Service domains

by Y.-S. Tan
V. Vellanki
J. Xing
B. Topol
G. Dudley

In this paper we introduce the concept of service domain, to be used as a major building block for implementing service-oriented architectures in large computing grids in which tens or hundreds of services are offered to customers. A service domain maps a collection of comparable or related services to a single logical service. We describe an architecture for service domains that uses a set of policy rules for managing the collection of services and that automatically dispatches the "best" service available to satisfy user requests. The architecture has built-in autonomic properties in that a service domain implementation monitors the events within its control and triggers adjustment actions in its member services, including recovery from service failure and handling of topology changes. We describe a reference implementation of the service domain architecture that is publicly available as a development toolkit, and we discuss its application in the implementation of a large grid now in progress.

Service-oriented computing and the associated service-oriented architecture[1] (SOA) are based on services as self-describing, open components that can be used to build distributed applications. A service is implemented by a software module that responds to queries and commands by performing a specified function.

There is a large degree of standardization in the operation of Web services.[2,3,4] Figure 1A illustrates the Web-services model that includes a service requestor, a service registry, and a service provider.

To assess the challenges involved in using Web services in environments in which tens or hundreds of services are offered to customers, consider exception handling. If a service bound to a client fails, there is no easy way for the client to switch to a comparable service. The client has to be able to query the service registry, locate an available service, establish a new binding, back out from the point of failure, and rerun the service request. It is also likely that there are several service instances to choose from and the client has to include the logic to select an appropriate service instance. The logic becomes complex and needs to be included in each client. The situation may become unmanageable if hundreds of services need to be handled this way.

Although recovery services and highly available platforms are often provided on the service provider end, clients are still required to handle exceptions. This is so because SOA enables dynamic mixing and matching of requestors and providers. If the service providers are independent businesses, the services provided may be viewed by the client as less reliable. In addition to the possibility of disturbances such as hacker attacks and network outages, additional factors such as business relationship and operation policy need to be considered as well. Availability will

©Copyright 2004 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

0018-8670/04/$5.00 © 2004 IBM

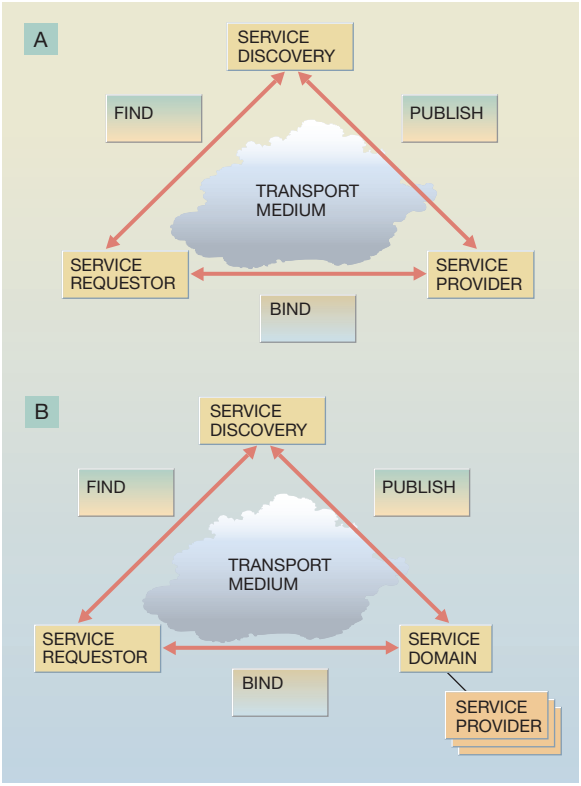
continue to be a critical consideration for stand-alone service implementations.

Consider the issue of integration, which is present in most modern IT systems. In today's dynamic business environment, Web services are ideal for implementing either internal or external business functions, such as inventory control and order processing. When businesses are forming partnerships and alliances, the number of suppliers can grow as businesses are expanding. The characteristics of suppliers can also become more diverse. For example in a catalog sales scenario, the suppliers could specialize in different classes and categories of merchandise, carry merchandise of different quality levels, and adhere to different volume-discount agreements. Suppliers may come and go, may fail on their deliveries, may have relationship constraints, and so on. There is no fast and easy way to manage these changes dynamically in real time. The main business processes can be implemented as workflows, but keeping the processes robust may depend on the performance of specific suppliers. The logic becomes unmanageable when tens and hundreds of suppliers need to be managed. This could be one reason why organizations tend to implement SOA with only a small number of suppliers. We believe that environments with hundreds of competing or collaborating suppliers will be a common phenomenon of the future. In order to achieve the full benefit of SOA in such environments, new techniques to overcome complexity are required.

A *service domain* (SD) maps a collection of comparable or related services to a single logical service. It is implemented as a *service domain node* consisting of a service entry component, a rule-based policy component, a service catalog, and a service-rendering component. In addition, an aggregation engine contains node services, such as monitoring, logging, error recovery, and so on. The SD presents a single Web-services interface, describable by standard Web-services specifications, but it provides a higher-level interface for managing a group of Web services (a service may be an application, a software function, a data access, or an IT resource used in a solution implementation).

The SD aggregates and manages multiple service instances as a single virtual service. It offers a SOA solution that reduces the complexity of building business applications because the applications need only focus on the services and user interfaces (UIs) specific to the business while the SD provides most of the enabling logic. Figure 1B illustrates the Web services model with an SD replacing the individual service provider. A client needs to find and bind once to an SD. The SD hides the complexities involved with using services: discovery, selection, exception handling, and so on. Solution deployment is faster and easier, and client access is simpler.

Figure 1 Web-service model (A) and Web-service model with a service domain (B)



The SD model starts with what is available today and builds a service management and brokering middleware solution designed to address the previously mentioned challenges. Its objective is not to define new application programming interfaces (APIs) or new standards, but to construct from the existing components a new, higher-level structure that can hide complexities from service users, simplify deployment for service suppliers, provide self-managing capabilities, and give administrators a set of tools for managing the IT solution.

The SD, which implements Web services and grid concepts, is extendable to include new grid standards that are still evolving. SDs often can be structured as a logical hierarchy. For example, an SD that provides portfolio management and purchase-ordering services may direct requests to a descendant (child) SD that provides services for executing the transactions (e.g., the submission and processing of jobs). A further descendant SD may provide resource-allocation services, such as allocation of computing power, storage capacity, and other execution-related resources (similar to data and resource management in grid computing). Such an implementation of an SD hierarchy is a *service grid*.

The SD model enhances the Web services concepts of proxy, interceptor, handler, gateway, mediator, and broker by incorporating concepts of grid computing[5,6,7] and autonomic computing.[8] Instead of implementing individual functions for each service, such as action selection, routing, failover, and change management, the SD model provides a service grid, a pool of dynamically assembled service instances. The SD automatically dispatches the best service instance corresponding to the user request through built-in mechanisms for registering and discovering service instances.

The SD model is based on the standards specified by W3C**[9] and OASIS.[10] These include Web Services Description Language[11] (WSDL), Extensible Markup Language[12] (XML), Simple Object Access Protocol[13] (SOAP), Hypertext Transfer Protocol[14] (HTTP), Universal Description, Discovery, and Integration[15] (UDDI), Web Services Inspection Language[16] (WSIL), and Web Services Invocation Framework[17] (WSIF).

Kraft[18] presents a Web services collection model that can contain a group of abstract Web services objects. This approach, which provides a convenient way to group Web services, is useful for designing a distributed access-control mechanism because it facilitates the management of authorization specifications and meta-data and the composition and specialization of Web services. This model does not support rule-based automatic service aggregation.

The Global Grid Forum[5] (GGF) defines a Service Group port type as an interface to a collection of grid services. The grouping model does not assume any relationship among the member services in the group. It also does not support rule-based automatic service aggregation.

The emergence of the Web Services Resource Framework[19] (WSRF) and the associated WS-Notification are bringing together grid computing and Web services.[20] The Open Grid Service Architecture[21] (OGSA) defines key building-block services for grid computing. These defined services are fundamental to IBM's on demand operating environment[22] that provides system management, autonomic computing, data management and storage management, knowledge management and collaboration, and business-computing services. Given the independent nature of its service abstraction (whether it is for a specific IT resource, system service, software solution, or business-application function), the SD model could become a key building block for the OGSA fabric as well.

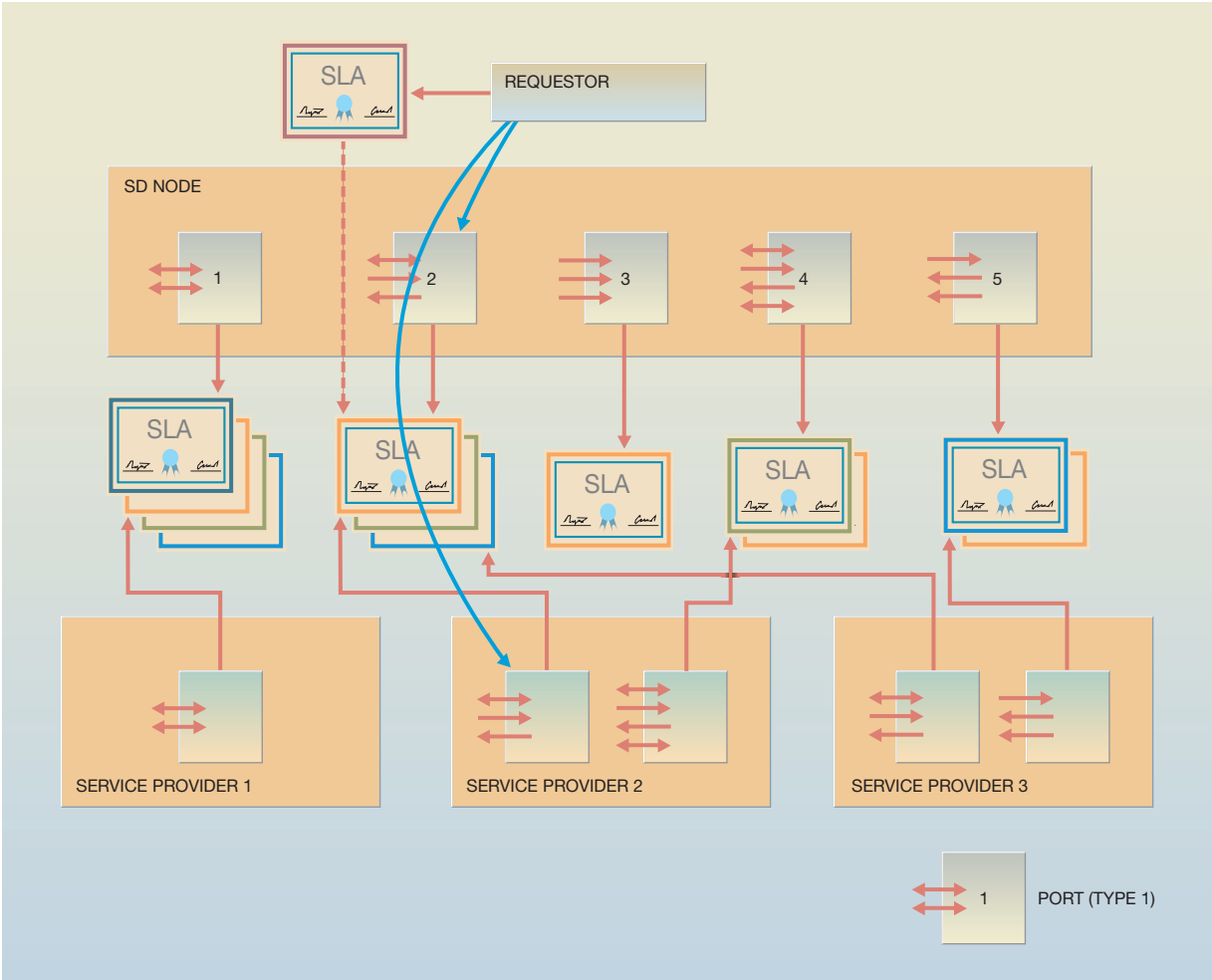
The rest of the paper is organized as follows. In the next section on the architecture of SDs, we cover the concepts of port-type aggregation and SD hierarchy, we describe the structure of an SD node and its components, and we discuss the handling of the state associated with a Web service. In the section that follows, "Implementation," we describe our reference implementation and its application to the implementation of a large grid now in progress. Next, in the section "Implementation aspects," we provide additional implementation details by discussing setup and deployment of services, failover and recovery, and rule-based service selection. We conclude with a summary of the paper and directions for future work.

