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 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-

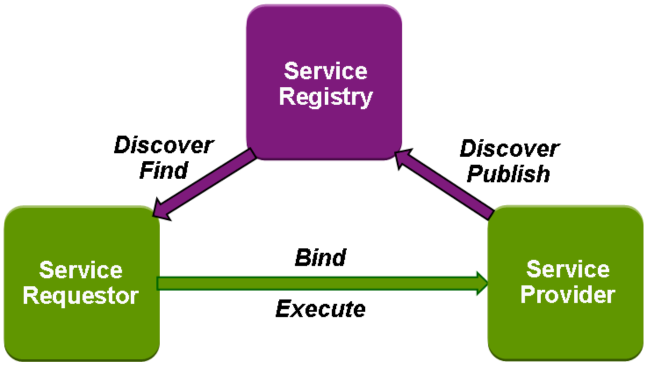

Figure 1. Service oriented architecture.

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.

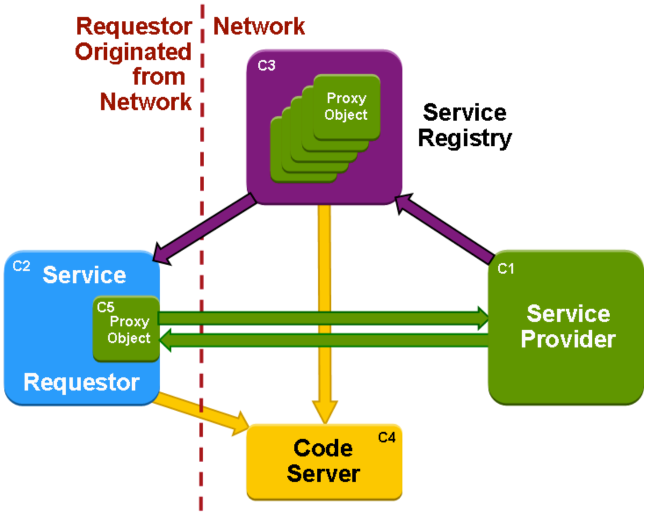

Figure 2. Service object-oriented architecture.

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 behavior. 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-

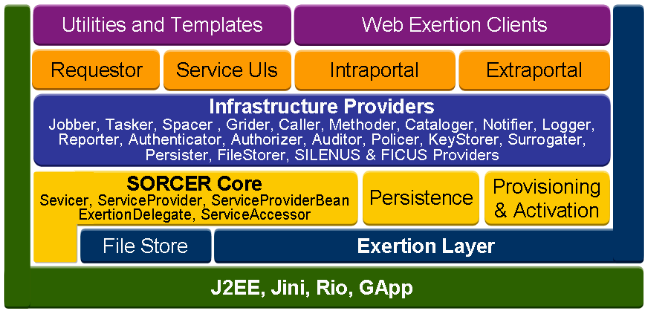
Utilities and Templates
Web Exertion Clients
Requestor
Service UIs
Intraportal
Extraportal
Infrastructure Providers
Jobber, Tasker, Spacer , Grider, Caller, Methoder, Cataloger, Notifier, Logger, Reporter, Authenticator, Authorizer, Auditor, Policer, KeyStorer, Surrogater, Persister, FileStorer, SILENUS & FICUS Providers
SORCER Core
Sevicer, ServiceProvider, ServiceProviderBean ExertionDelegate, ServiceAccessor
Persistence
Provisioning & Activation
File Store
Exertion Layer
J2EE, Jini, Rio, GApp

