Using a utility computing framework to develop utility systems

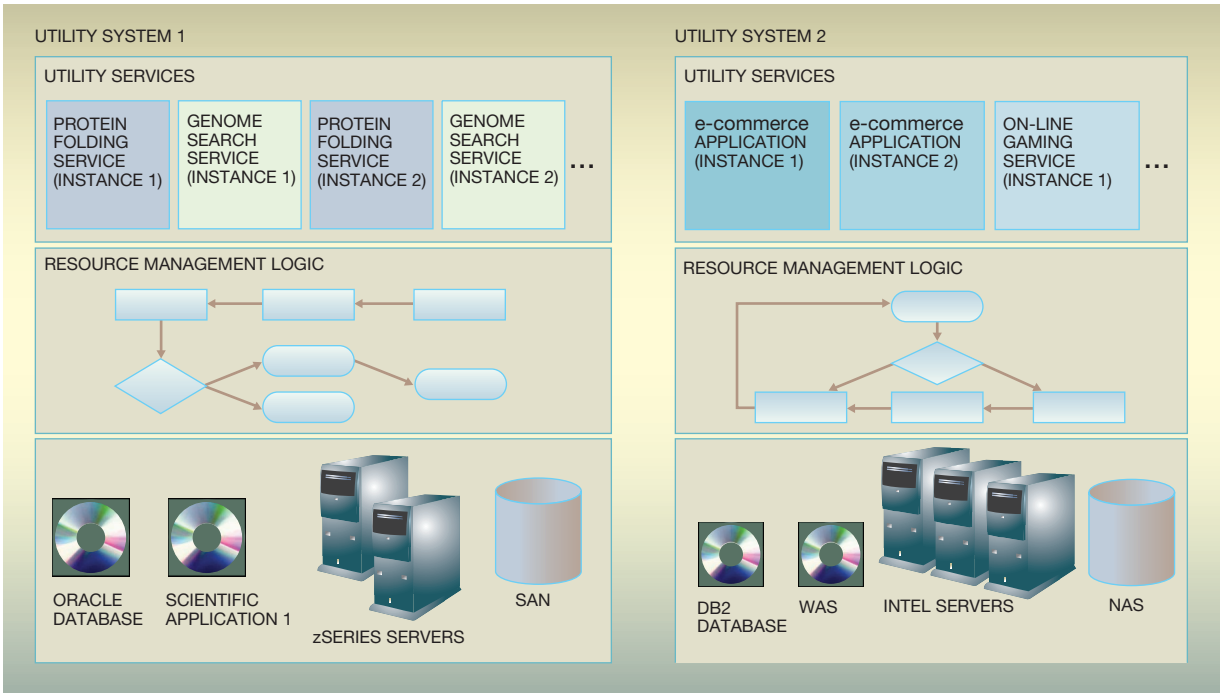
by T. Eilam
K. Appleby
J. Breh
G. Breiter
H. Daur
S. A. Fakhouri
G. D. H. Hunt
T. Lu
S. D. Miller
L. B. Mummert
J. A. Pershing
H. Wagner

In this paper we describe a utility computing framework, consisting of a component model, a methodology, and a set of tools and common services for building utility computing systems. This framework facilitates the creation of new utility computing systems by providing a set of common functions, as well as a set of standard interfaces for those components that are specialized. It also provides a methodology and tools to assemble and re-use resource provisioning and management functions used to support new services with possibly different requirements. We demonstrate the benefits of the framework by describing two sample systems: a life-science utility computing service designed and implemented using the framework, and an on-line gaming utility computing service designed in compliance with the framework.