Architecture

The SD model allows the aggregation and sharing of multiple WSDL-described Web services. Services from various sources are virtualized as a single logical service. An SD has a set of port types representing the set of distinct services it offers. Each port type corresponds to a collection of similar service instances. Rules are provided to manage and control the behavior of the aggregated services.

Figure 2 illustrates the aggregation of services and the resulting port types. The SD node shows five aggregated ports. Each port has a set of arrows indicating the operations and message flows. There are three service providers shown; provider 1 supports port type 1, provider 2 supports port types 2 and 4, and provider 3 supports port types 2 and 5. A requestor sends a request for port type 2 of the SD. The request can be forwarded either to port type 2 of provider 2 or port type 2 of provider 3.

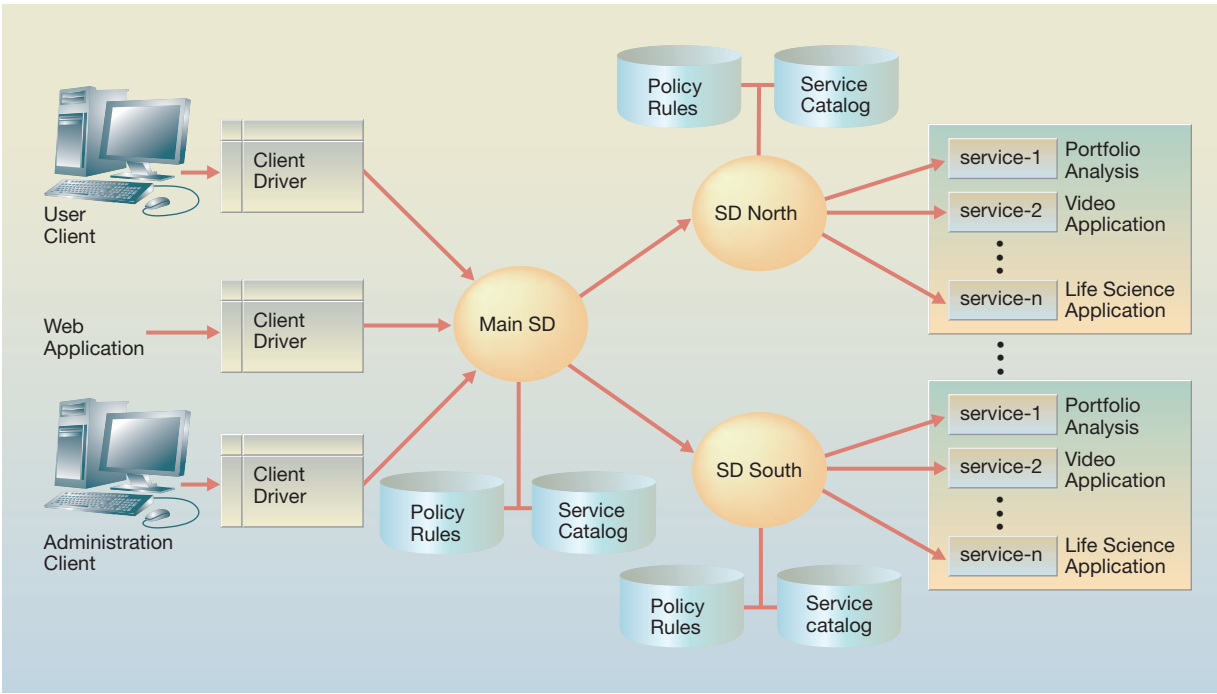
Figure 2 Service aggregation and the resulting port types



Each port type is associated with a set of service level agreements (SLAs), each specifying a service level. In order to offer a service to clients, the service provider registers the service with the SD, specifying the port type and an SLA. The SD can then incorporate that service into its own version of service. In the scenario depicted in Figure 2, the SD matches the user service level with the provider service level and selects provider 2. The two service levels are compatible but need not be identical. By using SDs a service broker is thus established that can manage a heterogeneous group of service suppliers in order to provide virtual and additional value-add services to consumers.[23,24] The simplest SD is a single collection of similar service instances, much like a customer service desk in a store in which the service desk is staffed with several attendants. Whereas a store may have several customer service desks, an SD implementation may have a nested architecture consisting of a hierarchy of SD nodes. In that case, a service requestor interfaces with a single logical SD. The SD implemented as a hierarchy of nested SDs allows the creation of a large virtual business complex referred to as a service grid.

The motivation behind the SD concept is to achieve manageability when dealing with a large number of services and service providers. Although we consider how the SD can exploit existing standards, products, and emerging technologies, the main focus is on re-

Figure 3 Service domain topology



ducing the complex issues to simple Web service invocations and thus view Web services as the ultimate standard.

Here is a list of the SD-related terms used in this paper:

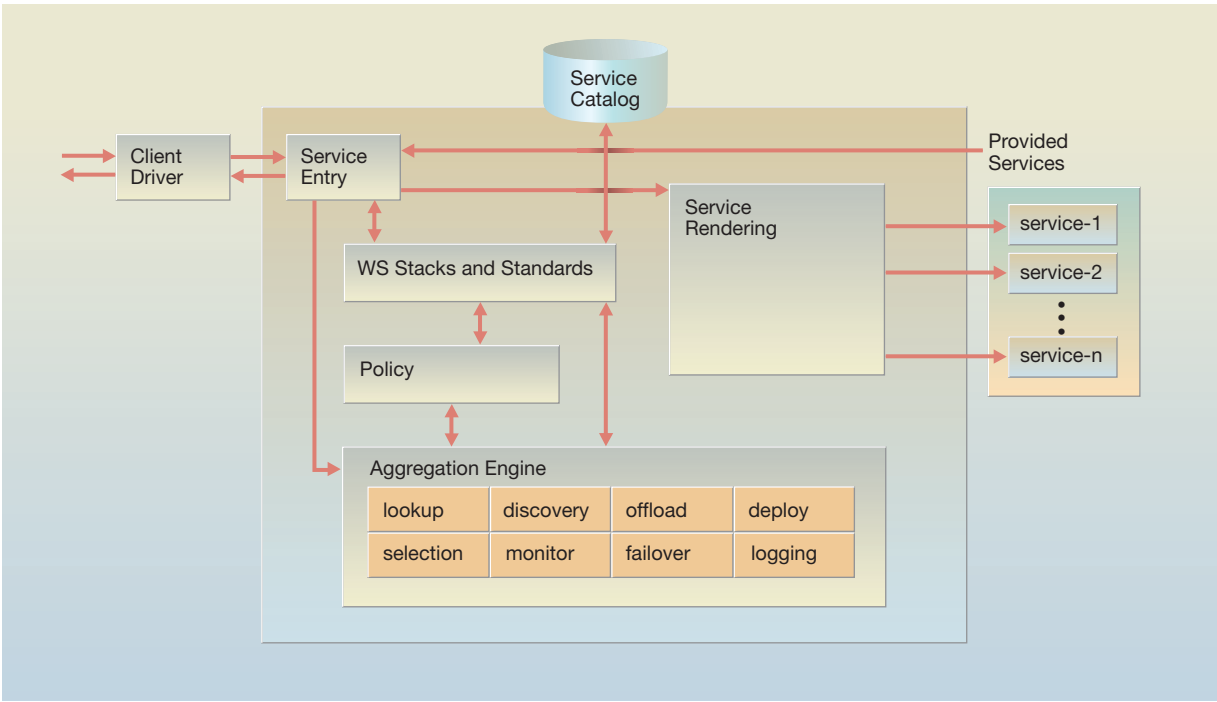
- Two service instances are *similar* when they implement the same functionality but provide a different API or different service level characteristics.
- Two service instances are *compatible* when they implement functionalities that overlap.
- Two service instances are *related* when they implement functionalities in the same category. They may be disjoint and thus augment each other
- A *virtual service* is the service offered by an SD that results from aggregating several service instances.
- A *virtual port* is the port visible to the SD clients, which is implemented by a mapping to the port of a service instance.

Service domain hierarchy. Figure 3 shows the topology of a logical SD hierarchy. It is a simple topology that consists of a *main* SD node and two *secondary* SD nodes: SD North and SD South. Service instances of various types, such as portfolio analysis, video applications, and life science applications, are provided through the two secondary SD nodes. The service instances that are similar are aggregated on the main SD node and presented as virtual ports. Each SD has a set of policy rules that governs its operations and a service catalog (or registry) that stores the information collected about its services. The SD nodes and their services are distributed over a set of hardware resources.

The clients interact with the SD through the main SD node using the Web services client interface. Typically a *client driver* that includes a Web application server mediates between the user browser and the SD node. When a client calls the SD for a service, e.g., portfolio analysis, the main SD consults the policy rules and dispatches the request to, say, SD North for processing. SD North will in turn consult its own policy rules to select a service instance to service the request.

In this example SD North had been "registered" with the main SD prior to the service dispatching. This operation and other similar ones, such as setting of

Figure 4 Architecture of an SD node



policy rules, are managed by the administrator through an administrative client. As an implementation note, a single administrative client can be set up to control an entire logical hierarchy.

The topology of the SD hierarchy is *self-configuring* because it supports automatic expansion or contraction of the tree, as SD nodes join or leave the SD hierarchy. A service instance can be assigned to an SD node at any level. Services that conform to a given port type interface are logically aggregated into a virtual port type that is surfaced at the main SD node (at the highest nesting level of the topology).

Architecture of an SD node. Figure 4 shows the architecture of an SD node. The main components are: service entry, service catalog, Web services stacks and standards (WSSS), service rendering, policy, and aggregation engine.