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

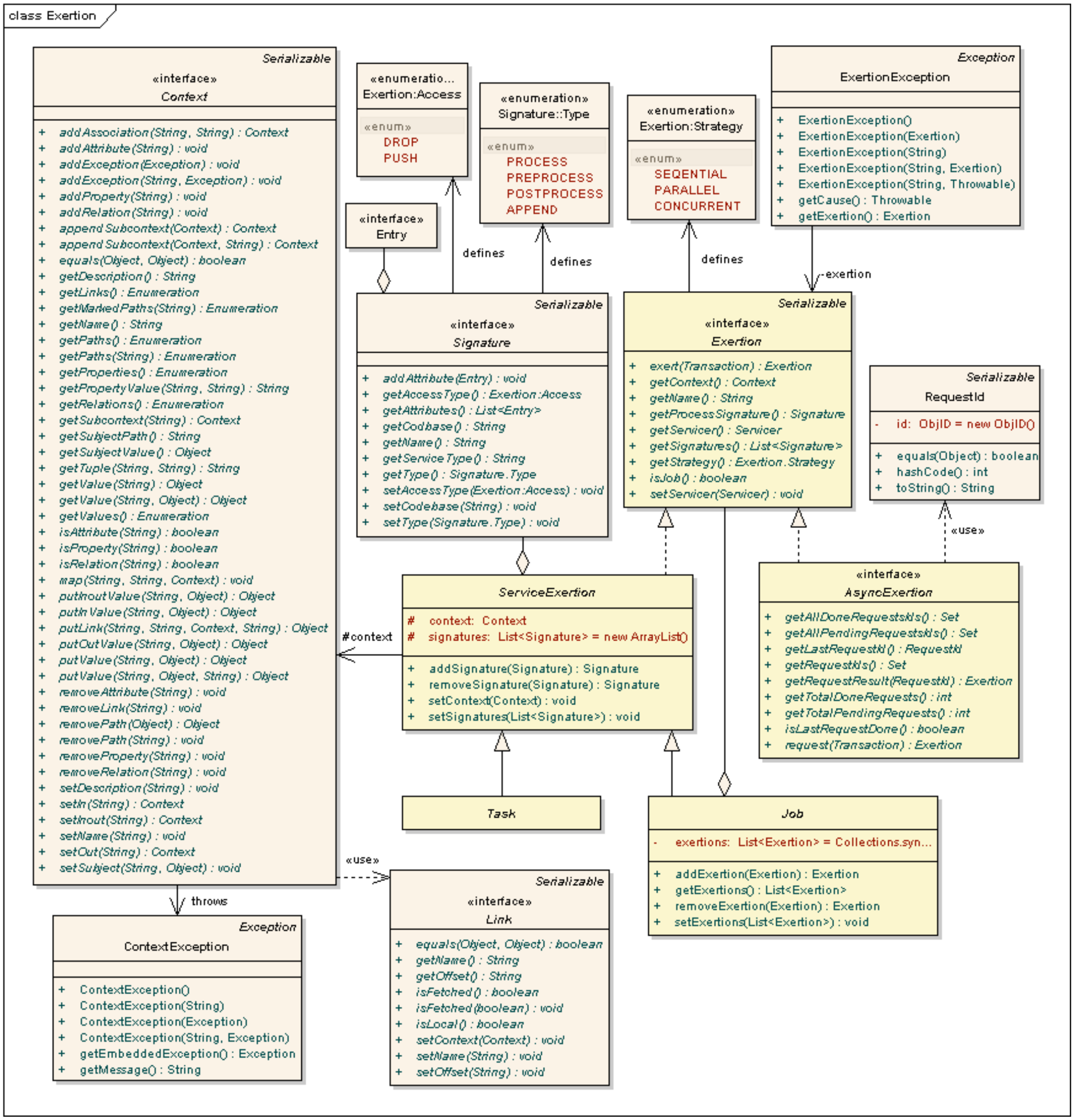
class Exertion
Serializable
«interface»
Context
+ addAssociation(String, String) : Context
+ addAttribute(String) : void
+ addException(Exception) : void
+ addException(String, Exception) : void
+ addProperty(String) : void
+ addRelation(String) : void
+ appendSubcontext(Context) : Context
+ appendSubcontext(Context, String) : Context
+ equals(Object, Object) : boolean
+ getDescription() : String
+ getLinks() : Enumeration
+ getMarkedPaths(String) : Enumeration
+ getName() : String
+ getPaths() : Enumeration
+ getPaths(String) : Enumeration
+ getProperties() : Enumeration
+ getPropertyValue(String, String) : String
+ getRelations() : Enumeration
+ getSubcontext(String) : Context
+ getSubjectPath() : String
+ getSubjectValue() : Object
+ getTuple(String, String) : String
+ getValue(String) : Object
+ getValue(String, Object) : Object
+ getValues() : Enumeration
+ isAttribute(String) : boolean
+ isProperty(String) : boolean
+ isRelation(String) : boolean
+ map(String, String, Context) : void
+ putInoutValue(String, Object) : Object
+ putInValue(String, Object) : Object
+ putLink(String, String, Context, String) : Object
+ putOutValue(String, Object) : Object
+ putValue(String, Object) : Object
+ putValue(String, Object, String) : Object
+ removeAttribute(String) : void
+ removeLink(String) : void
+ removePath(Object) : Object
+ removePath(String) : void
+ removeProperty(String) : void
+ removeRelation(String) : void
+ setDescription(String) : void
+ setIn(String) : Context
+ setInout(String) : Context
+ setName(String) : void
+ setOut(String) : Context
+ setSubject(String, Object) : void
throws
Exception
ContextException
+ ContextException()
+ ContextException(String)
+ ContextException(Exception)
+ ContextException(String, Exception)
+ getEmbeddedException() : Exception
+ getMessage() : String
«enumeratio...
Exertion::Access
«enum»
DROP
PUSH
«interface»
Entry
defines
«enumeration»
Signature::Type
«enum»
PROCESS
PREPROCESS
POSTPROCESS
APPEND
defines
«enumeration»