We describe in this paper a *utility computing framework* (framework, for short) that consists of a component model, a methodology, and a set of tools and common services for building utility computing systems. A *utility computing system* (also referred to as a *utility system*, or simply a *utility*) is a system that can automatically create and manage multiple *utility computing services* (*utilities services*, for short) on a shared infrastructure. The infrastructure consists of pools of hardware resources, such as servers, storage, and network appliances, as well as software resources, such as operating systems, middleware, and applications. The utility services that can be created are not limited to a specific domain, and may range from e-commerce services, to scientific applications, to on-line gaming. A utility system ensures the smooth operation of the supported services by dynamically adjusting the allocation of resources. If the total demand for services exceeds the capacity of available resources, the utility system may dynamically procure and configure additional resources in order to support service level commitments. The utility system makes resource allocation and configuration decisions based on factors such as performance monitoring, SLA (service level agreement) goals, business objectives, or human interaction.

Utility computing systems differ in the types of resources used, the topology of the network connecting these resources, the services offered, and the business and operational constraints that govern their operations. Figure 1 shows two examples of utility systems. Utility system 1 offers and manages services that perform scientific computations and queries. Instances of services, such as the protein folding service, are illustrated at the upper tier in Figure 1. Utility system 1 uses an infrastructure that includes IBM eServer* zSeries* servers, an Oracle** database, a SAN-(storage area network) based storage system, and a scientific application (these are shown at the bottom tier in Figure 1). Utility system 2 offers e-commerce services and on-line gaming services. It uses Intel** servers, an IBM DB2 Universal Database*, IBM WebSphere* application servers (WAS), and a NAS (network-attached storage) component.

Figure 1 Two examples of utility computing systems



The middle tier in the illustrations of Figure 1 represents the resource management logic of a utility system. These are idealized diagrams that represent resource allocation knowledge—what to allocate, when to allocate, and how to allocate available resources. The resource management logic also includes functions such as performance monitoring, event correlation, problem determination, and optimization. As Figure 1 illustrates, the resource management logic can vary depending on the utility system. Here are some specific examples.

- The logic required to allocate logical partitions on an IBM zSeries machine (a logical partition, also known as an LPAR, represents a share of the machine's resources) is different from the logic required to allocate an entire Intel machine. Thus, the sequence of steps required to allocate a resource varies, depending on the resource type, even for resources of the same family.
- An e-commerce service may require a multitier network topology with firewalls between tiers to support strict security procedures. The resources needed include load balancers, Web servers, Web application servers, and databases. For an on-line gaming service a simple one-tier network topology may suffice, as security is more relaxed. The resources needed are servers running various proprietary gaming applications. Thus, the required resource types and their configuration, as well as the network topology, differ for different services.
- CPU utilization is a good performance indicator for a scientific computation, whereas Web server response time should be monitored for an e-commerce service. Thus, the set of performance parameters to be monitored is different for different services.

When a new service type is introduced, the resource management logic must be modified to support its requirements, structure, and operational constraints. Similarly, when a new resource type (e.g., a new computing platform) is incorporated, or an existing resource is upgraded, changes are required to the infrastructure that must be reflected in the management logic.

Thus, building new utility systems, or changing existing ones, is labor-intensive and error prone—a

daunting task. The task is complicated by the many interdependencies between resources, which are often hidden and not well understood, by the combinatorial large number of possible allocations and configurations of a given set of resources, by the variety of possible services with different requirements, and by the many and rapidly evolving types of hardware and software resources. Without a conceptual model, a suitable methodology, and a good set of tools, developing new utility systems, or changing existing ones, is an expensive and time-consuming process.

The utility computing framework described in this paper addresses this challenge. The framework does not assume a set of resource types, a network topology, or specific types of utility services. Indeed, it can be used to build utility systems that support a variety of services using a variety of resources. Moreover, a utility system that is built using the framework can be modified to accommodate changes in infrastructure or services offered.

Under the framework, the creation of a new utility service requires that the service be defined using a formal XML-(eXtensible Markup Language) based specification language developed for this purpose. A component, named the Planner, translates the description to a set of plans (workflows) that implement the operations performed on the service. Although the set of plans varies depending on the service definition, it always includes a "create" plan and a "delete" plan; when executed, these create (or terminate) an instance of the service. Other common plans are "add resource" and "remove resource." These plans are represented in Figure 1 by the idealized diagrams labeled "resource management logic."