The SD node is implemented as a J2EE**-instantiated[25] object in a platform such as WebSphere* Application Server[26] or Apache Tomcat.[27] The object's behavior is similar to a simple Java** bean, and it can be represented as a WSDL-described service. It is built using grid and Web services standards such as WSDL, OGSA, SOAP, XML, UDDI, WSIL, and WSIF. All dependencies on these standards are limited to the WSSS component so that the evolving nature of the standards will not impact other SD node components. This design of WSSS makes the SD node architecture extendable and able to exploit OGSA defined services[28] and emerging Web services standards[29] as they mature.

As Figure 4 illustrates, three of these components provide external interfaces to the SD node: service entry, service rendering, and policy. The service-entry component provides standard Web services or other means for accepting requests and returning responses. Aside from service requests from clients, the service-entry component also supports administrative operations for managing the SD node. The latter are viewed as operations directed to the "home" port. For example, an administrative client may send a register command in order to register a service instance with the SD node. The information entered is stored into the service-catalog component using the "standard" registry service interfaces.

The service-rendering component provides an *attachment interface* that supports any WSDL-described Web service offered by a service provider. The attachment interface allows the inclusion of information beyond WSDL definitions; we refer to this information as *augmented information*. The augmented information consists of XML-specified service attributes in a format that is consistent with current standards such as the XML-defined nouns and adjectives, J2EE conventions for object properties, and the early grid specifications of service data elements. These service attributes are useful for service discovery and service aggregation. An example of such an attribute is the service "origin" type used to support a variety of Web-services dispatching approaches, including SOAP, WSIF, grid, Microsoft .Net,[30] and document style,[31] for the service instances. It is stored in the service-catalog component when a service instance is registered. The service-rendering component uses the attribute to set up the correct Web services client-proxy call to dispatch the service instance.

The SD architecture does not impose any initial state requirements or other restrictions on the services beyond conformance to WSDL standards. Any initial state requirements are driven by the implementation of the specific services. A stock quotation, for example, does not require the assignment of an initial state. A job-execution service, on the other hand, is assigned an initial state of "ready."

The service-catalog component provides an internal mechanism to store the WSDL, service attributes, state (operational or down), and status (collected statistics) data in one place. Where it makes sense to have constituent services sharing state and data, SD can share its service catalog with the individual services. The sharing uses the same "home" port operations that SD provides to an administrative client. The service catalog is built on top of a registry such as UDDI. A service catalog can be shared by several SDs, as illustrated in Figure 4.

The SD node stores the (static) WSDL information in the registry. It stores the augmented information in a supplementary data structure in memory, and it stores the dynamic operational data (status) in persistent storage (a file).

Administrative clients use the home port as a higher-level interface that eliminates the need to deal with low-level interfaces such as registry APIs and data formats and enables operations for configuring and managing registries. For example, a client can make a simple call and obtain the list of registered service instances without the need to specify the registry used. If the registry is based on UDDI, this will be shown as an attribute value that indicates the type of registry.

Because the service-discovery function of the SD uses real-time data, it is different from conventional registry queries. The discovery can include service features, business relationships, and performance characteristics. Both static and dynamic information are maintained in its processing storage.

The policy component provides a policy interface that allows configuration of all the operating rules for the SD. A service policy is a set of XML rules that includes service-level definitions and rules for handling security, recovery, events, discovery, service selection and routing, service mapping, and various business considerations. The XML document is constructed and entered through a setPolicy command to the service-entry component. When the command is processed, the policy rules are entered into the policy component using a "standard" policy service interface.

Rules are the centerpiece of the SD model. They differentiate the model from other service management middleware such as registry, grouping, proxy, gateway, and intermediaries. These rules are interrelated; for example, the selection rules are related to the service level definitions and service mapping; selection errors are resolved by the recovery rules; business-relationship-based selections depend on other miscellaneous rules.

When the service-entry component receives an incoming request, it uses WSSS to interact with the aggregation engine. Within the aggregation engine the lookup subcomponent is called first to identify service instances available to process the request. Then the selection subcomponent is called to identify the best service instance for this request.

The service-entry component then calls the service-rendering component to invoke the specified service instance. On completion a response message is returned to the service-entry component, which forwards it to the requestor.

The aggregation engine obtains policy rules from the policy component and uses them to enforce specific behaviors in the various subcomponents on the re-

quest processing path. For example, the lookup subcomponent communicates with the deployment subcomponent to obtain information about registered service instances. The monitor subcomponent collects various metrics and tracks appropriate thresholds, whose values may affect the selection factors used by the selection subcomponent.

The aggregation engine also contains subcomponents for handling exceptions. When the ability of the SD to handle additional incoming requests becomes limited, the offload subcomponent can redirect further requests to peer SDs. Similarly, when a service instance fails, the failover subcomponent can dispatch another service instance by calling service rendering. When a shortage of service instances is detected on a virtual port, the SD can query and acquire service instances from other SDs. All these actions are controlled by policy rules.

Stateless and stateful Web services. The SD node maintains a session for each active client. If the service selection rule is set to a "transaction history" directive, the client request is routed to the same service instance. For example, stock quotation requests from user A are routed to the financial service that serviced A's previous request for a stock quotation. There are other directives, such as "designated" and "fixed" directives, that are associated with the user and extend across sessions. For example, financial requests from user A are routed only to those service instances in which A has an account.

When a failure occurs, the state can be recovered in a client-transparent way if the failure affected a single invocation of a service, in which case the SD node dispatches another service instance for handling the failed request. If the client invokes a transaction sequence that is not under the control of a transaction manager and a failure occurs, the transaction manager is not able to handle the recovery. Ideally, such a sequence should be handled by a workflow service that the client can invoke and that is able to handle recovery as part of its logic. The recovery can be handled by a rule that listens for a workload-service-exception event and then invokes a back-out service to reverse the sequence.

In general, individual service requests can be stateless (have no memory of previous requests), but because most services maintain their own states, the result is a stateful application (an application that keeps track of the state of interactions). The SD does not interfere with application-managed state, whether managed by a simple client or by a workflow service, as suggested above. For example, a request for a travel reservation leads to relevant information being saved by the service instance. In a follow-on transaction involving the same service instance, the reservation leads to a purchase.

There are situations in which several service instances need to share state information. Consider, for example, a financial service that aggregates stock quotations from three independent service providers: DOW JONES**, NASDAQ**, and FTSE.** Creating such a service that allows the client to monitor a list of stocks using all three service providers requires shared state. Although we have not yet implemented this type of state sharing, it is not hard to achieve. A service request to monitor a specified list of stocks is sent to all three services. Because the SD node presents common interfaces to these services, it is able to compose the individual responses into a single response to the client. Thus, the SD node controls the sharing of information without the need for sharing among the service instances.

For more information on SDs, see References 32–37.

Implementation

Our reference implementation of the SD architecture is packaged as a toolkit that can be used in pilot projects. The toolkit includes three major components: (1) node objects, (2) service objects, and (3) the user interface. There are four node objects in the toolkit, each associated with a type of SD node and spanning a range of node capabilities. The service-objects component supports internal node functions. The user-interface component includes a client driver that includes a Web application server. The toolkit provides HTML, J2EE, Java, and Web-services interfaces. A version of this toolkit is publicly available as Service Domain Technologies in the Emerging Technology Toolkit[24] (ETTK) at the IBM alphaWorks* Web site.

The use of the toolkit involves the following tasks:

- WSDL-defined services are created.
- The offered services are attached (plugged in) to SD node objects in the SD hierarchy.
- A client driver is prepared—the client driver included in the toolkit may be used, or a customized one may be built as a Web-services interface client that accesses the main SD node and interacts with end users.

- The presentation layer is built on top of the client driver layer using JavaServer Pages** [38] (JSP**), servlets, beans, portlets,[39] or other suitable technology.

Reference implementation. The SD node provides a number of administrative functions, which are mostly used by the administrator, although some can also be used by clients or suppliers. These can be viewed as operations addressed to a special port, the home port. The administrative functions include:

- Register/Unregister: Support the addition or removal of a service instance.
- setPolicy/getPolicy: Support setting up a registered service instance or the selection of a service instance. The setPolicy function requires a number of XML-encoded parameters that include an SLA (the contract), service selection rules, recovery rules, and so on. A service supplier selects a "supplier service level" when entering in a contract with the SD at service registration time. A user profile system determines "user service levels" for user contracts with the SD and assigns them to a service requestor at login time.
- setServiceInfo/getServiceInfo: Support (1) the creation or querying of SDs, and (2) the setting or querying of registration data for service instances. Specifically, data and attributes associated with the SD (e.g., performance metrics), and data and attributes associated with individual service instances (e.g., SLA) can be queried or updated through these operations.
- Login/Logout: Start or end a session. The SD uses the session to monitor the state of service execution for a client.
- InvokeOperation: Process a service request and forward the request to the appropriate service instance.

The degree of aggregation (the nesting level) of the SD determines the form of the SD node object. The simplest SD node object supports an arbitrary collection of service instances without aggregation. The next level up—a single level hierarchy—is an SD node that aggregates similar service instances, which are presented to the user as a virtual port. Next we can have a two-level hierarchy, and so on.

There are six steps involved in operating an SD node object: (1) creating the SD node object, (2) deploying the node object as a Web service, (3) preparing and activating XML-encoded policy rules, (4) registering service instances, (5) having clients join a user group registered at the SD, and (6) becoming operational.

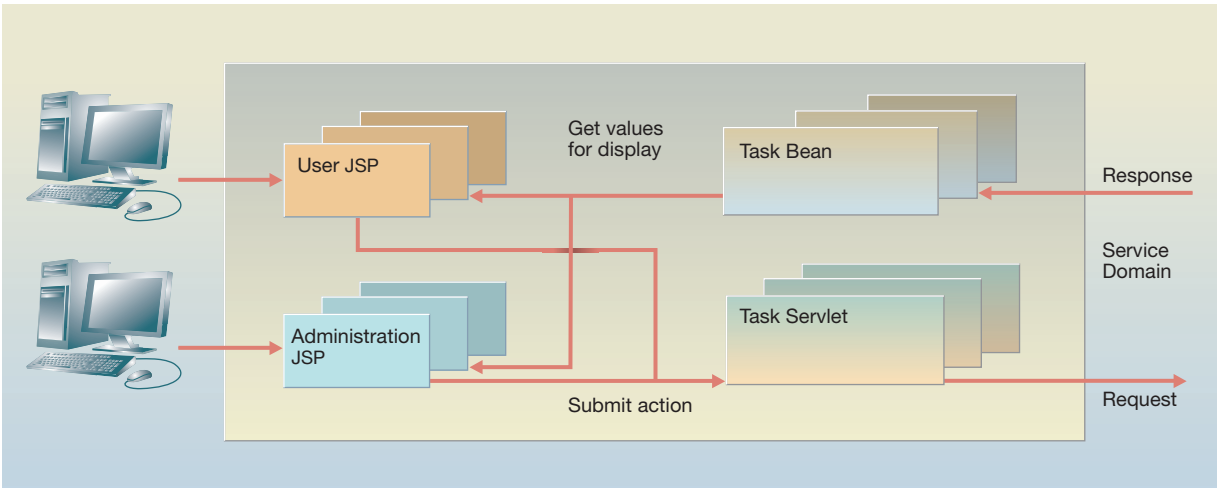
Step 1 represents the initialization of an SD node object with Java interfaces. Step 2 deploys the object as a Web service by using a WebSphere Application Server. Step 3 involves the invocation of the setPolicy operation. At Step 4, the Register operation is invoked through the administrative client. Step 5 involves assigning user service levels of SD to user groups by using existing user management systems. Users invoke the Login operation in order to access the SD node. The login operation contacts the user management system and obtains the user service level. At Step 6, the operations setServiceInformation/getServiceInformation, setPolicy/getPolicy, and InvokeOperation are invoked repeatedly in the course of normal operation. Solution developers provide application-specific service instances, such as finance, trade, and travel, to form and extend the service grid.

