IADIS International Journal on Computer Science and Information Systems
Vol. 3, No. 1, pp. 86-109
ISSN: 1646-3692

EXERTION ORIENTED PROGRAMMING

Michael Sobolewski *Computer Science, 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 presented in this paper. Some of them—CORBA, Java RMI, and Web/OGSA 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, difficult to use, with missing core object-oriented traits. A distributed system is not just a collection of distributed objects—it is the network of dynamic objects that come and go. In particular, the object wrapping approach does not help to cope with network-centric messaging, invocation latency, object discovery, dynamic object federations, fault detection, recovery, partial failure, etc. The Jini™ service architecture does not hide the network; it allows the programmer to deal with the network reality to form dynamic service federations. However, handling low-level networking details in Jini does not help to cope with complex network programming to dynamically cluster, federate, and utilize efficiently various network services easily. A new network programming methodology is presented in this paper. It uses the intuitive metacomputing semantics and the Triple Command design pattern. The pattern defines how service objects communicate by sending one another a form of service messages called exertions that encapsulate data, operations, and control strategy.

KEYWORDS

service-oriented programming, metacomputing, grid computing, dynamic computing federations, large-scale distributed systems

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 [2]—Sun RPC (ONC RPC) and DCE RPC, which are language, architecture, and OS independent;
2. Second generation RPCs—CORBA [22] and Microsoft DCOM-ORPC, which add distributed object support;
3. Third generation RPC—Java RMI [19] is conceptually similar to the second generation but supports the semantics of object invocation in different address spaces that are built for Java only. Java RMI fits cleanly into the language with no need for standardized data representation, external interface definition language, and with behavioral transfer that allows remote objects to perform operations that are determined at runtime;
4. Fourth generation RPC—next generation of Java RMI, Jini Extensible Remote Invocation (Jini ERI [18]) with dynamic proxies, smart proxies, network security, and with dependency injection defining exporters, end points, and security properties in configuration files;
5. Fifth generation RPCs—Web/OGSA Services RPC [17, 29] and the XML movement including Microsoft WCF/.NET;
6. Sixth generation RPC—Federated Method Invocation (FMI), presented in this paper, allows for concurrent invocations on multiple federating hosts (virtual metacomputer) in the SORCER environment [27].

All the RPC generations are based on a form of service-oriented architecture (SOA) discussed in Section 2. However, CORBA, RMI, and Web/OGSA 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 does not mean that developed distributed objects have a great deal to do with object-oriented distributed programming. For example, CORBA defines many services, and implementing them using distributed objects does not make them well structured with core object-oriented features: encapsulation, instantiation, and polymorphism. Similarly in Java RMI, marking objects with the `Remote` interface does not help to cope with network-centric messaging, for example when calling on a dead stub. Network centricity here means that sending a message to a remote object, in fact is sending it onto the network in the first place, and then dispatching it to a live remote object provided by the network in a uniform way. Network-centric messaging should encapsulate object discovery, fault detection, recovery, partial failure, and others.

Programmers use abstractions all the time. The source code written in programming language is an abstraction of the machine language. From machine language to object-oriented programming, layers of abstractions have accumulated like geological strata. Every generation of software architects and programmers uses its era's programming languages and tools to build programs of the next generation. Each architecture and programming language used 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. Building on the object-oriented distributed paradigm is the service object-oriented paradigm exemplified by the Jini architecture [12] 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 [27], a service provider is a remote object that accepts network requests to participate in a collaboration—a process by which service providers work together to seek solutions that reach beyond what any one of them could accomplish on their own. While conventional objects encapsulate *data* and *operations*, the network requests called *exertions* encapsulate *data*, *opera-*

tions, and *control strategy*. An exertion can federate transparently on multiple hosts according to its *control strategy* by hiding all low-level Jini networking details as well.

The SORCER metacomputing environment adds an entirely new layer of abstraction to the practice of grid computing—*exertion-oriented (EO) programming*. The EO programming makes a positive difference in service-oriented programming primarily through a new metaprogramming abstraction as experienced in many grid-computing projects including applications deployed at GE Global Research Center, GE Aviation, Air Force Research Lab, and SORCER Lab [21, 8, 9, 14, 15, 24]. The new abstraction is about managing object-oriented distributed system complexity laid upon the complexity of the network of computers.

An exertion submitted onto the network dynamically binds to all relevant and currently available service providers in the object-oriented distributed system. The providers that dynamically participate in this invocation are collectively called an *exertion federation*. This federation is also called a *virtual metacomputer* since federating services are located on multiple physical compute nodes held together by the EO infrastructure so that, to the requestor submitting the exertion, it looks and acts like a single computer.

The SORCER environment provides the means to create interactive EO programs [24] and execute them using the SORCER runtime infrastructure presented in Section 3. Exertions can be created using interactive user agents downloaded on-the-fly from service providers. Using these interfaces, the user can create, execute, and monitor the execution of exertions within the EO metacomputer. The exertions can be persisted for later reuse, allowing the user to quickly create new applications or EO programs on-the-fly in terms of existing, usually persisted for reuse exertions.

SORCER is based on the evolution of concepts and lessons learned in the FIPER project [5, 23], a $21.5 million program founded by NIST (National Institute of Standards and Technology). Academic research on EO programming has been established at the SORCER Laboratory, TTU, [27] where twenty-five SORCER related research studies have been investigated so far [28]. The most recent version of EO programming used in SORCER is described in this paper.

The paper is organized as follows. Section 2 provides a brief description of two service-oriented architectures with a related discussion of distribution transparency; Section 3 describes the SORCER methodology; Section 4 presents EO programming and the related FMI framework; 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 (left). 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. Requestors and providers can use discovery and join protocols to locate registries and then publish or acquire services on the network.

We can distinguish the *service object-oriented architecture* (SOOA), where providers are network objects accepting remote invocations, from the *service protocol oriented architecture* (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.

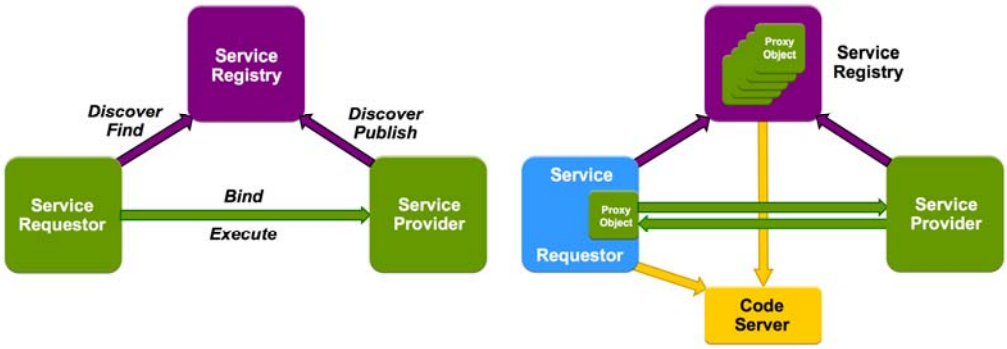

Figure 1. SOA versus SOOA

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

Web services and OGSA services cannot change the communication protocol between requestors and providers while the SOOA approach is protocol neutral [32]. 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) can grant access to local and remote resources; they can also communicate with multiple providers on the network regardless of who originally registered

the proxy. Thus, separate providers on the network can implement different parts of the smart proxy interface. Communication protocols may also vary, and a single smart proxy can also talk over multiple protocols including efficient application-specific protocols.