The Planner makes use of a *parts catalog*, where *parts* are entities describing resource-related capabilities at various levels of abstraction. At the lowest level of abstraction, a part may represent capabilities of physical resources, whereas higher level parts may make use of lower level parts to define more complex entities, such as a secure Web server.

As an example, suppose a new utility service is to be created and offered to customers; a service that provides Web sites capable of dynamically garnering additional resources in response to an increasing load. A utility service expert defines the service by creating a part that represents this new service and that is built using existing parts, such as "Web server," "load balancer," and "database." This service definition is then "published" and accessible to customers. Suppose a customer interested in outsourcing an e-commerce Web site subscribes to this service. Then, the customer, using a *service template*, may specify values for a number of parameters in the service definition, such as the initial number of Web servers to be provided.

After the subscription is received and validated, a service instance for the dynamic Web site service has to be created. First, an entity known as a utility service controller (USC) is set up; it tracks the state of the service instance, including handles to resources allocated to it. Next, the Planner is invoked; from a service template it generates a set of plans, including the "create" plan, and a plan to increase service capacity according to the load. Then, the "create" plan is executed; resources (load balancers, Web servers, databases) are allocated and configured. The plan may include steps requiring a response from the customer, such as uploading data to a database.

The service instance is now operational, and the public may access the new Web site. The Runtime Management component continuously monitors the load on the Web site, and when the response time exceeds a specified threshold defined in the subscription agreement, the plan to increase service capacity is invoked and executed.

In addition to describing the architecture and the design of the utility computing framework, we also discuss two utility systems based on the framework: the Life Science Computing Service (LSCS), and the on-line gaming service. We designed and implemented the LSCS using a preliminary version of the framework; we designed the on-line gaming service in compliance with the framework but have not implemented it yet.

Due to the enormous scope of the subject, and to facilitate the exposition of the material, we present only those elements of the framework that we actually used in our implementations. In particular, we focus on distributed management, on provisioning and configuration of resources, and on the process of assembling the resources and functions for a new service from existing building blocks. We defer other aspects of the framework, such as monitoring, metering, runtime management, and tuning of resources, to future publications.

The LSCS utility system offers services consisting of compute-intensive scientific applications, such as protein folding calculations for life-science research. Servers for these services are typically allocated for periods of weeks—the allocation being triggered by an explicit request from the customer. We chose the LSCS system as our first implementation because of its simplicity—there is no need for resource performance monitoring or for load-dependent dynamic allocation of resources.

The on-line gaming utility system provides on-line gaming services implemented using servers running various proprietary gaming applications. Resources are allocated among game instances based on load, typically represented by the number of participating players.

Related work. Early work on utility computing systems provided a single service type that used a small number of resource types, such as servers, with fixed assumptions about network infrastructure. The Océano system provided dynamic provisioning of front-end servers for Web sites consisting of a load balancer, a variable number of front-end servers, and a fixed number of back-end database servers.[1] Initially, a minimum set of servers was allocated to each Web site service subscriber, with unassigned servers constituting a free pool.[1] Additional front-end servers were allocated automatically, in response to events triggered by server workload or response time measurements. Isolation between subscribers was achieved using physically separate servers and virtual local area networks (VLANs) in a reconfigurable switched-network infrastructure. The Muse system also allocated server resources in response to load, but modeled both the value of allocating resources to each customer and the cost of employing those resources, based on energy cost.[2]

The Racquarium system provided a variety of multitier Web sites through application complexes, described by a software plug-in component which defined the tier structure of the complex, the content of each tier, whether it could be modified by adding application instances, and under what circumstances these modifications could occur (e.g., excess server load).[3] Racquarium did not provide network isolation. Provisioning support was limited to configuration of servers preloaded with operating system and application software. Application complexes were configured via a GUI (graphical user interface).