As an example, a stock portfolio management service can be established by instantiating a base SD node object, registering individual service providers (e.g., stock quotation service provider), and setting up the selection rules for various groups of users. The stock portfolio management service could be extended by also enrolling market information and analysis services as secondary providers. Customers see an integrated offering from a single service. The service owner has the flexibility to enter into partnerships with other businesses.

Toolkit service objects are core building blocks corresponding to subcomponents of the aggregation engine. Their design is modular and provides a customization and migration path to future environments when Web-services and grid standards mature. As shown in Figure 4, discovery, selection, logging, monitoring, and lookup are in this category. A base SD node object provides the linkage interface to add and delete the SD service objects to the SD node object. For example, the Register operation of an SD node object is composed by adding the "registry" service to the SD node object. The actual implementation of the "registry" service can be customized further to support a third-party-provided registry.

The administrative client driver included in our reference implementation is ODSG (On Demand Service Grid). ODSG enables the administrator to perform SD control and configuration tasks. In addition, ODSG provides additional views for users, service providers, and the SD owner. The users can view the list

Figure 5 ODSG architecture



of services available on the SD. The service providers can register their services with the SD.

Figure 5 shows the architecture of ODSG. It includes task servlets, task beans, user JSPs, and administrative JSPs. User and administrative JSPs invoke the appropriate user or administrative task servlets. The task servlets then send requests to the SD ports. The response JSPs include the task beans associated with the result of the service calls in order to get related information for navigating the tasks. For example, a user portfolio-management service-invocation JSP drives an "invocation task" servlet to issue the "get my portfolio" service operation. The result from "get my portfolio" is packaged by the "invocation task" servlet as a task bean in the portfolio-management service-result JSP, which automatically executes the bean to generate the HTML page for display.

Administrative tasks include invoking control actions, activating policy rules, configuring service level definitions, registering suppliers, handling supplier and user contracts, and monitoring status. User tasks include listing active service ports and launching services. Further navigation of service interactions after the launching can be controlled by a user client driver that can be built by extending the administrative client driver.

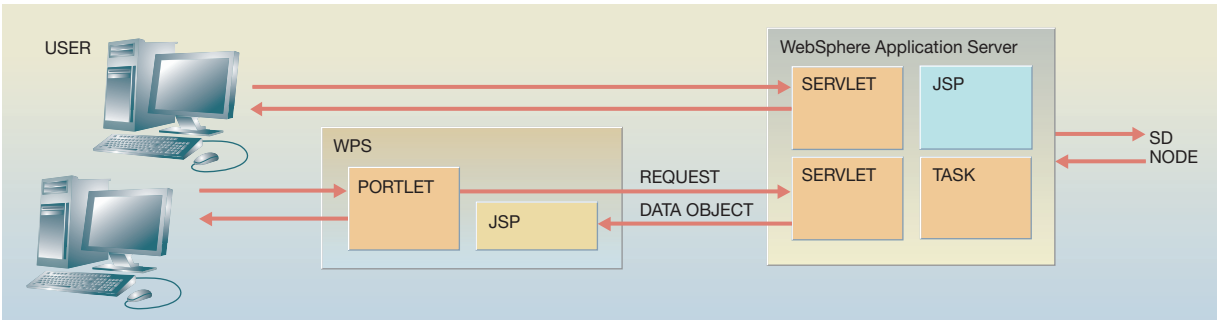
Figure 6 illustrates a client driver with access through a portal and the use of the WebSphere Portal Server[39] (WPS) in lieu of plain JSPs to gain access to the services provided by an SD. Customer portlets interact with existing client driver servlets and beans, which in turn access the SD node.

Setting up a service grid involves coordination among grid owners and administrators, solution developers, and system developers. Grid owners and administrators use the ODSG UI to configure the service grid, set operating rules, and register secondary nodes and service instances. The grid owner defines the WSDL interfaces of the port types to which the service instances need to conform. The administrator operates the service grid using an administrative UI. End users access services through a UI that allows access only to services that they are authorized to use. Solution developers focus on implementing services, coding policy rules, and functionally enhancing the service grid. System developers focus on implementing and customizing SDs.

Customer use cases. The reference implementation is being tested with several pilot projects. These pilots will help us understand whether the concept of "letting services manage services" has merit and whether the SD model can make it "fast and easy" for customers to implement new business applications. Early experience and preliminary data are promising.

The China Education and Research Grid (China Grid) is such a pilot project. IBM and China's Ministry of Education announced in October 2003 that they had begun using grid technology to enable universities across China to collaborate on research, sci-

Figure 6 Client driver with portal access



entific, and educational projects.[40] China Grid is one of the world's largest implementations of grid computing, in which untapped application, data, and computing resources from different computing systems are made available where and when they are needed, resulting in a single, virtual system.

When the project is completed, the China Grid will link more than 200 000 students and faculty members at nearly 100 universities across China. When Phase 1 of the project is completed in 2005, the grid will perform more than 6 teraflops, or trillions of calculations per second, and eventually will be capable of more than 15 trillion calculations per second.

The grid relies on a technology preview (experimental version) installed on WebSphere Application Server 5.02, which implements the SD concept and exploits OGSA standards. It will simplify how students and researchers access educational and computing resources across China. Universities will be connected to a common virtual hub that automatically finds the appropriate application resources, from life science research to video courses and e-learning. China's university system will save on development costs because each school can focus on its area of expertise—e-learning or life science, for example—and tap into other applications as needed via the grid.

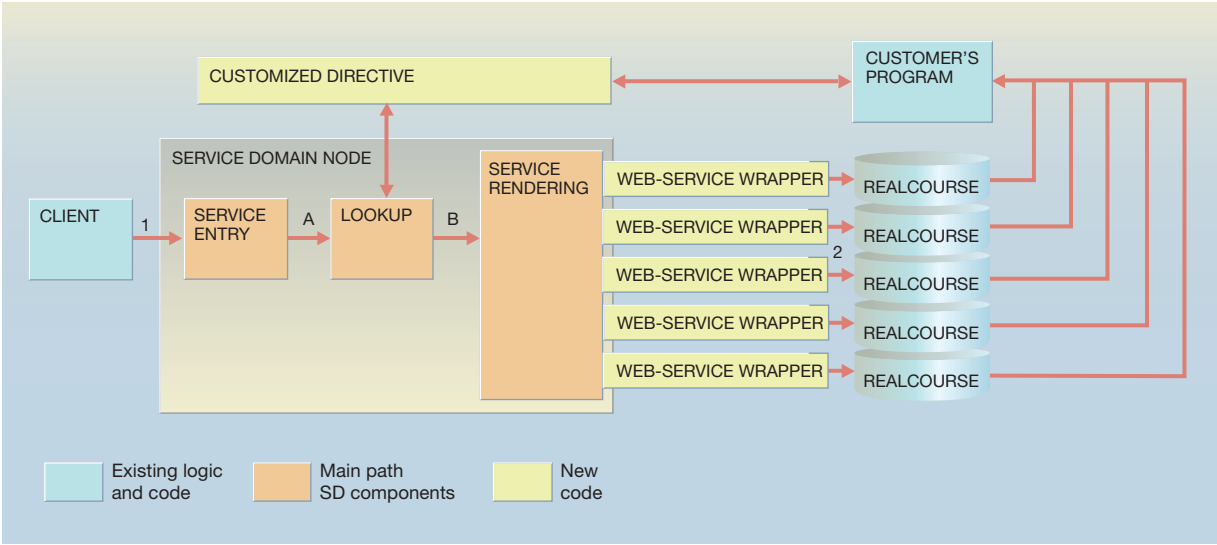
The first applications deployed on the China Grid include a life science application that runs complex computation tasks such as protein structural analysis, a "video course on demand" application that provides students with speedier access to video courses by distributing information through distributed servers, and an e-learning application that enables students to practice Mandarin through an integrated learning portal. In the next section, we provide a more thorough description of the video course application, as it exemplifies a common SD usage pattern for integrating legacy applications.

Figure 7 illustrates a video course application deployed on an SD node. The servers that host the application (also referred to as data servers) are represented as devices labeled RealCourse. The application is presented to the SD node as a set of service instances that are registered at the node and attached through the service-rendering component. A client browser accesses the application through a Web interface by requesting service from the SD (this step is labeled in the figure as 1). The SD must select a service to forward the request to. Because the video course application is a legacy application with existing data server selection logic, the SD node allows it to be plugged in as custom selection logic and invokes it for service lookup and selection. Starting at the lookup step labeled "A," the lookup component calls a customized selection directive, which calls the existing selection logic to return a service instance representing a data server. In the service-rendering step "B," the SD passes the selected service instance to its service-rendering component. The service-rendering component invokes the service instance to hand over the request to the data server. In step 2, the instance simply launches the data server and exits. The data server then interacts directly with the client browser.

A total of nine universities were attached to the grid by the end of 2003. The SD software was deployed and tested in a matter of days. The application-porting time, which involved the customized directive component and the Web service wrappers (depicted in Figure 7 as "new code") was also quite low.

We have used our implementation in additional internal company and customer-proof-of-concept en-

Figure 7 A video course application deployed on an SD node



gagements. All have shown that the SD model can have a wide applicability to SOA-based solutions. The SD model provides a repeatable solution pattern that starts with WSDL service definitions, followed by service deployment to SDs, and finally the creation of policy rules and GUI implementation. With this pattern, a solution can be quickly developed and deployed and then successfully maintained.

Ease of deployment, ease of use, continuous operation, and nondisruptive refinement are the notable characteristics of this solution pattern. We tested this pattern at a prototype level successfully, using an internal laboratory experiment environment. The types of services deployed included a diversity of high-performance computing tasks, Internet commercial and information services, system-oriented services, and data-oriented services. We built service grids for our sister labs across the oceans and registered them to our service grid at Raleigh. Service instances were discovered, aggregated, and shared immediately according to the operating rules we set up.