SPOA and SOOA differ in their method of discovering the service registry (see Figure 1). 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 OGSA services, a UDDI registry is sometimes even omitted when WSDL descriptions are shared via files; in SOOA, lookup services are mandatory due to the dynamic nature of objects identified by service types. 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.

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

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

Do you remember the eight fallacies [4] of network computing? We cannot just take an object-oriented program developed without distribution in mind and make it a distributed system ignoring the unpredictable network behavior. Most RPC systems, except Jini, hide the network behavior and try to transform local communication into remote communication by creating distribution transparency based on a local assumption of what the network might be. However every single distributed object cannot do that in a uniform way as the network is a heterogeneous distributed system and cannot be represented completely within a single entity.

The network is dynamic, cannot be constant, and introduces latency for remote invocations. Network latency also depends on potential failure handling and recovery mechanisms so we cannot assume that a local invocation is similar to remote invocation. Thus complete 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 type of distributed object or network wrapper; it is a metasystemic issue in object-oriented distributed programming.

EO programming is introduced to handle the metasystemic distribution in SORCER by using indirect remote method invocation with no service provider directly specified in the network request called *service exertion*. Specific infrastructure services support EO programming combined with the FMI framework presented in this paper. That infrastructure defines SORCER's distributed object encapsulation, modularity, extensibility, and reuse of service-oriented components consistent with the metacomputing granularity, behavioral transfer, and

configuration setup with dependency injection—the key features of object-oriented distributed programming that are usually missing in SPOA programming environments.

3. FEDERATED SERVICE OBJECT-ORIENTED COMPUTING ENVIRONMENT: 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. 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, SORCER is focused on EO programming and the execution environment for exertions. SORCER uses Jini discovery/join protocols to implement its *exertion-oriented architecture* (EOA) using FMI, but hides all low-level programming details of the Jini programming model [3].

In EOA, a service provider is an object that accepts remote messages from service requestors to execute a service collaboration. These messages, called *service exertions*, describe service data, operations and provider's control strategy. An *exertion task* (or simply a *task*) is an elementary service request, a kind of elementary remote instruction (elementary network statement) executed by a single service provider or a small-scale federation. A composite exertion called an *exertion job* (or simply a *job*) is defined hierarchically in terms of tasks and other jobs, a kind of network procedure (block of network statements) executed by a large-scale federation. The executing (active) exertion is dynamically bound to all required and currently available service providers on the network. This collection of providers identified in runtime is called an *exertion federation*. The federation provides for the implementation of the executing exertion. When the federation is formed, then each exertion's operation has its corresponding method (code) on the network available. Thus, the network *exerts* the request with the help of the service federation formed in runtime. In other words we send the request onto the network implicitly, not to a particular network object explicitly.

While this sounds similar to the object-oriented paradigm, it really is not. In the object-oriented paradigm, the object space is a program itself; here the exertion federation is the *execution environment* for the exertion, and the exertion is the *object-oriented* program—*specification* of service collaboration. This changes the programming paradigm completely. In the former case the object space is hosted by a single computer, but in the latter case the parent and its component exertions along with related service providers are hosted by the network of computers.

The overlay network of service providers is called the *service grid* and the exertion federation is in fact 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 consists of required metainstructions, service-oriented control strategy, and a service context representing the metaprogram data. The *service context* describes the data that tasks and jobs work on. EO programs can be created interactively using zero-install service user interfaces [31] and allow for transparent monitoring and debugging [24, 25] their execution within the grid. Please note that these metacomputing concepts are defined differently in traditional grid

computing where a job is just an executing process for a submitted executable code with no federation being formed.

Each service provider offers services to other service peers [8] on the object-oriented overlay network. These services are exposed *indirectly* by methods in well-known public interfaces and considered as operations that can be specified in exertions. Thus a service is essentially an interface (service) type (in EOA a Java interface) that has a service provider implementing it. Indirectly means here, that you cannot invoke any operation defined in provider's interface directly. These operations can be specified in a requestor's exertion only, and the exertion can be passed on to any service provider via the top-level `Servicer` interface required by all service providers. Thus all service providers in EOA implement the `service(Exertion, Transaction):Exertion` operation of the `Servicer` interface. When the provider accepts its received exertion, then the exertion's operations can be invoked by the provider itself, if the requestor is authorized to do so.

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. In EOA requestors do not have to lookup for any network provider at all, they can submit an exertion, onto the network by calling `Exertion.exert(Transaction):Exertion` on the underlying exertion. The `exert` operation will create a required federations that will execute the collaboration specified by the exertion and return the resulting exertion back the requestor. Since exertion encapsulates everything needed (data, operations, and control strategy) for the program execution, the results of the execution can be found in the returned exertion's service context.

Domain specific providers within the federation, or task peers (*taskers*), execute service tasks. Two types of *rendezvous peers*, `Jobber` and `Spacer`, coordinate execution of job exertions. Providers of the `Tasker`, `Jobber`, and `Spacer` type are core infrastructure services of the metacompute operating system in SORCER (see Figure 1). In view of the P2P architecture defined by the `Servicer` interface, a job can be sent to any service provider (peer). A peer that is not of a `Jobber` or `Spacer` type is responsible for forwarding the job to one of available rendezvous peers in the SORCER environment and returning results back to the initiating requestor.

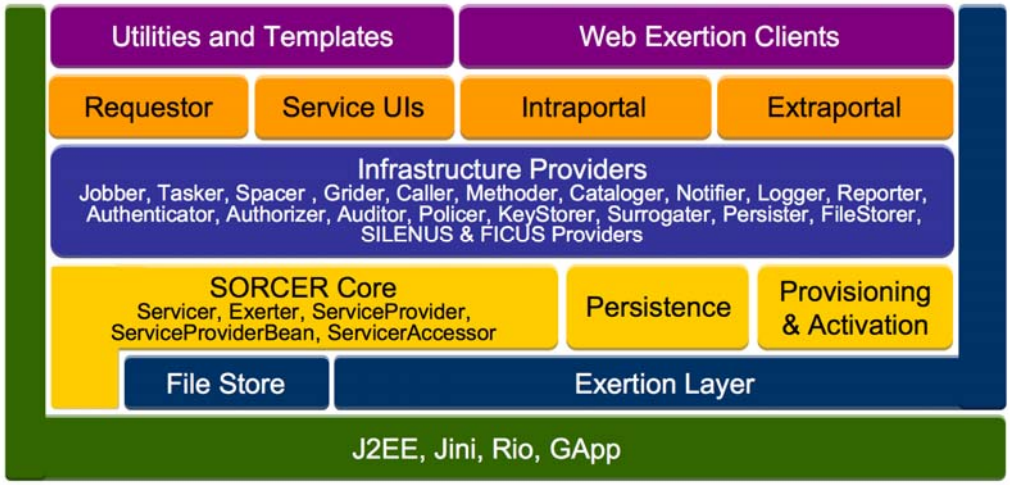

Figure 2. The SORCER layered functional architecture

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 [30] and Globus [29]. Here, instead of exertions being executed by services providing own business logic, the business logic comes from the service requestor's executable codes that seek compute resources on the network.