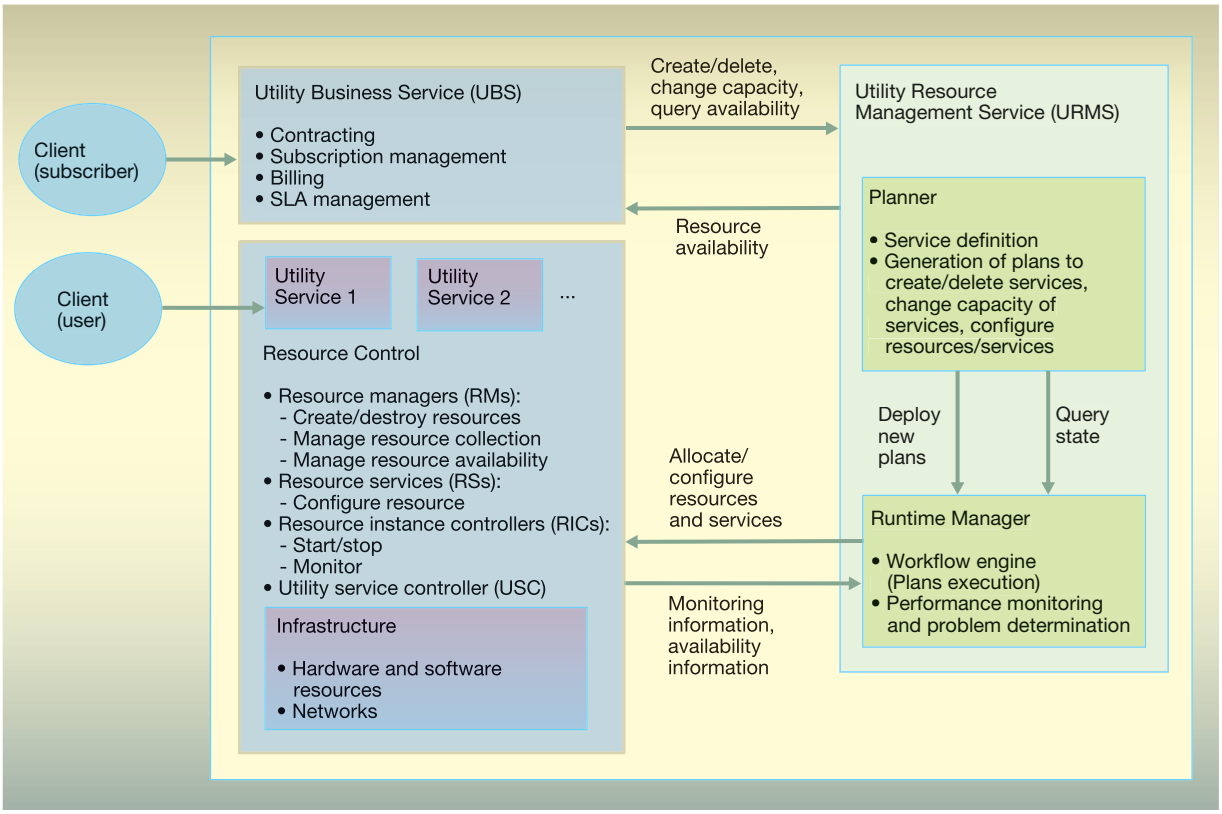
The cluster-on-demand (COD) system provided servers for general use by dynamically partitioning a physical server cluster into virtual clusters.[4] Each virtual cluster contained server and storage resources in a private IP address block on its own VLAN, configured with a user-specified software environment. New environments were constructed using ad hoc tools and captured as partition images. Management of the cluster was hierarchical, with virtual cluster managers negotiating resource allocations with a single global COD resource manager.

A growing number of industrial products aim to provide multitier application environments over a physical infrastructure consisting of a variety of resources, such as Hewlett Packard's Utility Data Center,[5] ThinkDynamics,[6] Sun Microsystems' N1[7,8,9] and Jareva's OpForce suite.[10] They vary in many respects: the types of resources provided (e.g., servers and storage); operating systems and middleware supported; characteristics of the network infrastructure (e.g., whether network isolation is provided by VLAN); level of resource virtualization; level of monitoring support (e.g., resource usage, failure detection, SLA, threshold-based alerting); level of support for resource discovery; level of support for modifying the application environment and whether modifications can occur automatically (e.g., triggered by SLA); and the extent to which the products can (or must) be customized to fit pre-existing hosting center infrastructures. For example, a customer can define a two-tiered structure "virtual server farm" that lays out the desired resources and network topology to be configured on request.

