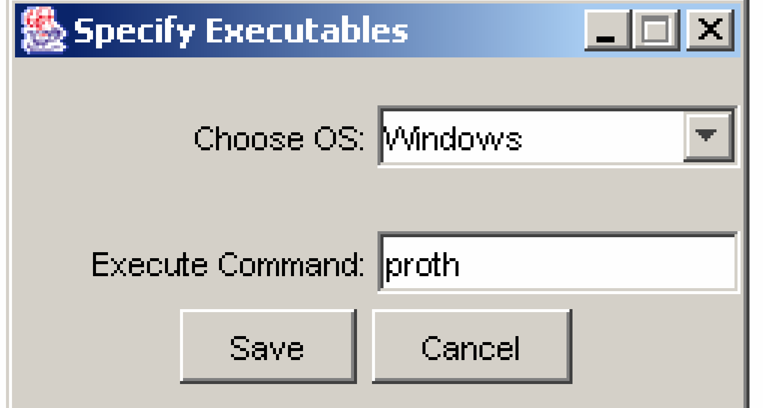
Up Down Delete | Up Down Delete | Up Down Delete

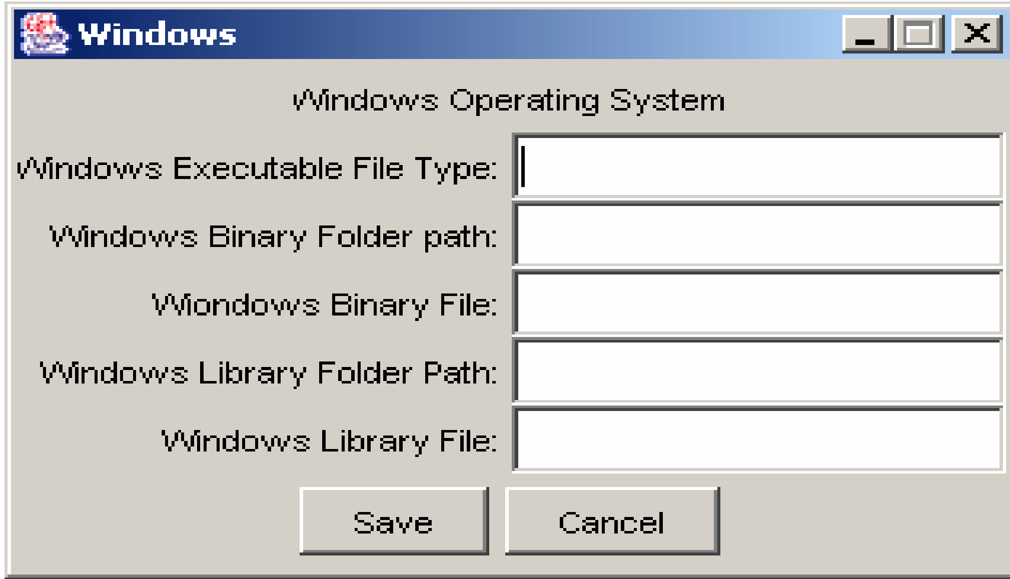
Arguments
Above Add
Inputs
Above Add
Outputs
Above Add

Save Cancel

- Specify Arguments, Input Files, Output Files for the Application
- Can be added above or below the selected option
- Can be reordered according to user's requirement (Up, Down, Delete buttons)

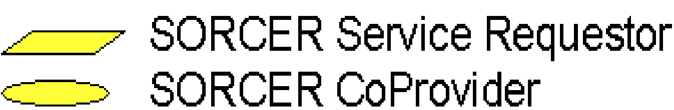
Executables - Windows

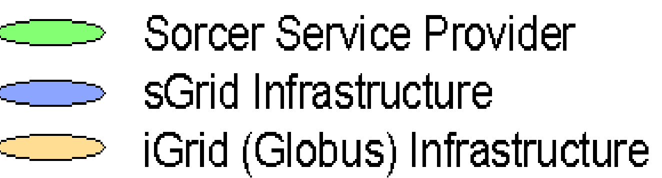




- Specify Windows Executables and Library Files
- The files can be dynamically downloaded from File Store

Communication across sGrids





Q&A

TEXAS TECH
UNIVERSITY
Computer Science
Ravi Malladi
SORCER LAB