Traditional grid-based services in the SORCER environment include `Grider` services collaborating with `Jobber` and `Spacer` services for traditional grid job submission, and `Caller` and `Methoder` services for task execution [14]. `Callers` execute legacy codes via a system call as described in the standardized service context of submitted task. `Methoders` can download required Java code (task method) from requestors to process any submitted context accordingly with the code downloaded. In either case, the business logic comes from the requestors; it is an executable code invoked by `Callers` with the standardized `Caller`'s service context or mobile Java code executed by `Methoders` with a relevant service context provided by the requestor. A SORCER layered functional architecture with a collection of infrastructure services is presented in Figure 2.

4. EXERTION-ORIENTED PROGRAMMING

Each programming language provides a specific computing abstraction. Procedural languages are abstractions of assembly languages. Object-oriented languages abstract entities in the problem (application) domain that refer to "objects" communicating via message passing as their representation in the corresponding solution domain, e.g., SORCER objects. EO programming is a form of distributed programming that allows us to describe the distributed problem in terms of the intrinsic unpredictable network domain instead of in terms of distributed objects that hide the notion of the network domain.

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 distributed object (compute node) in that it has a network state, network behavior, and network types(s). Service providers act also as *network peers;* they implement the same top-level interface; they are replicated and dynamically provisioned for reliability to compensate for network failures [20]; they can be found dynamically at 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 provider's *control strategy*. 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 abstraction of EO programming, let's look into a simple analogy to Unix shell scripts execution and then in detail at how EOA is defined.

Let's first look at the EO approach to see how it works. EO programs consist of *exertion* objects called tasks and jobs. An exertion *task* corresponds to an individual network request to be executed on a service provider. An exertion *job* consists of a structured collection of tasks

and other jobs. The data upon which to execute a task or job is called a s*ervice context*. Tasks are analogous to executing a single program or command on a computer, and the service context would be the input and output streams that the program or command uses. A job is analogous to a batch script that can contain various commands and calls to other scripts. Pipelining Unix commands allows us to perform complex tasks without writing complex programs. As an example, consider a script `sort.sh` connecting simple processes in a pipeline as follows:

```
cat hello.txt | sort | uniq > bye.txt
```

The script is similar to an exertion job in that it consists of individual tasks that are organized in a particular fashion. Also, other scripts can call the script `sort.sh`. An exertion job can consist of tasks and other jobs, much like a script can contain calls to commands and other scripts.

Each of the individual commands, such as `cat`, `sort`, and `uniq`, would be analogous to a task. Each task works with a particular service context. The input context for the `cat` "task" would be the file `hello.txt`, and the "task" would return an output context consisting of the contents of `hello.txt`. This output context can then be used as the input context for another task, namely the `sort` command. Again the output context for `sort` could be used as the input context for the `uniq` task, which would in turn give an output service context in the form of `bye.txt`.

To further clarify what an exertion is, an exertion consists mainly of three parts: a set of *service signatures*, which is a description of operations in a collaboration, the associated *service context* upon which to execute the exertion, and control strategy (default provided) that defines how signatures are applied in the collaboration. A *service signature* specifies at least the provider's interface that the service requestor would like to use and a selected operation to run within that interface. There are four types of signatures that can be used for an exertion: `PREPROCESS`, `PROCESS`, `POSTPROCESS`, and `APPEND`. An exertion must have one and only one `PROCESS` signature that specifies what the exertion should do and who works on it. An exertion can optionally have multiple `PREPROCESS`, `POSTPROCESS`, and `APPEND` signatures that are primarily used for formatting the data within the associated service context. A *service context* consists of several data nodes used for either input, output, or both. A task may work with only a single service context, while a job may work with multiple service contexts since it can contain multiple tasks. The programmer can define a control strategy as needed for the underlying exertion by choosing relevant exertion types (see Section 4.4 and 4.5) and configuring attributes of service signatures accordingly (see Section 4.2 for details).

Here is the basic structure of the EO program that is analogous to the `sort.sh` script.

```
1.  // Create service signatures
2.  Signature catSignature, sortSignature, uniqSiganture;
3.  catSignature = new ServiceSignature("Reader", "cat");
4.  sortSignature = new ServiceSignature("Sorter", "sort");
5.  uniqSiganture = new ServiceSignature("Filter", "uniq");
6.
7. // Create component exertions
8.  Task catTask, sortTask, uniqTask;
9.  catTask = new ServiceTask("t-cat", catSignature);
10. sortTask = new ServiceTask("t-sort", sortSignature);
11. uniqTask = new ServiceTask("t-uniq", uniqSiganture);
12.
13. // Create top-level exertion
14. Job sortJob = new SevriceJob("Sort job");
```

```
15. sortJob.addExertion(catTask);
16. sortJob.addExertion(sortTask);
17. sortJob.addExertion(uniqTask);
18.
19. // Create service contexts
20. Context catContext, sortContext, uniqContext;
21. catContext = new ServiceContext("c-cat");
22. sortContext = new ServiceContext("c-sort");
23. uniqContext = new ServiceContext("c-uniq");
24.
25. catContext.putInValue("/text/in/URL", "http://host/hello.txt");
26. catContext.putOutValue("/text/out/contents", null);
27.
28. sortContext.putInValue("/text/in/contents", null);
29. sortContext.putOutValue("/text/out/sorted", null);
30.
31. uniqContext.putInValue("/text/in/sorted", null);
32. uniqContext.putOutValue("/text/out/URL", "http://host/bye.txt");
33.
34. // Map context output to input parameters (paths)
35. catContext.map("/text/out/contents",
36.           "/text/in/contents", sortContext);
37. sortContext.map("/text/out/sorted",
38.           "/text/in/sorted", uniqContext);
39.
40. catTask.setContext(catContext);
41. sortTask.setContext(sortContext);
42. uniqTask.setContext(uniqContext);
43.
44. // Activate the top-level job exertion
45. sortJob.exert(null);
```

In the above EO program we create three signatures (lines 2-5), each signature is defined by the interface name and the operation name that we want to run by any remote object implementing the interface. We use the three signatures to create three tasks (lines 8-11) and by line 12, we have three separate commands `cat`, `sort`, and `uniq` to be used in the `sort.sh` script. The three tasks are combined into the job by analogy to piping Unix commands in the `sort.sh` script. Thus, by line 18, we have added these commands to `sort.sh` script, but have not provided input/output parameters nor piped them together:

```
as is:       cat             sort   uniq
to be:       cat hello.txt | sort | uniq > bye.txt
```