Implementation aspects

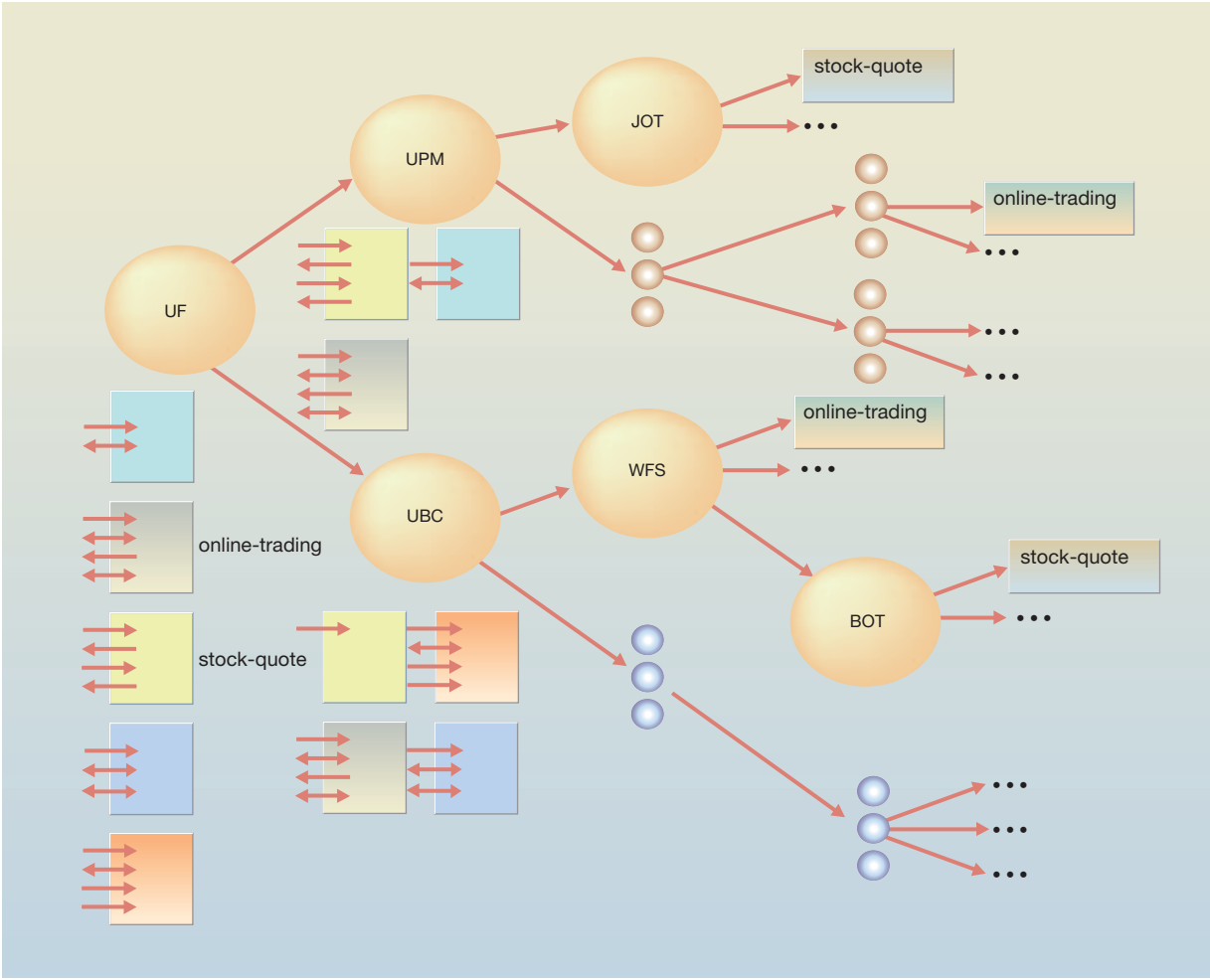
In the following sections we use application scenarios to illustrate the details of the implementation focusing on three aspects of the SD model. We discuss setup and deployment of services, failover and recovery, and rule-based service selection.

Setup and deployment of services. In this scenario, a number of financial services are deployed within an SD hierarchy consisting of a main SD and two secondary SDs. We focus here on three steps: the registration of service instances, the configuration of policy rules, and service discovery.

Registration of service instances. An SD owner creates the WSDL service interfaces with which all service providers must comply. The SD administrator defines the set of SLAs to be associated with each service interface. Service instances offered by various service providers are deployed by registering them with specific SD nodes. The functions provided by a service instance need not cover the entire service interface specification; moreover, each service instance may have different performance characteristics. The only requirement is that the functions supported and the performance characteristics comply with the SLAs (in the policy rules) committed to the SD.

Figure 8 depicts a service grid for financial applications. The SD owner, United Financing (UF), has a number of partners: United Business Consulting (UBC) provides three services (stock quotations, online trading, and personal loan services); whereas, US Portfolio Management (UPM) provides four services (stock quotations, online-trading, personal investment banking, and commodity-trading services). The SD topology reflects these relationships: the main

Figure 8 A service grid for financial applications



SD node, which is labeled UF and represents the logical aggregation of services, has two descendants labeled UBC and UPM.

As a result of the aggregation, UF supports five virtual service ports to clients; whereas, UBC supports four ports, and UPM supports three ports (see Figure 8). It is of no consequence to UF how the lower-level services are implemented. As shown in the figure, the UBC and UPM services are implemented as aggregations of lower-level services by business partners.

The aggregation process associates each service instance with the appropriate port type automatically by inspection of its interface definition. This process also applies filter control and generates a virtual endpoint address for every new port type created. For example, Bob's Online Trading (BOT) is registered with World Finance Services (WFS), which in turn is registered with UBC. John's Online Trading (JOT) is registered with UPM. As previously mentioned, UBC and UPM are registered with UF. As a result UF offers a virtual port that accepts stock-quotation requests and forwards these requests to one of the two available service instances, BOT and JOT. Only the virtual port of the main SD node is visible to clients.

Configuring policy rules. Policy rules enable an SD owner to perform administrative tasks such as ag-

gregating services with different interfaces, function, and quality-of-service characteristics. The SD owner defines a set of service levels and then configures the operating rules that control the runtime behavior of the SD node, that is, service selection and recovery, provider relationship management, SD peer interaction, service discovery, event management, and so on. Providers declare the service levels they support at registration time. Clients in turn select a set of service levels that governs their interactions with the service grid.

As an example, UF can offer all the services aggregated from UBS and UPM to users under one "platinum" package: stock quotation, online trading, personal loan, personal investment banking, and commodity trading. It can also offer a "trial" package that consists only of stock quotation and online trading. Users can use only the services they subscribe to. UF could set the following operating rules to differentiate the services received by clients with different subscriptions:

- For platinum users, enable the UBC service, which provides multiple quotations, data analysis, and overseas portfolio-tracking operations related to a stock quotation service.
- For trial users, enable the UPM service, which offers a simple stock quotation service.
- For online trading operations, select either the UBC or the UPM service to process requests.

Furthermore, the SD model allows for selection logic that uses the user profile information to determine the appropriate service instances to invoke. For example, if user Joe has accounts with financial-service companies A and B, the administrator can choose to route requests for stock quotations to either A or B and other companies; whereas, stock purchase requests are routed to only A or B.

Service discovery. Registration of new service instances changes the SD topology dynamically by adding leaves to the SD hierarchy. The SD topology is dynamically adjusted by the service discovery process, which uses a coordinated iterative polling algorithm. The discovery process identifies all service instances that can be used to respond to client requests. The polling process is initiated periodically by the top SD node. Each intermediate node responds to its parent node with information about the active ports that are hosted either by the node itself or by one of its descendants. The top SD node publishes the virtual ports surfaced at its level. These ports become the (external) ports in the SD WSDL. Their endpoint addresses point to the top domain node with enough information for that node to dispatch to a local service instance or to an immediate-child node; likewise, the immediate-child node dispatches locally or to one of its immediate children (optionally, the top domain node could also publish the WSDL for each port type individually).

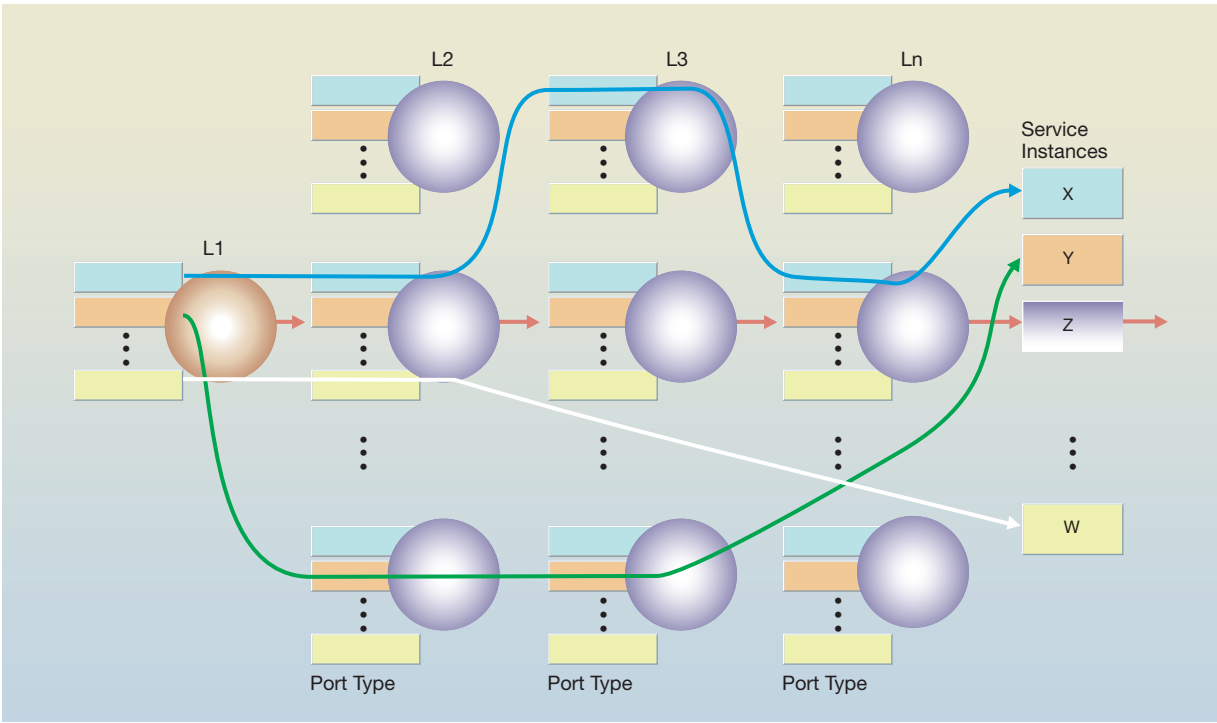
Figure 9 illustrates the service discovery process, which starts at the main SD node at the highest level of the hierarchy labeled L1 and terminates at a service instance. There are three paths illustrated, each one corresponding to such a selection process. The service instances are attached to various levels of the SD hierarchy, and thus the path lengths may vary. For example, the path that leads to service Y passes only through the first two levels of the SD hierarchy.

Failover and recovery. An SD has some failover capability built in by nature of its service virtualization feature. For example, an SD that selects among comparable service instances by using a simple round-robin algorithm can, in case of service failure, transfer the request to the next service instance, the requestor remaining unaware of any problems.