At the application layer, SmartFrog[11] is a framework for deployment and management of distributed applications. An application is described as a collection of related, reusable components, which may represent resources or subsystems. The description includes dependency information to ensure that, for example, components are started in the correct sequence. Once deployed, applications may be monitored, and actions, such as automatic failover or restart, may be specified in case of component or resource failures. SmartFrog is not used for low level resource configuration tasks such as installing operating systems on servers, but for higher level application-specific configuration. The parts catalog and the Planner components of our framework are similar in approach to SmartFrog.

Several systems provide utility functionality in wide area networks. Opus[12] is an overlay peer utility ser-

Figure 2 The architecture of the utility computing framework



vice that provides common functions for hosting distributed applications over wide area networks. Given a specification of application performance and availability targets, Opus allocates nodes to applications and modifies allocations dynamically, based on observed conditions and SLAs. For example, if demand for an application increases in a particular area of the network, Opus may allocate additional nodes to the application in that area.

The grid was initially developed to enable sharing of computing resources over wide area networks for scientific applications.[13] More recently, the Open Grid Services Architecture (OGSA) is an effort to generalize grid technologies to support the assembly and maintenance of collections of resources for distributed services.[14] A related effort, GARA, describes an advance resource reservation system using structures similar to those in our framework.[15] Our work relies on OGSA concepts and is compliant with the architecture.

The rest of the paper is structured as follows. In the next section we present an overview of the framework and discuss the main concepts behind the technology. In the following two sections we describe the two implementations based on the framework: the LSCS utility system and the on-line gaming utility system. In the last section we present our conclusions based upon the work presented here.

Overview of the utility computing framework

The architecture of the framework, illustrated in Figure 2, includes three main components: Utility Business Service (UBS), Utility Resource Management Service (URMS) and Resource Control. UBS is the gateway to the utility. It includes such functions as contracting, subscription management, SLA management, and billing. URMS is responsible for provisioning and managing utility services. Resource Control directly controls resources and utility service in-

stances. In this paper we focus on URMS and Resource Control components, with the emphasis on the functions that were used to develop the sample utilities.

In Resource Control, resource services (RS), resource managers (RM), and resource instance controllers (RIC) are resource control entities that are used to allocate and manage resources. Specifically, an RM manages a collection of resources of the same type, their "creation," allocation, and availability. Every resource is represented and controlled by a RIC, which encapsulates the state of the resource, and which can be used to start it, stop it, or monitor its performance. Finally, an RS is a helper function that configures a single resource or a combination of resources.

A utility service controller (USC) is used to represent and control a utility service instance. The USC encapsulates its state and the operations that can be performed on it. The state of a service instance includes the set of handles to the resources currently allocated to it. The operations that can be performed on it include creating and destroying the service instance as well as changing its capacity (by allocating or deallocating resources).

The Utility Resource Management Service (URMS) creates and manages utility service instances. To create a utility service instance, a combination of resources must be created, allocated, and configured. To manage a utility service instance over time, allocated resources must be monitored and reconfigured. The URMS includes two main components: the Planner and the Runtime Manager. The Planner uses the definition of the service to generate plans for creating and managing the utility services. The plans, which are realized as workflows, consist of sequences of instructions that perform operations on the control entities (RMs, RSs, and RICs). There are two mandatory plans that are always generated for every service instance: "create" and "delete." Other plans, for example, a plan to change the capacity of a service instance, are optional and depend on the definition of the service. The Runtime Manager includes a plan executor (workflow engine) and a rule-based correlation engine that is activated by events detected through monitoring. An event may trigger the processing of a rule, leading to the execution of a plan to allocate or configure resources.