Lines 20-32 create and define three service contexts for our three tasks. By line 32, we have specified some input and output parameters, but still no piping:

```
as is:       cat hello.txt   sort   uniq   bye.txt
to be:             cat hello.txt | sort | uniq > bye.txt
```

Lines 35-38 define mapping of context output to the related context input parameters. The parameters are context paths from a source context to a target context. The target context is the last parameter in the `map` operation. By line 43, we have piping setup and by the analogy our `sort.sh` script is complete now:

```
as is:       cat hello.txt | sort | uniq > bye.txt
```

On line 45, we execute the script. If we use the Tenex C shell (`tcsh`), invoking the script is equivalent to: `tcsh sort.sh`, i.e., passing the script `sort.sh` on to `tcsh`. Similarly, to

invoke the exertion `sortJob`, we call `sortJob.exert()`. Thus, the exertion is the program 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 distributed 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 encapsulated in the exertion that creates the required federation on-the-fly. The federation provides the implementation (metacomputer instructions) as specified in signatures of the EO program before the exertion runs on the network.

The `sortJob` program described above can be rewritten with just one exertion task only instead of exertion job as follow:

```
1.  // Create service signatures
2.  Signature catSignature, sortSignature, uniqSiganture;
3.  catSignature = new ServiceSignature("Reader", "cat",
4.           Type.PREPROCESS);
5.  sortSignature = new ServiceSignature("Sorter", "sort",
6.           Type.PROCESS);
7.  uniqSiganture = new ServiceSignature("Filter", "uniq",
8.           Type.POSTPROCESS);
9.
10. // Create an exertion task
11. Task sortTask;
12. sortTask = new ServiceTask("task-sort", sortSignature);
13. sortTask.addSignature(catSignature);
14. sortTask.addSignature(sortSignature);
15. sortTask.addSignature(uniqSiganture);
16.
17. // Create a service context
18. Context taskContext;
19. taskContext = new ServiceContext("c-sort");
20.
21. taskContext.putInValue("/text/in/URL", "http:// host/hello.txt");
22. taskContext.putOutValue("/text/out/contents", null);
23.
24. taskContext.putInValue("/text/in/contents", null);
25. taskContext.putOutValue("/text/out/sorted", null);
26.
27. taskContext.putInValue("/text/in/sorted", null);
28. taskContext.putOutValue("/text/out/URL", "http:// host/bye.txt");
29.
30. // Map context output to input parameters (paths)
31. taskContext.map("/text/out/contents", "/text/in/contents",
32.           taskContext);
33. taskContext.map("/text/out/sorted", "/text/in/sorted",
34.           taskContext);
35. sortTask.setContext(taskContext);
36.
37. // Activate the task exertion
38. sortTask.exert(null);
```

In this version of the `sort.sh` analogy—`taskSort`, we create three signatures (lines 2-8), but in this case three signature types are assigned, so we can batch them into a single task (lines 12-15). In the `jobSort` version all signatures are of the default `PROCESS` type and each

task is created with its own context. Here we create one common `taskContext` (lines 18-35) that is shared by all signature operations. Finally, on line 38, we execute the exertion task `sortTask`.

The major difference between the two EO programs `jobSort` and `taskSort` is in the exertion execution. The execution of `jobSort` is in fact coordinated by a `Jobber`, but the execution of the `taskSort` is coordinated by the service provider implementing the `Sorter` interface that binds to the `PROCESS` signature `sortSignature`. If the provider implementing the `Sorter` interface, implements two other interfaces `Reader` and `Filter`, then the execution of `taskSort` is more efficient as all three operations can be executed by the same provider with no need of network communication between a `Jobber` and collaborating providers in the `jobSort` federation.

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 implementation (method) for that message. 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 (identifier) of the receiving object, the name (selector) of operation to be invoked, and its parameters. 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 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*. An exertion encapsulates multiple *service signatures* that define operations, a s*ervice context* that defines data, and a *control strategy* that defines how signature operations flow during exertion execution. Different types of control flow exertions (Section 4.4) can be used to define collaboration control strategies that can also be configured with signature flow type and access type attributes. Two basic exertion categories are distinguished: elementary and composite exertion called *exertion task* and *exertion job*, respectively. Task and job control strategies are described in Section 4.5.

An exertion can be activated by calling exertion's `exert` operation: `Exertion.exert(Transaction):Exertion`, where a parameter of the `Transaction` type is required when a transactional semantics is needed for all participating nested exertions within the parent one. Thus, EO programming allows us to submit an exertion onto the network and to perform executions of exertion's signatures on various service providers indirectly, but where does the service-to-service communication come into play? How do these services communicate with one another if they are all different? Top-level communication between services, or the sending of service requests (exertions), is done through the use of the generic `Servicer` interface and the operation `service` that all SORCER services are required to provide—`Servicer.service(Exertion, Transaction):Exertion`. This top-level service operation takes an exertion as an argument and gives back an exertion as the return value. In Section 4.7 we describe how this operation is used in the FMI framework.

So why are exertions used rather than directly calling on a provider's method and passing service contexts? There are two basic answers to this. First, passing exertions helps to aid with the network-centric messaging. A service requestor can send an exertion out onto the net-

work—`Exertion.exert()`—and any service provider can pick it up. The provider can then look at the interface and operation requested within the exertion, and if it doesn't implement the desired interface or provide the desired method, it can continue forwarding it to another service provider who can service it. Second, passing exertions helps with fault detection and recovery. Each exertion has its own completion state associated with it to specify if it has yet to run, has already completed, or has failed. Since full exertions are both passed and returned, the user can view the failed exertion to see what method was being called as well as what was used in the service context input nodes that may have caused the problem. Since exertions provide all the information needed to execute an exertion including its control strategy, a user would be able to pause a job between tasks, analyze it and make needed updates. To figure out where to resume an exertion, the service provider would simply have to look at the exertion's completion states and resume the first component one that wasn't completed yet.

4.2 Service Signatures

An activated 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. An exertion signature is compared to the operations defined in the service provider's interface along with a set of signature attributes describing the provider, and if a match is found, the appropriate operation can be invoked on the remote provider. In federated method invocation (FMI) signatures specify indirect invocations of provider methods via the `service` operation of the top-level `Servicer` interface as described in Section 4.1.
A service `Signature` is defined by:

- signature name—a custom name
- service type name—a service name corresponding to the provider's type
- service flow type—`FlowType`: `SEQUENTIAL` (default), `PARALLEL`, `CONCURRENT`
- selector of the service operation—an operation name defined in the service type
- operation type—`Signature.Type`: `PROCESS` (default), `PREPROCESS`, `POSTPROCESS`, `APPEND`
- service access type—`Signature.Access`: `PUSH` (default) direct binding to service providers, or `PULL` using a shared exertion space via the `Spacer` service
- priority—integer value used by exertion's control strategy
- execution time flag—if true, the execution time is returned in the service context
- notifyees—list of email addresses to notify upon exertion completion
- service attributes—required requestor's attributes matching provider's registration attributes

An exertion can comprise of a collection of `PREPRROCESS`, `POSTPROCESS`, and `APPEND` signatures, but with only one `PROCESS` signature. The `PROCESS` signature defines the binding provider for the exertion. An `APPEND` signature defines the context received from the provider specified by this signature. The received context is then appended in runtime to the context later processed by `PREPROCESS`, `PROCESS`, and `POSTPROCESS` operations of the exertion. Appending a service context allows a requestor to use actual network data in runtime not available to the requestor when the exertion is submitted.