Exertion::Strategy
«enum»
SEQENTIAL
PARALLEL
CONCURRENT
defines
Exception
ExertionException
+ ExertionException()
+ ExertionException(Exertion)
+ ExertionException(String)
+ ExertionException(String, Exertion)
+ ExertionException(String, Throwable)
+ getCause() : Throwable
+ getExertion() : Exertion
-exertion
Serializable
«interface»
Signature
+ addAttribute(Entry) : void
+ getAccessType() : Exertion::Access
+ getAttributes() : List<Entry>
+ getCodbase() : String
+ getName() : String
+ getServiceType() : String
+ getType() : Signature.Type
+ setAccessType(Exertion::Access) : void
+ setCodebase(String) : void
+ setType(Signature.Type) : void
Serializable
«interface»
Exertion
+ exert(Transaction) : Exertion
+ getContext() : Context
+ getName() : String
+ getProcessSignature() : Signature
+ getServicer() : Servicer
+ getSignatures() : List<Signature>
+ getStrategy() : Exertion.Strategy
+ isJob() : boolean
+ setServicer(Servicer) : void
Serializable
RequestId
- id: ObjID = new ObjID()
+ equals(Object) : boolean
+ hashCode() : int
+ toString() : String
«use»
#context
ServiceExertion
# context: Context
# signatures: List<Signature> = new ArrayList()
+ addSignature(Signature) : Signature
+ removeSignature(Signature) : Signature
+ setContext(Context) : void
+ setSignatures(List<Signature>) : void
«interface»
AsyncExertion
+ getAllDoneRequestsIds() : Set
+ getAllPendingRequestsIds() : Set
+ getLastRequestId() : RequestId
+ getRequestIds() : Set
+ getRequestResult(RequestId) : Exertion
+ getTotalDoneRequests() : int
+ getTotalPendingRequests() : int
+ isLastRequestDone() : boolean
+ request(Transaction) : Exertion
Task
Job
- exertions: List<Exertion> = Collections.syn...
+ addExertion(Exertion) : Exertion
+ getExertions() : List<Exertion>
+ removeExertion(Exertion) : Exertion
+ setExertions(List<Exertion>) : void
«use»
Serializable
«interface»
Link
+ equals(Object, Object) : boolean
+ getName() : String
+ getOffset() : String
+ isFetched() : boolean
+ isFetched(boolean) : void
+ isLocal() : boolean
+ setContext(Context) : void
+ setName(String) : void
+ setOffset(String) : void