The state of service execution is monitored for each individual client session. The SD node monitors exceptions, records the response time between the issuing of the request and the arrival of the response at the client, tracks time-out values, records the number of requests received in a monitoring interval, tracks the percentage of requests received while service is not available, and records the history of services used and operations invoked for each client session. This abundance of information is available because all service requests and responses flow through the SD node. From the service instance side, the SD also provides an interface for a service instance to record a small amount of state data.

Figure 10 shows a recovery scheme involving "percolation" of the error through the SD hierarchy, which consists of Main SD at the root and its three descendants. When SD A detects the failure of service instance Y, instead of dispatching service instance X (the course of action if a round robin algorithm were applied), A notifies Main SD. Main SD selects C to handle the request. C applies a fixed selection order algorithm that leads to the selection of service instance W to handle the service request.

Figure 9 Service discovery



The decision at A not to select X for recovery may have been based on the likelihood that the selection of X would also generate an exception condition. For example, assume the exception at Y occurred due to insufficient capacity at the service provider. Moreover, assume that the provider of X is known to have similar capacity limitations. Then, policy rules at SD A would correctly infer that the selection should target a provider that is known to have spare capacity.

Recovery from failure of a service instance. In this section, failover processing is described more fully by using a credit-check service as an example. In this scenario, an SD node detects a problem with one credit-check service provider and automatically switches to a different provider. When a lower-level SD node becomes unavailable, Main SD uses a global problem determination mechanism to tap into available service instances from a backup SD node.

Using Figure 11 as an illustration, we assume a "credit check" port type is configured in the Main SD node. The port is implemented by mapping corresponding ports in two secondary SD nodes, A and C. Service instances X, Y, V, and W are registered as credit-check services at their respective SDs. The SDs in this scenario are configured to percolate error exceptions to Main SD. A personal-loan business that needs to verify the credit status of the loan requestor addresses its request to the credit-check port of Main SD. The failure of service instance X is percolated upward to Main SD, and then to SD C for handling the recovery.

The failure of X could be caused by a connection failure, by a service-unavailable condition (such as "unregistered"), and so on. Whereas some failures are detected when they occur, the detection of other failures may depend on a time-out value or on a fixed polling cycle. Until the next polling cycle, Main SD will still consider SD A to be capable of handling credit-check service requests and will continue to send it requests for processing even though its service collection is empty. SD A can be viewed as being in a "stand-by" mode; although for the time being, A may not be able to provide service, it is advantageous to preserve its state in order to restart service quickly when service instances become available.

Figure 10 Error percolation

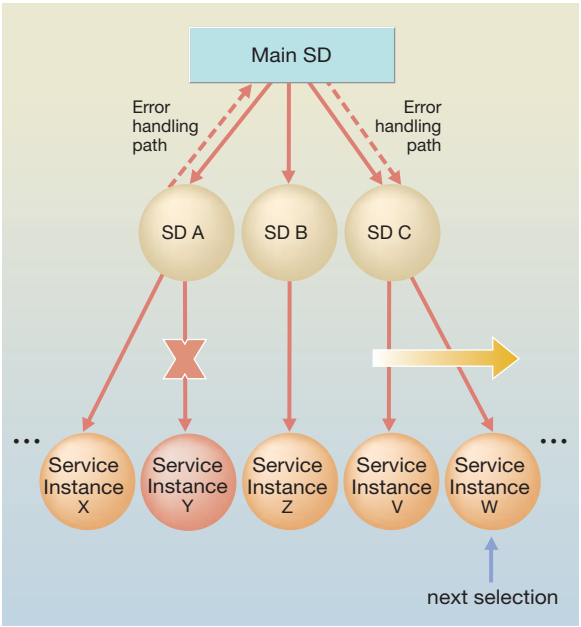
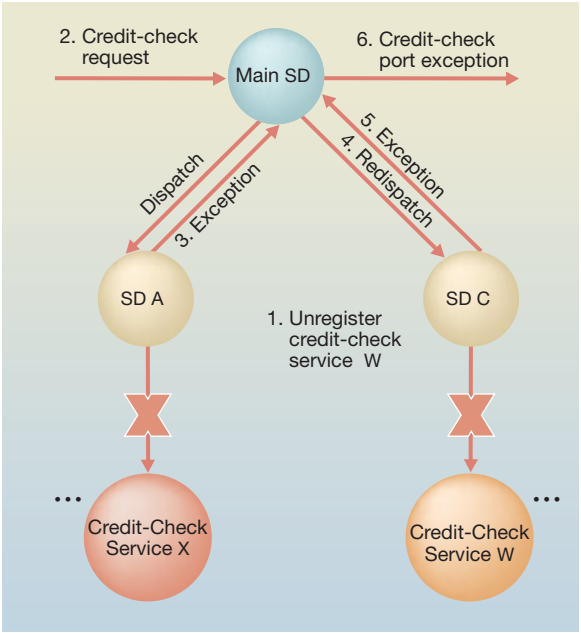


Figure 11 Processing of a virtual port failure



Components such as SDs that have self-healing capabilities can recover from failure without external intervention. A problem-determination framework such as the autonomic computing track of ETTK is useful when communication between components is needed for global error recovery. This is described in more detail in the next section, which deals with a scenario in which the credit-check port of Main SD becomes unavailable.

Recovery from virtual port failure. Figure 11 illustrates the processing of a virtual port failure in an SD hierarchy consisting of Main SD and two descendants, A and C. At step 1, credit-check service W in SD C is taken out of service (unregister command) for maintenance purposes. A credit-check request arrives at Main SD at step 2. Main SD discovers service instance X and dispatches the request to SD A, to which X is attached. At step 3, the failure of X is detected by A, and A notifies Main SD. At step 4, Main SD redispatches the request to SD C. C detects an exception condition at service instance W, and at step 5, it notifies Main SD. With no remaining service instances available to process the pending request, Main SD throws a "credit-check port exception" at step 6.

If an SD becomes unavailable, its parent SD (i.e., the domain to which the SD is registered) initiates the failover process by using another SD. When the failing SD is the main SD, however, normal recovery is not possible. In this case, a global problem-determination mechanism may be able to provide recovery. A global algorithm can correlate events from multiple components to pinpoint the cause of error and initiate proper recovery actions. For example, the SD may be down due to a communication link failure. In this case, the failing SD would not be able to generate an event. However, the hardware and business-process error events generated can help identify the cause of the error and trigger a recovery action to correct the link failure or connect the business process to a new SD.

As illustrated in Figure 11, the exception could be caused by the loss of all available service instances. (The SD detects loss of a service instance if the service does not respond to a request within a specified interval or an exception is detected when contacting the service instance. The time-out value—an attribute of the individual service on the SD—either can be set by the administrator or can be determined by the business policy.) Under this circumstance, the

Figure 12 Global problem determination and recovery

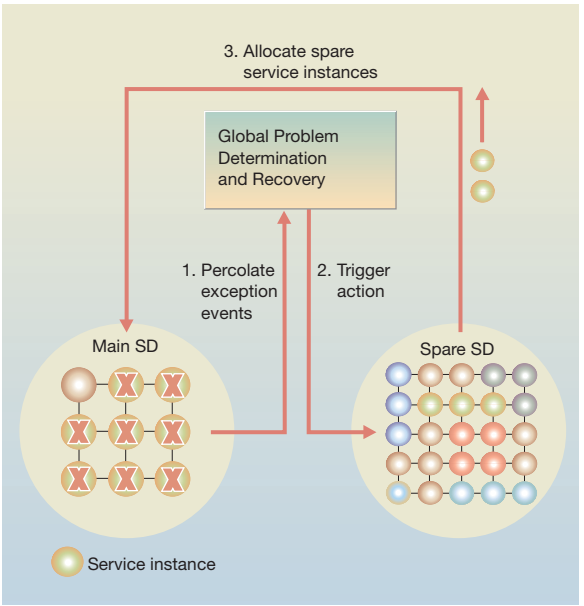
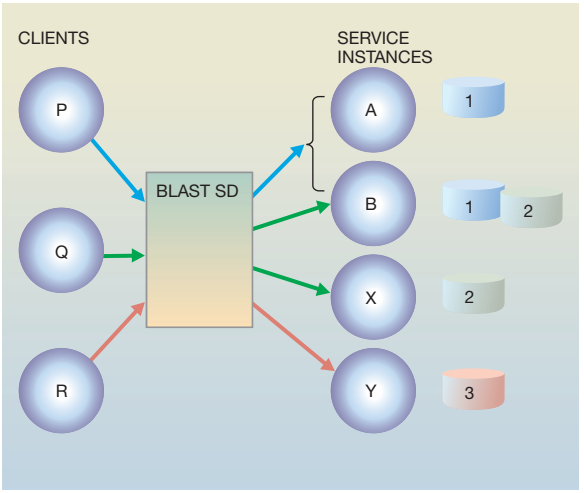


Figure 13 An SD for BLAST services



SD generates a "service feature unavailable" event to a global recovery framework. The framework can then correlate this event with related business-process error events and trigger a recovery action that would enable some spare instances to be allocated to the virtual port.

Figure 12 illustrates a global recovery scenario that involves nodes Main SD and Spare SD and a "global problem determination and recovery" component (GPDR). At step 1, Main SD detects the loss of all available service instances for one of its virtual ports, and it percolates the exception event to GPDR as a "service feature unavailable" event. At step 2, GPDR correlates this event with previously received events, determines that this event is related to another business-process error event, and invokes a recovery action to add service instances from Spare SD. At step 3, the recovery action is carried out and results in the allocation of additional service instances from the spare domain. As a result, Main SD is now able once again to accept service requests from the business process.

The recovery action in this scenario consists of service discovery and registration updating, which are simple functions in the SD model. In fact, the implementation of this scenario, including the integration of a WebSphere Portal Server portlet to control its flow and execution, was completed in two weeks. Without the SD framework, the effort would take longer as it involves managing multiple providers, handling failure recovery, providing backup connections, and so on. In this scenario, it is not sufficient to just bring several providers together. The implementation needs to ensure that the addition and removal of providers is dynamic and the routing selection can be easily managed. It also needs to cope with different interfaces and establish a backup site.

Rule-based service selection. BLAST**[41] (Basic Local Alignment Search Tool) is a set of similarity search programs used in life science to explore all of the available sequence databases to find a specific protein or DNA. We describe a scenario involving the BLAST application to illustrate how to configure SD policy rules. Figure 13 illustrates an SD for BLAST services. The typical request has the form run (sequence, database) where sequence is a character string and database the name of the database to be used. Because the databases are large, application providers usually support only subsets of all the databases. For example, service instance B has database 1 and 2 installed, but service instance A has only database 1 installed. Normally, a user would be required to select the provider that supports a particular database.

BLAST appears to be suited to the SD model. Currently, the database name is a parameter passed to

the command-line call operation. Conventional wisdom suggests the routing and selection logic could be implemented by logic that checks the parameter value, but this would be time consuming and inflexible. Even worse, if the logic depended on the message content, opening every message envelope would be required.