Different languages have different interpretations as to what constitutes and operation signature, For example, in C+ and Java the return type is ignored. In FMI the parameters and

return type are all of the `Context` type. Using the UML advanced operation syntax, the exertion operation (prefixing it with the <<service>> stereotype and postfixing with tagged values), can be defined as follow:

```
<<service>> operation-name ( operation-context : Context ) : Context { interface = service-type-name, type = operation-type, access = access-type, flow = flow-type, priority = integer, timing = boolean, notfiees = notfiees-list, attributes = registration-attribute-list }
```

4.3 Service Contexts

A *service context*, or simply a *context*, defined by the `Context` interface, is a data structure that describes service provider ontology along with related data. A provider ‘s ontology is controlled by the provider vocabulary that describes data 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 as it specifies how the context data is interpreted and used by the provider. In service context, *attributes* and their *values* are used as atomic conceptual primitives, and *complements* are used as composite ones. A *complement* is an attribute sequence (path) with a value at the last position. A *context property* consists of a *subject complement* and a set of defining *context complements*. The context property usually corresponds to a simple sentence of natural language (a subject with multiple complements).

A service context is a tree-like structure described conceptually by 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 service context, let’s consider the following example (graphically depicted in Figure 3 where the subject *SORCER* is indicated in green color and the *person* association in red):

/laboratory/name = SORCER: /university=TTU;
/university/department/name=CS;
/university/department/room;
number=20B;

```
phone/number=806-742-1194;
phone/ext=237;
/director<person | Mike | W | Sobolewski>/email=sobol@cs.ttu.edu;
```

where absolute and relative paths are used, and the relation `person` is defined as follows: `person | firstname | initial | lastname`.

A context leaf node, or *data node* is where the actual data resides. All absolute context paths define an application domain namespace. The context namespace with data nodes appended to its context paths is called a *context model*, or simply a *context*. A context path is a hierarchical name for a data item in a leaf node. Note that a service context can be represented as an XML document—what has been done in SORCER for interoperability—but the power of the `Context` type 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 by providers as needed to run complex iterative programs, e.g., nonlinear multidisciplinary optimization [15].

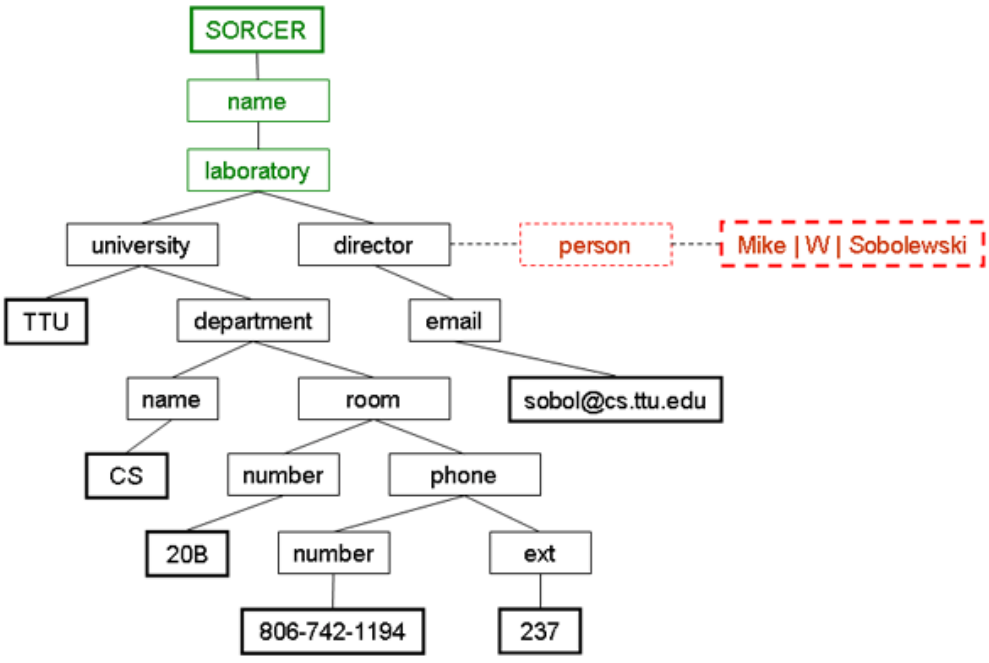

Figure 3. An example of a service context

4.4 Exertion Types

A `Task` instance specifies an elementary step of EO program. It is an analog of a statement in conventional programming languages (see the examples in Section 4). Thus, it is a minimal

unit of structuring in EO programming. If the provider binds to a `Task`, it has a method for the task's `PROCESS` signature. Other signatures associated with the `Task` exertion provide for preprocessing and postprocessing by the same provider or its collaborating providers. An `APPEND` signature defines the context received from the provider specified by this signature. The received context is then appended in runtime to the task context later processed by `PREPROCESS`, `PROCESS`, and `POSTPROCESS` operations of the task. Appending a service context allows a requestor to use actual network data in runtime not available to the requestor when the task is submitted. A `Task` is the single means of passing control to an application service provider in EOA. Note that a task can specify a batch of operations that operate on the same service context—a `Task` shared execution state. All operations of the `Task`, that are defined by its signatures, can be executed by the same provider or a group of federating providers coordinated by the receiving provider.

A `Job` instance specifies a "block" of task and other jobs. It is the analog of a procedure in conventional programming languages. In EO programming it is a composite of exertions that makeup the network collaboration. A `Job` can reflect a workflow with branching and looping by concatenating control flow exertions.