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

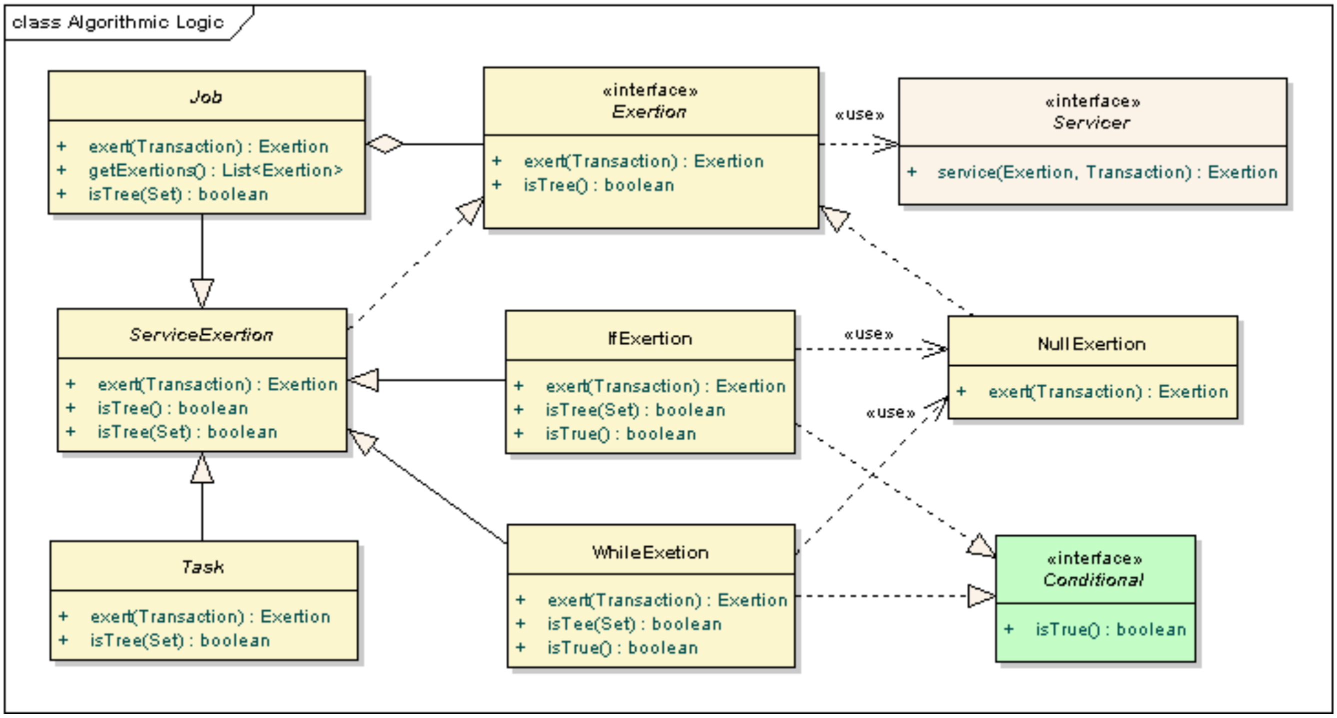

Figure 5. Flow control exertions, conditional IfExertion and looping WhileExertion, used in SORCER.

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

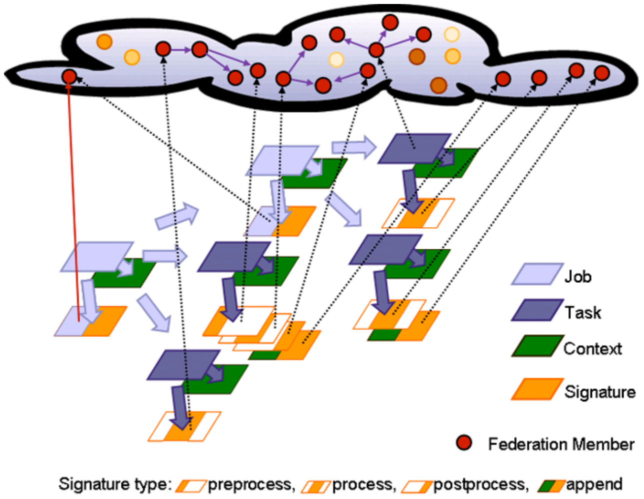

Figure 6. A job federation. The red line (the first from the left) indicates the originating FMI invocation: Exertion.exert(Transaction) or Servicer.service(Exertion, Transaction). The root job with component exertions is depicted below the provider grid (a cloud). Late bindings of all signatures are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

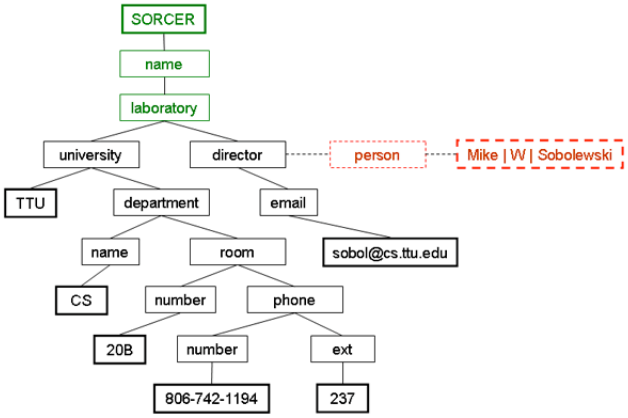

Figure 7. Example of a service context.

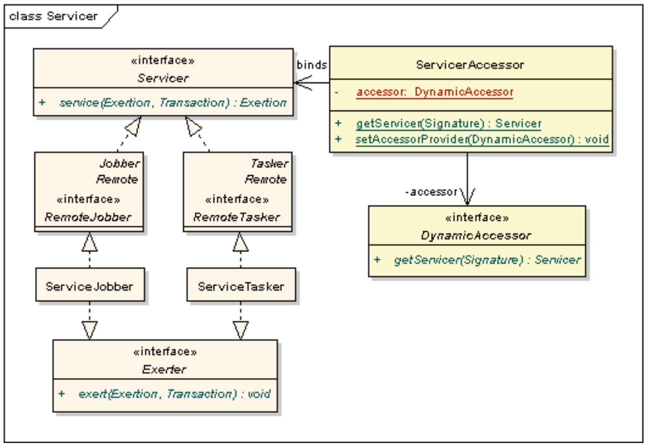
class Servicer
«interface»
Servicer
+ service(Exertion, Transaction) : Exertion
binds
ServicerAccessor
- accessor: DynamicAccessor
+ getServicer(Signature) : Servicer
+ setAccessorProvider(DynamicAccessor) : void
-accessor
Jobber
Remote
«interface»
RemoteJobber
Tasker
Remote
«interface»
RemoteTasker
«interface»
DynamicAccessor
+ getServicer(Signature) : Servicer
ServiceJobber
ServiceTasker
«interface»
Exerter
+ exert(Exertion, Transaction) : void

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, File-Storers, 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)