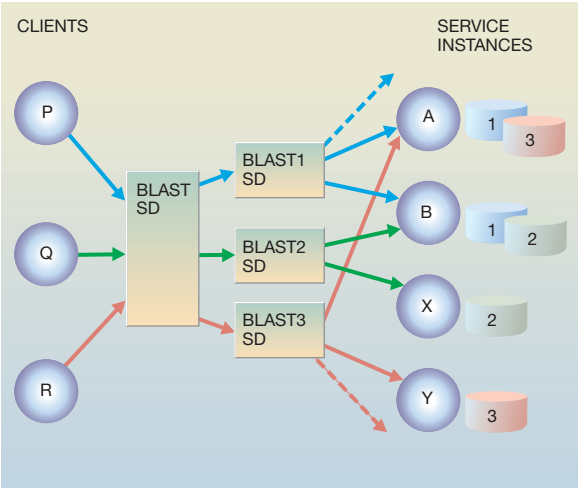
Use of the SD model in this case obviates the need to write code. It provides a comprehensive set of parameters for rule configurations by administrators. One possible solution would be to create different port types that associate BLAST commands with specific databases. For example, service instance B could supply two port types (for databases 1 and 2) while instance A could supply one port type (for database 1). However, there could be many databases, and it may not be realistic to create a port type for each one. Other configuration options are possible. We discuss next several ways to set up an SD for BLAST services: (1) create a different port type for each database, (2) create user and supplier service levels for each database, (3) use a different operation for each database, (4) select service by content, (5) use your own custom selection logic, (6) evolve the SD topology over time, and (7) design selection rules based on performance criteria.

Create a different port type for each database. Creating a different port type for each database is the simplest possible option but is perhaps the most work for the administrator and the least flexible for clients. Service instances are registered under the port types based on the database they support. Users use a specific port type to get to the desired database. In this example, BLAST1, BLAST2, and BLAST3 are created as three port types with registered service instances A and B under BLAST1, service instances B and X under BLAST2, and service instance Y under BLAST3.

A user request to BLAST1, for example, is routed to A or B randomly by default. Instance B needs to implement two versions of its Web services because it has "hard bindings" with two databases. This is a good option for a small number of databases and providers.

Create user and supplier service levels for each database. A second possible solution would be to wrap BLAST into a Web service that mimics the command-line interface. Keeping sequence and database parameters leads to BLAST1 as a single common port type. Using this option, the administrator would create user and supplier service levels as follows:

Figure 14 Refinement of selection rules for BLAST



1. Create one user service level for each database. For example, using the color scheme of Figure 14, create three user service levels: UserBlue, UserGreen, and UserRed. Subscribe all users of database 1 to UserBlue, all users of database 2 to UserGreen, and all users of database 3 to UserRed.
2. Create one supplier service level for each database: supplier levels ServerBlue, ServerGreen, and ServerRed. Register service instances under specific supplier levels, for example, service instances A and B under ServerBlue, service instances B and X under ServerGreen, and service instance Y under ServerRed.
3. Set up service maps to map user service levels to supplier service levels. For example, map UserBlue users to ServerBlue suppliers.

When a user of the UserBlue service level sends a BLAST request, for example, the request will be routed randomly to either A or B. This approach can be used if the databases used by distinct user groups can be mapped to the databases supported by providers almost one to one.

Use a different operation for each database. Very likely most users and suppliers would be interested in multiple databases. Defining service levels for all the possible combinations could be unrealistic. A third possible solution is to introduce new Web services operations that associate BLAST with specific databases. For this scenario, the operation names would be used as a selection factor in service selec-

tion; the Web services wrapper would be constructed and the rules would be configured as follows:

1. Implement an operation for each database used. For example, implement three operations: runBlue, runGreen, and runRed. Implementations for the three operations are the same. Internally, each performs the same task, but the database parameter is set to a specific database.

2. Create one supplier service level for each database. For example, create supplier levels ServerBlue, ServerGreen, and ServerRed. Register service instances under specific supplier levels, that is, service instances A and B under ServerBlue, server instances B and X under ServerGreen, and server instance Y under ServerRed.

3. Create three selection rules using SD-defined affinities and variables:

   a. Rule 1: (AllDay, runBlue, ServerBlue, Round Robin).
   b. Rule 2: (AllDay, runGreen, ServerGreen, Round Robin)
   c. Rule 3: (AllDay, runRed, ServerRed, Round Robin)

   Rule 1, for example, represents a selection policy that applies throughout the day; if the operation specified is runBlue, then supplier ServerBlue is selected. If more than one service instance is available for service, a simple round-robin algorithm is used to select a service instance.

For this option, users choose specific operations associated with the databases. The user service level is not needed. When a user requests BLAST for the runBlue operation, for example, the request will be routed to A or B randomly by default. This option can be used if the databases used by the users are random, that is, not always limited to the databases supported by specific suppliers.

The SD owner has the option not to create the actual Web services wrapper for adding the three operations: runBlue, runGreen, and runRed. Instead, he or she can include the operations in the SD WSDL and use a transformation utility to transform the operations to a generic BLAST Web service with the required parameter settings. The generic BLAST Web service maps identically to the command-line interface. For example, the runBlue operation is transformed to the run operation of the BLAST Web service with a database parameter value of database 1 in the input message. The Web service will invoke the BLAST command-line interface to run the application against database 1.

Content-based selection. This is used on the main domain to select an appropriate secondary domain to receive incoming requests. For this solution, there is a trade-off between improved manageability and the small performance loss associated with examining the message content. However, there is no need to examine the content at each lower-level node; instead, a value representing the message content is inserted into a request context object that is passed from node to node. Using this approach, the administrator instructs the SD to look at specific message parameter values. For example, a rule can be created that is applicable only if the second argument in the message matches 'database1'. This option should be used if the number of databases and BLAST service instances are expected to grow to large numbers that can be better organized into secondary domains for manageability.

Use your own custom selection logic. As a last resort when all other options are unsatisfactory, the domain owner can write a custom directive implementation class to specify the selection logic. This approach may be a common choice for the traditional solution model, but it is least attuned to the on demand environment.

Evolve the SD topology over time. The main guidelines for providing rules are: define configurable patterns, avoid the need to examine the message content, and avoid custom code. To achieve on demand objectives, a customer needs a robust set of building blocks such that solutions can be composed and modified quickly and easily. The options suggested earlier are not exclusive. Customers can build and refine a set of configuration rules at their own pace. As an example, Figures 13 and 14 illustrate that a customer can initially use the simplest but least flexible "one port type per database" configuration approach, but later refine the configuration to use a more flexible multi-operations BLAST SD as the business expands. The configuration illustrated is also an alternative way for organizing service instances in multiple overlapping groups of suppliers to achieve simpler manageability.

Selection rules based on performance criteria. Selection factors can be based on service features and QoS characteristics. Features are the business functions

that an application offers, such as financial service calls, information subscriptions, human-resource services, travel planning, and so forth. The QoS performance characteristics are fairly traditional but offer a different point of view that might be helpful to workload distribution. Not all SDs need very detailed metrics for the fine-grained load-balancing function of transaction throughput and response times; such a function belongs to the operating-system platforms such as WebSphere Application Server, z/OS*, Linux**, and so on. SD metrics are task-oriented and can determine which provider should receive the next service request. For example, an SD could distribute requests to its providers based on the observed queue length of outstanding requests. All things being equal, a provider running on a cluster of four machines should display a shorter queue length than a provider running on a single machine and can thus service more requests than the other.

Discussion and conclusion

The SD's SLA enforcement of dynamic service-level mapping can be viewed as a self-adapting feature of an autonomic system. Specifically, in service-level mapping user SLAs are associated with pools of service instances based on the SLA commitments made by the provider at registration time. As runtime information, such as the distribution of request types, service availability, and turnaround time, becomes available, the pools can be reconfigured based on this information.

Foster suggests that a grid is a system that coordinates resources that are not subject to central control, using standard, open, general-purpose protocols and interfaces, to deliver nontrivial qualities of service.[42] SDs satisfy this definition. The resources managed in an SD are WSDL-defined service instances from multiple providers and are not subject to central control. The providers have their own policies and implementations. SDs are built on Web services and XML standards, and they can also be implemented using the Globus Toolkit**.[43] SDs can be extended to exploit additional OGSA standards. By negotiating with the constituent service instances, the SD node produces a service capability that is superior to the mere collection of services. Selecting a service instance to process a service request can be performed in a variety of ways that range from simple to complex, according to rules that can be based on the states of the requestor and the service provider, the request load, the business relationships between participants, and so on.

Leymann et al.[44] describe a workflow model that applies Web services to business process management and includes a flow definition language and a process choreography engine. The workflow model can be applied to SD nodes directly. In addition the SD model provides simplicity, robustness, and opportunities for lightweight workflow engines. It also allows multiple simple services to be composed into a complex service; in this case, the complex service would be associated with a single port type, and a service requestor would be unaware that the service was actually a composition of simpler services.

SD supports utility computing[45] by generating metering records, enabling usage-based fees to be associated with SLAs, and supporting third-party billing. A hosting manager interface isolates SDs from platform-specific user-management, security, instrumentation, and problem-determination subsystems. The interface provides access to local contracting, identity, and metering services. Dynamic-resource-provisioning technologies, such as IBM Tivoli Intelligent ThinkDynamic* Orchestrator and IBM Tivoli Provisioning Manager, augment the capabilities of SDs with respect to resource provisioning.[46]

Although the examples used in this paper assume no interaction among the SD constituent services, the SD approach also applies to middleware and system functions. Constituent service instances can use SDs for peer communication and state sharing. Dynamic discovery capability between peer SDs can be implemented by policy rules. The SD hierarchy is a logical concept although it could be built from a physical peer-to-peer network.

We believe that environments with tens and hundreds of competing or collaborating suppliers will be a common phenomenon of the future. In order to achieve the full benefit of SOA in such environments, new techniques to overcome complexity are required. In this paper, we showed how the SD model can help reap the benefits of Web services, grid computing, and autonomic computing and provide robust SOA solutions.

Analogous to the concept of grid computing as a hardware resource balancer and job scheduler, the SD manages multiple software sites as a grid:

- Service providers offer services under terms and conditions, and their services are accessed via application calls and not at the hardware level. Examples are life science, stock-portfolio management, financial services, and weather forecasting.

- The SD acts as broker for different service providers and provides a single, logical image that reduces complexity for the users of the service.
- The SD, in a simplified form, is:
  —An interface definition for a service (e.g., WSDL)
  —A set of instances that implement the service
  —A set of policies for directing an operation on the logical service to one of the instances of which it is composed.

The SD handles services as commodities. The incoming service requests are the "bids," and the registered service providers are the "offers." A service domain "clears" the market for all.

Service domains can also exploit a hardware-grid environment when one is available. The SD pilot projects are the first steps to establishing manageable IT service brokerage, thus enabling widespread deployment of on demand solutions.

The journey of SOA from Web services to grid computing has begun. The SD concept, which we believe is key to on demand computing, has manifested itself in recent IBM product announcements.