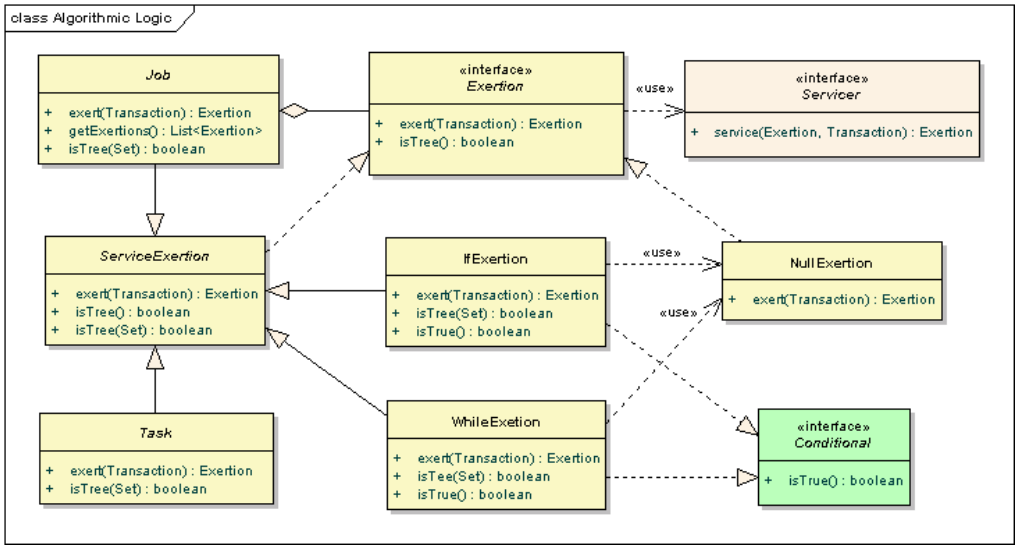

Figure 4. Exertion types including control flow exertions that allow for algorithmic logic in EO programming

The following control exertion types define algorithmic logic of EO programming: `ServiceTask`, `ServiceJob`, `IfExertion`, `WhileExertion`, `ForExertion`, `DoExertionThrowExertion`, `TryExertion`, `BreakExertion`, `ContinueExertion`. Currently implemented control `Exertion` types in SORCER are depicted in Figure 4.

4.5 Exertion Control Strategies

In Section 4.1 and 4.2 top-level exertion messaging and service signatures were described. This section will present how they are used, at the task level and job level, to execute an EO program. Before we delve into a task and job execution strategy, let's look at three related infrastructure providers identified by the following interfaces: `Jobber`, `Spacer`, and `Cataloger`.

To begin processing a job, a service requestor must exert the job that finds its way dynamically to a `Jobber` service using its implementation of `Exertion.exert(Transaction):Exertion`. The `Jobber` is responsible for coordinating the execution of the job, much like a command shell coordinates the execution of a batch script (see the programming examples in Section 4). The `Jobber` acts as a service broker by calling upon the proper service providers to execute the component exertions within the given job. The `Jobber` can dispatch nested service requests either explicitly, where the jobber finds a proper provider by way of a `Cataloger` service or falling back to the Jini lookup service, or it can be dispatched implicitly to a shared exertion space through the use of a `Spacer` service.

SORCER extends the discovery and registration capabilities of the service-oriented architecture through the use of a service called the `Cataloger` service. A cataloger service looks through all the Jini lookup services that it is aware of and requests all the SORCER service registration it can get. The cataloger organizes these registrations that include service proxies, into groups of the same type. Whenever a service requestor needs a certain service, it can go to a cataloger instead of a lookup service to find what it needs. The cataloger will distribute registrations for the same service in a round-robin fashion to help balance the load between service providers of the same service type.

SORCER also extends task/job execution abilities through the use of a `Spacer` service. The spacer service can drop a task into a shared object space, provided by the Jini JavaSpace service [7], in which several providers can retrieve relevant exertions from the object space, execute them, and return the results back to the object space.

As defined before, an exertion is associated with a collection of signatures. There is only one `PROCESS` signature in this collection and multiple instances of `APPEND`, `PREPROCESS`, and `POSTPROCESS` signatures. The `PROCESS` signature is responsible for binding to the service provider that executes the exertion. The exertion activated by a service requestor (`Exertion.exert(Transaction):Exertion`) can be submitted directly or indirectly to the matching service provider. In the direct approach, when signature's access type is `PUSH`, the exertion's `ServicerAccessor` (see Figure 5) finds the matching service provider against the service type and attributes of the `PROCESS` signature and submits the exertion to the matching provider. Alternatively, when signature's access type is `PULL`, a `ServiceAccessor` can use a `Spacer` provider that simply drops the exertion into the shared exertion space to be pulled by matching providers. Each service provider looks continuously into the space for exertions that match a provider's interfaces and attributes. Each service provider that picks up a matched exertion from the exertion space returns the exertion being executed back into the space, then the requestor picks up the executed exertion from the space. The exertion space provides a kind of automatic load balancing—the fastest available service provider gets an exertion from the space and joins the federation.

When a receiving service provider gets a task (directly or indirectly) then the task signatures are executed in the following order:

1. First, all `APPEND` signatures are processed by the receiving provider in the order specified in the task. The order of signatures is defined by signature priorities, if the task's flow type is `SEQUENTIAL`, otherwise they are dispatch in parallel. In the result the task's service context is appended with dynamic data delivered from context providers specified by these append signatures. Obtained complementary shared context data is managed by the receiving provider according to the remote Observer/Observable design pattern [10].
2. Second, all `PREPROCESS` signatures are executed in the order specified in the task. The order is as defined in 1). In the result the task context is ready for applying its `PROCESS` method.
3. Third, the `PROCESS` signature is executed and results are captured in the task context including any exceptions and errors.
4. Forth, all `POSTPROCESS` signatures are executed in the order specified in the task. The order is as defined in 1). Finally the resulting task with the processed context is returned to the requestor.

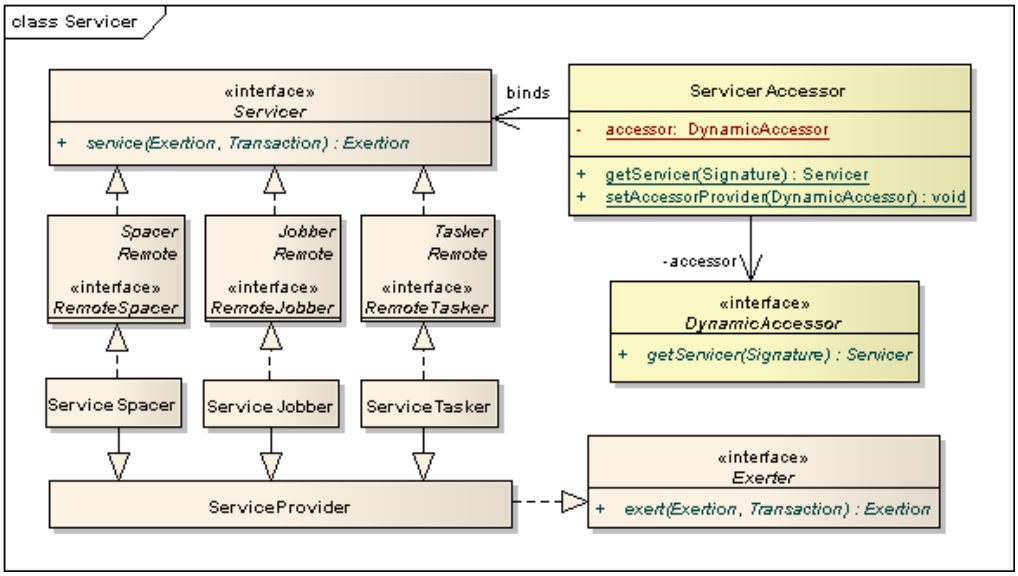

Figure 5. The primary types of SORCER providers: `Tasker`, `Jobber`, and `Spacer` with supporting `ServicerAccessor`

The default job's `PROCESS` signature defines a runtime binding to a `Jobber` provider. Two major parameters: job `PROCESS` signature's access type and its flow type determine the top-level control strategy. Additionally, an implicit job's service context, called a *control context*, defines job's execution preferences. When a `Jobber` gets an exertion job then a relevant dispatcher is assigned to a `Jobber` by a dispatcher factory that takes into account job's access type, flow type, and a control context configuration. In the SORCER environment there are twelve types of dispatchers that implement different types of control strategies. The assigned

dispatcher manages the execution of the job's component exertions either sequentially or in parallel (depending on the value of flow type), and accessing collaborating providers either directly or indirectly (depending on the value of access type). The top-level control strategy implements a master/slave computing model with sequential or parallel execution of slave exertions with the master exertion executed as the last one, if any. In general, full algorithmic logic operations: concatenation, branching, and looping are supported. A job's workflow can be defined in terms of flow control exertion types defined in Section 4.4. The job signature's access type specifies the way a jobber accesses collaborating service providers: directly or indirectly. While the `Spacer` provider is used for indirect access, the `Cataloger` service is usually used by a `Jobber` to find directly needed service providers.

4.6 Service-to-Service (S2S) Infrastructure

Exertion tasks are usually executed by service providers of the `Tasker` type and exertion jobs by rendezvous providers of `Jobber` or `Spacer` type. While a `Tasker` manages a single service context for the received task, a rendezvous provider manages a shared context (shared execution state) for the job federation and provides substitutions for input parameters in service contexts of component exertions. Either one, a `Tasker` or rendezvous provider creates a federation of required service providers in runtime, but federations managed by rendezvous providers are usually larger in size than those managed by `Taskers`. All SORCER service providers implement the top-level `Servicer` interface. A peer of the `Servicer` type 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.

Thus, each `Servicer` can initiate a federation created in response to `Servicer.service(Exertion, Transaction)`. `Servicers` come together to form a federation participating in collaboration for the activated 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 required interfaces. Despite the fact that every `Servicer` can accept any exertion, `Servicers` have well defined roles in EOA:

a) `Taskers` – process service tasks
b) `Jobbers` – process service jobs
c) `Spacers` – process tasks and jobs via exertion space for space-based computing [7]
d) `Contexters` – provide service contexts for `APPEND Signatures`
e) `FileStorers` – provide access to federated file system providers [1, 25]
f) `Catalogers` – service registries
g) `Persisters` – persist service contexts, tasks, and jobs to be reused for interactive exertion-based programming
h) `Relayers` – gateway providers, transform exertions to native representation, for example integration with Web services and JXTA [12]
i) `Autenticators`, `Authorizers`, `Policers`, `KeyStorers` – provide support for service security
j) `Auditors`, `Reporters`, `Loggers` – support for accountability, reporting and logging

k) `ServiceProviderBeans` – to enable autonomic provisioning with the Rio framework [20]
l) `Griders`, `Callers`, `Methoders` – support traditional grid computing
m) Generic `ServiceTasker`, `ServiceJobber`, and `ServiceSpacer` implementations are used to configure domain-specific providers via dependency injection—configuration files for smart proxying and embedding business objects, called service beans, into service providers. Also, domain-specific providers can subclass either one and implement required domain-specific interfaces with operations returning a service context and taking a service context as its single parameter. These domain-specific interfaces and operations are used in task signatures.

4.7 FMI Triple Command Pattern

Polymorphism let us encapsulate a request 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 in a generic interface and defines various implementations of the interface. In Federated Method Invocation (FMI), the three interfaces are defined with the following three commands:

1. `Exertion.exert(Transaction):Exertion`—join the federation;
2. `Servicer.service(Exertion, Transaction):Exertion`—request a service in the federation from the top-level `Servicer` obtained for the activated exertion;
3. `Exerter.exert(Exertion, Transaction):Exertion`—execute the argument exertion by the target provider in the federation.

These three commands define the *Triple Command* pattern that makes EO programming possible via various implementations of the three interfaces: `Exertion, Servicer, and Exerter`.

The FMI approach allows for:

- the P2P environment via the `Servicer` interface,
- extensive modularization of programming P2P collaborations by the `Exertion` type,
- the execution of exertions by providers of the `Exerter` type, and
- vast common synergistic extensibility from the triple design pattern.

Thus, requestors can exert simple (tasks) and structured metaprograms (jobs with control exertions) with or without transactional semantics as defined in 1) above.

The Triple Command pattern in SORCER works as follows:

1. An exertion is invoked by calling `Exertion.exert(Transaction)`. The `Exertion.exert` operation implemented in `ServiceExertion` uses `ServicerAccessor` to locate in runtime the provider matching the exertion's `PROCESS` signature.
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`, `ServiceJobber`, or `ServiceSpacer` depending on the type of the exertion (`Task` or `Job`) and its control strategy. Then the provider by reflection calls the method specified in the exertion

PROCES signature (interface and selector). All application domain methods that are used in exertion signatures have the same signature: a single Context type parameter and a Context type return vale. Thus a custom (application) interface looks like a common Java RMI interface with the above simplification on the common signature for all domain-specific operations defined in the provider remote interfaces.

In the FMI approach, a requestor can create an Exertion, composed from any hierarchically nested Exertions, with required service contexts received from providers. The provider's object proxy, service namespace, and registration attributes are network-centric; all of them are part of the provider's registration so they can be accessed via Cataloger or lookup services. Thus, any requestor on the network, e.g., service browsers [11] or custom service UI [31] user agents. In SORCER, using these zero-install service UIs, the user can define data in downloaded context provided by the provider and create related tasks/job to be executed on the virtual metacomputer.

Individual service providers either Taskers or rendezvous peers, implement their own exert(Exertion, Transaction) method according to their service semantics and control strategy In SORCER taskers, jobbers, and spacers are implemented by ServiceTasker, ServiceJobber, and ServiceSpacer classes respectively (see Figure 5). A SORCER specific-domain provider can be a subclass of ServiceTasker, ServiceJobber, or ServiceSpacer. Alternatively, one of these three providers can be set up as an application provider by dependency injection—using the Jini configuration methodology. Twelve proxying methods have been developed in SORCER to configure off-the-shelf ServiceTasker, ServiceJobber, or ServiceSpacer. In general, many different types of taskers, jobbers, and spacers can be used in SORCER at the same time and exertions via their signatures will make appropriate runtime choices as to what virtual collaboration to run.

Invoking an exertion, let's say program, is similar to invoking an executable program program.exe at the command prompt. If we use the Tenex C shell (tcsh), invoking the program is equivalent to: tcsh program.exe, i.e., passing the executable program.exe to tcsh. Similarly, to invoke a metaprogram using FMI, in this case the exertion program, we call program.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 in the context of FMI 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. In Figure 6 a cloud represents a service grid wile the metacomputer is a subset of providers that federate for the job shown below the cloud.