Future work will be directed to helping the adoption of products that exploit WSRF, system- and data-oriented domains, and cross-domain orchestration, to conducting more customer engagements, and to enhancing existing standards. Work is also planned to develop additional tools that are needed for implementing robust and efficient domains with minimal effort. In addition, we see a continuing need to enhance the performance of the system, improve problem determination collaboration across multiple service sources, and push for the pervasive use of the SD model. With enhancements such as these, we expect that SDs will become the cornerstone of on demand service integration. The future is challenging but also exciting.

Acknowledgments

We gratefully thank Dr. George Wang, Vice President, System Performance, and former Director of IBM China Software Development Lab and his grid-computing team for their enthusiasm and dedication in support of the customer-engagement pilot projects for service domains in China.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Massachusetts Institute of Technology, Sun Microsystems, Inc., Dow Jones & Company, Inc., National Association of Security Dealers, Inc., The Financial Times Ltd. and the London Stock Exchange, Linus Torvalds, the National Library of Medicine, or the University of Chicago.

Cited references

1. *Communications of the ACM* **46**, No. 10 (October 2003), Special Section: Service-Oriented Computing.
2. *Web Services Activity*, World Wide Web Consortium (W3C), http://www.w3.org/2002/ws/.
3. *SOA and Web Services*, developerWorks, IBM Corporation, http://www.ibm.com/developerworks/webservices/.
4. WS-I, Web Services Interoperability Organization, http://www.ws-i.org/.
5. Global Grid Forum (GGF), http://www.gridforum.org/.
6. The Globus Alliance, http://www.globus.org/.
7. IBM Grid Computing, IBM Corporation, http://www.ibm.com/grid/.
8. Autonomic Computing, IBM Research, IBM Corporation, http://www.research.ibm.com/autonomic/.
9. The World Wide Web Consortium (W3C), http://www.w3.org/.
10. Organization for the Advancement of Structured Information Standards (OASIS), http://www.oasis-open.org/.
11. *Web Services Description Language (WSDL)*, World Wide Web Consortium (W3C), http://www.w3.org/TR/wsdl.
12. *W3C—Extensible Markup Language (XML)*, World Wide Web Consortium (W3C), http://www.w3.org/XML.
13. *SOAP Version 1.2 Primer*, World Wide Web Consortium (W3C), http://www.w3.org/TR/2003/REC-soap12-part0-20030624/.
14. *HTTP - Hypertext Transfer Protocol*, World Wide Web Consortium (W3C), http://www.w3.org/Protocols/.
15. *OASIS UDDI Specifications TC—Committee Specifications*, Organization for the Advancement of Structured Information Standards (OASIS), http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm#uddiv3.
16. T. Appnel, *An Introduction to WSIL*, O'Reilly & Associates (October 16, 2002), http://www.onjava.com/pub/a/onjava/2002/10/16/wsil.html.
17. M. J. Duftler, N. K. Nirmal, A. Slominski, and S. Weerawarana, "Web Services Invocation Framework (WSIF)," *OOPSLA 2001 Workshop on Object-Oriented Web Services*, ACM, New York (2001), http://www.research.ibm.com/people/b/bth/OOWS2001.html.
18. R. Kraft, "Designing a Distributed Access Control Processor for Network Services on the Web," *Proceedings of the ACM Workshop on XML Security*, Nov. 22, 2002, ACM, New York (2002).
19. I. Foster, *WS-Resource Framework: Globus Alliance Perspectives*, Presented at Globus World, San Francisco, CA, January 20, 2004, http://www.globusworld.org/.
20. D. Sabbah, *Bringing Grid and Web Services Together*, Presented at Globus World, San Francisco, CA, January 20, 2004, http://www.globusworld.org/.
21. I. Forster, C. Kesselman, J. M. Nick, and S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems," The Globus Alliance, http://www.globus.org/research/papers/ogsa.pdf.
22. The IBM On Demand Operating Environment, IBM Corporation, http://www.ibm.com/software/info/openenvironment/index.html.

23. Y.-S. Tan, B. Topol, V. Vellanki, and J. Xing, "Business Service Grid, Part 1: Introduction," developerWorks, IBM Corporation (February 1, 2003), http://www.ibm.com/developerworks/library/gr-servicegrid/index.html.
24. *Emerging Technologies Toolkit*, alphaWorks, IBM Corporation, http://www.alphaworks.ibm.com/tech/ettk.
25. *J2EE Tutorials and Code Camps*, Sun Microsystems, http://java.sun.com/j2ee/learning/tutorial/index.html.
26. D. F. Ferguson and R. Kerth, "WebSphere as an e-Business Server," *IBM Systems Journal* **40,** No. 1, 25–45 (2001).
27. The Apache Software Foundation, http://www.apache.org.
28. OGSA-WG, Open Grid Services Architecture, Global Grid Forum (GGF), http://forge.gridforum.org/projects/ogsa-wg/document/draft-ggf-ogsa-spec/en/13.
29. D. F. Ferguson, T. Storey, B. Lovering, and J. Shewchuk, "Secure, Reliable, Transacted Web Services: Architecture and Composition," IBM Corporation, http://www.ibm.com/software/solutions/webservices/pdf/SecureReliableTransactedWSAction.pdf.
30. Microsoft .NET, Microsoft Corporation, http://www.microsoft.com/net/default.asp.
31. J. McCarthy, *Reap the Benefits of Document Style Web Services*, developerWorks, IBM Corporation, http://www.ibm.com/developerworks/webservices/library/ws-docstyle.html.
32. Y.-S. Tan, B. Topol, V. Vellanki, and J. Xing, "Business Service Grid, Part 2: Implementing a Business Service Grid," developerWorks, IBM Corporation, http://www.ibm.com/developerworks/library/gr-servicegrid2/index.html.
33. Y.-S. Tan, B. Topol, V. Vellanki, and J. Xing, "Business Service Grid, Part 3: Setting up Rules," developerWorks, IBM Corporation, http://www.ibm.com/developerworks/grid/library/gr-servicegrid3.html.
34. Y.-S. Tan, B. Topol, V. Vellanki, and J. Xing, "Business Service Grid, Part 4: Service Domain Deployment," developerWorks, IBM Corporation, http://www.ibm.com/developerworks/library/gr-servicegrid4/index.html.
35. Y.-S. Tan, B. Topol, V. Vellanki, and J. Xing, "Business Service Grid, Part 5: Setting up Contracts," developerWorks, IBM Corporation, http://www.ibm.com/developerworks/grid/library/gr-servicegrid5.html.
36. Y.-S. Tan, B. Topol, V. Vellanki, and J. Xing, "Business Service Grid, Part 6: In Operation," developerWorks, IBM Corporation, http://www.ibm.com/developerworks/grid/library/gr-servicegrid6.html.
37. Y.-S. Tan, B. Topol, V. Vellanki, and J. Xing, "Business Service Grid, Part 7: Keeping Informed," developerWorks, IBM Corporation, http://www.ibm.com/developerworks/library/gr-servicegrid7/index.html.
38. *JavaServer Pages Technology - White Paper*, Sun Microsystems, http://java.sun.com/products/jsp/whitepaper.html.
39. R. Will, S. Ramaswamy, and T. Schaeck, "WebSphere Portal: Unified User Access to Content, Applications, and Services," *IBM Systems Journal* **43,** No. 2, 384–419 (2004).
40. IBM and China's Ministry of Education Launch "China Grid", IBM Press Release (October 13, 2003), http://www.ibm.com/grid/grid_press/pr_1013.shtml.
41. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic Local Alignment Search Tool," *Journal of Molecular Biology* **215,** No. 3, 403–10 (October 5, 1990).
42. I. Foster, "What is a Grid? A Three Point Checklist," *Grid Today* **1,** No. 6 (July 22, 2002), http://www.gridtoday.com/02/0722/020722.html.
43. *The Globus Toolkit*, The Globus Alliance, http://www-unix.globus.org/toolkit/.
44. F. Leymann, D. Roller, and M.-T. Schmidt, "Web Services and Business Process Management," *IBM Systems Journal* **41,** No. 2, 198–211 (2002).
45. *IBM System Journal* **43,** No. 1 (2004), Special Issue: Utility Computing.
46. E. Manoel, S. C. Brumfeld, K. Converse, M. DuMont, L. Hand, G. Lilly, M. Moeller, A. Hemati, and A. Waisanen, *Provisioning On Demand: Introducing IBM Tivoli Intelligent ThinkDynamic Orchestrator*, IBM Redbooks (2004), SG24-8880-00, http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/sg248888.html?Open.

Accepted for publication July 5, 2004.

Yih-Shin Tan *IBM Software Group, 4205 South Miami Boulevard, Durham, NC 27709 (ystan@us.ibm.com).* Yih-Shin Tan, a Senior Technical Staff Member, is the lead architect for Web services and grid computing in the System House Design and Technology Group. He pioneered the concept of the service domain and played a key role in developing the service domain component of the Emerging Technology Toolkit on the IBM alphaWorks™ Web site and in applying this technology to several customer pilot projects. He is an IBM Master Inventor and a member of the IBM On Demand Operating Environment Architecture and Technology team.

Vivekanand Vellanki *IBM Software Group, PO Box 12195, 3039 Cornwallis Road RTP, NC 27709 (vellanki@us.ibm.com).* Dr. Vellanki is involved in advanced technology projects in the areas of Web services, grid computing, and autonomic computing. He received a Ph.D. in computer science from the Georgia Institute of Technology in 2001. His interests include distributed computing, Web servers, and peer-to-peer computing.

Jie Xing *IBM Software Group, PO Box 12195, 3039 Cornwallis Road RTP, NC 27709 (jiexing@us.ibm.com).* Dr. Xing, an advisory software engineer, is currently involved in advanced technology projects in the areas of Web services, grid computing, and autonomic computing. He received a Ph.D. in operations research and computer science from North Carolina State University in 2000. His interests include multiagent systems, distributed systems, and workflows.

Brad Topol *IBM Software Group, 4205 South Miami Boulevard, Research Triangle Park, North Carolina 27709 (btopol@us.ibm.com).* Dr. Topol received B.S. and M.S. degrees in mathematics and computer science from Emory University in 1993 and a Ph.D. degree in computer science from the Georgia Institute of Technology in 1998. He joined IBM in 1998. As a member of the SWG System House Design and Technology Group, his current focus is on integrating IBM software products within IT solutions. He is also involved in advanced technology projects in the areas of autonomic computing and Web services. In 2000, he received an IBM Outstanding Technical Achievement Award for contributions to the WebSphere Transcoding Publisher product. He is an IBM Master Inventor and a member of the IBM Autonomic Computing Architecture Board.

Gary Dudley *IBM Software Group, PO Box 12195, 3039 Cornwallis Road RTP, NC 27709 (dudleyg@us.ibm.com).* Mr. Dudley holds a B.S.E degree from Duke University and an M.S. degree from North Carolina State University, both in electrical engineering. Prior to joining the IBM Software Group, he worked in storage systems and networking architecture. Currently, he is actively involved in advanced technology projects in the areas of Web services, aspect-oriented programming, and autonomic computing.