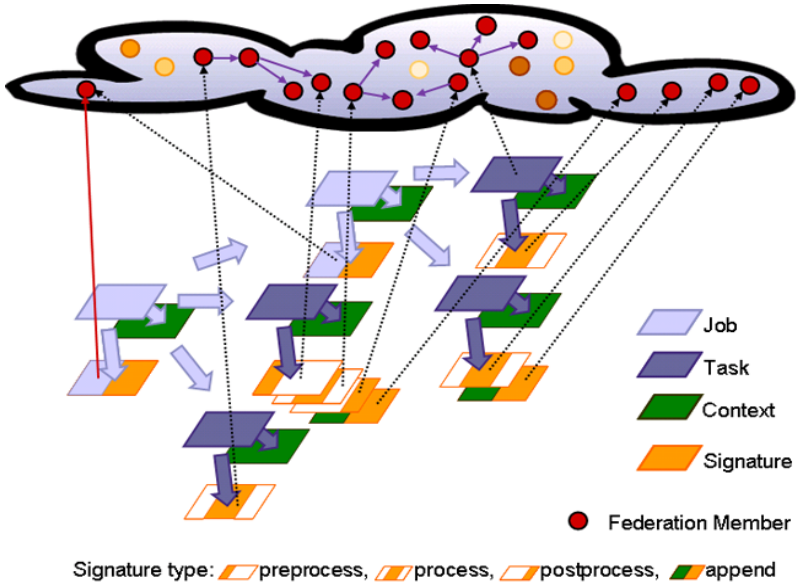

Figure 6. A job federation. The solid line (the first from the left) indicates the originating FMI invocation: `Exertion.exert(Transaction)`. The top-level job with component exertions is depicted below the service grid (a cloud). Late bindings of all signatures are indicated by dashed lines that define the job's federation (metacomputer).

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, `Exertion`s are not distributed objects, as do not implement any remote interfaces; they are local objects representing network requests only. `Servicers` are distributed objects, but `Servicers` collaborate with other infrastructure providers addressing different aspects of networking. The network intrinsic unpredictable behavior is addressed by the SORCER object-oriented distributed infrastructure: `Taskers`, `Jobbers`, `Spacers`, `Catalogers`, `FileStorers`, `Authenticators`, `Authorizers`, `KeyStorers`, `Policers`, etc. (see Figure 2). The FMI based infrastructure facilitates EO programming and concurrent metaprogram execution using the presented framework and allows for constructing large-scale reliable object-oriented distributed systems from unreliable distribute components—`Servicers`.

5. CONCLUSIONS

A distributed system is not just a collection of distributed objects—it is the network of dynamic objects that come and go. From the object-oriented point of view, the network of dynamic objects is the *problem domain* of object-oriented distributed system that requires relevant abstractions in the *solution space*—FMI. The exertion-based programming introduces the new abstraction of the solution space with *service providers* and *exertions* instead of object-

oriented conventional *objects* and *messages*. Exertions not only encapsulate operations, data, and control strategy, they encapsulate related federations of dynamic service providers as well.

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

The FMI framework allows for the P2P computing via the `Servicer` interface, extensive modularization of `Exertion`s and `Exerter`s, and extensibility from the Triple Command design pattern. The presented EO programming methodology has been successfully deployed and tested in multiple concurrent engineering and large-scale distributed applications [21, 8, 9, 14, 15, 24].

REFERENCES

1. Berger, M., and Sobolewski, M., SILENUS – A Federated Service-oriented Approach to Distributed File Systems, In Next Generation Concurrent Engineering, ISPE/Omnipress, 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 (2000)
4. Fallacies of Distributed Computing. Accessed on: January 15, 2008. Available at: http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing
5. FIPER: Federated Intelligent Product EnviRonmet. Available at: http://sorcer.cs.ttu.edu/fiper/fiper.html. Accessed on: January 15, 2008.
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: January 15, 2008.
12. JXTA. Available at: https://jxta.dev.java.net/. Accessed on: January 15, 2008.
13. Jini architecture specification, Version 2.1. Available at: http://www.sun.com/software/jini/specs/jini1.2html/jini-title.html. Accessed on: January 15, 2008(2001)
14. Khurana V., Berger M., Sobolewski M., A Federated Grid Env. with Replication Services. In Next Generation Concurrent Engineering, ISPE/Omnipress (2005)
15. 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)
16. Lapinski, M., Sobolewski, M., Managing Notifications in a Federated S2S Environment, International Journal of Concurrent Engineering: Research & Applications, Vol. 11, pp. 17-25 (2003)
17. McGovern J., Tyagi S., Stevens M.E., Mathew S., Java Web Services Architecture, Morgan Kaufmann (2003)
18. Package net.jini.jeri. Available at: http://java.sun.com/products/jini/2.1/doc/api/net/jini/jeri/package-summary.html. Accessed on: January 15, 2008.
19. Pitt E., McNiff K., java.rmi: The Remote Method Invocation Guide, Addison-Wesley Professional (2001)
20. Project Rio, A Dynamic Service Architecture for Distributed Applications. Available at: https://rio.dev.java.net/. Accessed on: January 15, 2008.
21. 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)
22. Ruh W.A., Herron T., Klinker P., IIOP Complete: Understanding CORBA and Middleware Interoperability, Addison-Wesley (1999)
23. Sobolewski M., Federated P2P services in CE Environments, Advances in Concurrent Engineering, A.A. Balkema Publishers, 2002, pp. 13-22 (2002)
24. 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)
25. Sobolewski, M., Soorianarayanan, S., Malladi-Venkata, R-K. 2003, Service-Oriented File Sharing, Proceedings of the IASTED Intl., Conference on Communications, Internet, and Information technology, pp. 633-639, ACTA Press (2003)
26. Soorianarayanan, S., Sobolewski, M., Monitoring Federated Services in CE, Concurrent Engineering: The Worldwide Engineering Grid, Tsinghua Press and Springer Verlag, pp. 89-95 (2004)
27. SORCER Research Group. Available at: http://sorcer.cs.ttu.edu/. Accessed on: January 15, 2008.
28. SORCER Research Topics. Available at: http://sorcer.cs.ttu.edu/theses/. Accessed on: January 15, 2008
29. Sotomayor B., Childers L., Globus® Toolkit 4: Programming Java Services, Morgan Kaufmann (2005)
30. Thain D., Tannenbaum T., Livny M. Condor and the Grid. In Fran Berman, Anthony J.G. Hey, and Geoffrey Fox, editors, Grid Computing: Making The Global Infrastructure a Reality. John Wiley (2003)
31. The Service UI Project. Available at: http://www.artima.com/jini/serviceui/index.html. Accessed on: January 15, 2008.
32. Waldo J., The End of Protocols, Available at: http://java.sun.com/developer/technicalArticles/jini/protocols.html. Accessed on: January 15, 2008.