In the Planner, the artifacts known as parts are used to formally represent capabilities of resources. Higher-level parts, such as a secure Web server, are used to represent complex entities and functions that can be realized using lower-level parts, such as physical resources. Thus, the parts serve as the vocabulary that is used to define utility services.

In the rest of this section we provide additional details on the structure and operation of the URMS and Resource Control.

Resource control. Resource managers (RMs), resource services (RSs), and resource instance controllers (RICs) are collectively termed *resource controllers*. They encapsulate functions that control resources, including their creation, allocation, configuration, and monitoring. The operation of these functions is exposed by using the OGSA WSDL (Web Services Description Language) mechanism (see Reference 16). When workflows are executed by the workflow engine, the resource controllers are activated; resources are created, allocated, and configured.

Resource managers. Every resource manager (RM) encapsulates the logic to create and allocate a particular type of resource. Resources can be physical (e.g., an IBM eServer xSeries* server) or virtual (e.g., an LPAR on an IBM eServer zSeries server). A resource may be allocated to a service instance or unallocated (free). Free resources are kept in a logical structure called a free-pool.

The two main functions of the RM are managing the free pool and managing (i.e., allocating and reclaiming) resources. Free-pool management includes tracking the available resources and selecting resources for allocation to a service instance. A resource reservation system is used to support this function. When the URMS needs to allocate a resource, it first has to be reserved. The RM has two mandatory operations for reserving resources: findAvailability and commitResource. The operation findAvailability is used to determine the time slots when a resource with certain specified properties is available for allocation. It is used when an advanced reservation is needed or when a combination of resources is concurrently needed. The operation commitResource is used to reserve a resource with certain properties for a particular time slot. It returns a reservation ticket (rsvTKT), which is used later for allocating the resource. The operation fails if no resource that satisfies the requirements is available.

To allocate a resource to a service instance the URMS invokes a create operation on the RM. The parameters of the create operation include the reservation ticket returned from a previously invoked commitResource operation. If the resource exists, create simply returns a handle to a RIC representing the selected resource. The RIC provides a mechanism to access and manage the resource (see the section "Resource instance controller" below). The resource may be further configured by URMS as part of the utility-service instance-creation process. If the resource does not exist, the RM has to go through the resource creation process. Examples include the creation of a VM guest machine on a zSeries machine or the setup of a VLAN on a network switch. In all these cases, the resource may not exist prior to the create request. It is the responsibility of the RM to create the resource before a handle to it is returned and the operation is completed. Whether the resource is actually created or allocated from the free-pool is transparent to URMS.

Resource services. A resource service (RS) encapsulates low-level configuration operations on a single or multiple resource types. These configuration operations are used in the process of creating or destroying a service instance or when its resources are allocated and deallocated. As an example consider a service instance that includes a secure network implemented with virtual LANS (VLANS). When a server is added to such an environment, it is necessary to assign the switch port connected to the server's network interface card (NIC) to the appropriate VLAN. This is done by using a switch configuration RS with an assign operation that receives two parameters: a switch port ID and a VLAN ID. As another example, consider a service instance that includes a shared file system and a set of servers. For the servers to be able to use the shared file system, an RS equipped with a mount operation enables the mounting of the shared file system on individual servers.

Resource instance controllers. A resource instance controller (RIC) is used to start, stop, and monitor a resource instance. It is created by a create operation on an RM, issued when a resource is allocated to a service instance by URMS. A handle to the RIC is returned from the call and kept in the corresponding USC. Other common operations exposed by a RIC include operations to query the state of a resource, or to reconfigure it. For example, a Web server resource is represented and controlled by a Web server RIC that can start, stop, or query it (e.g., whether it is "started," "starting," "stopped," or "failed").

Utility service controller. A utility service controller (USC) represents and controls a single utility service instance. The state of the service instance includes the set of handles to the resources that are currently allocated to it. The operations that can be performed on a USC are the invocation of plans (workflows) generated by the Planner. As mentioned, create and delete are mandatory operations; additional operations are known as *custom* operations. The USC provides interfaces to trigger the execution of the workflows or to query its state.

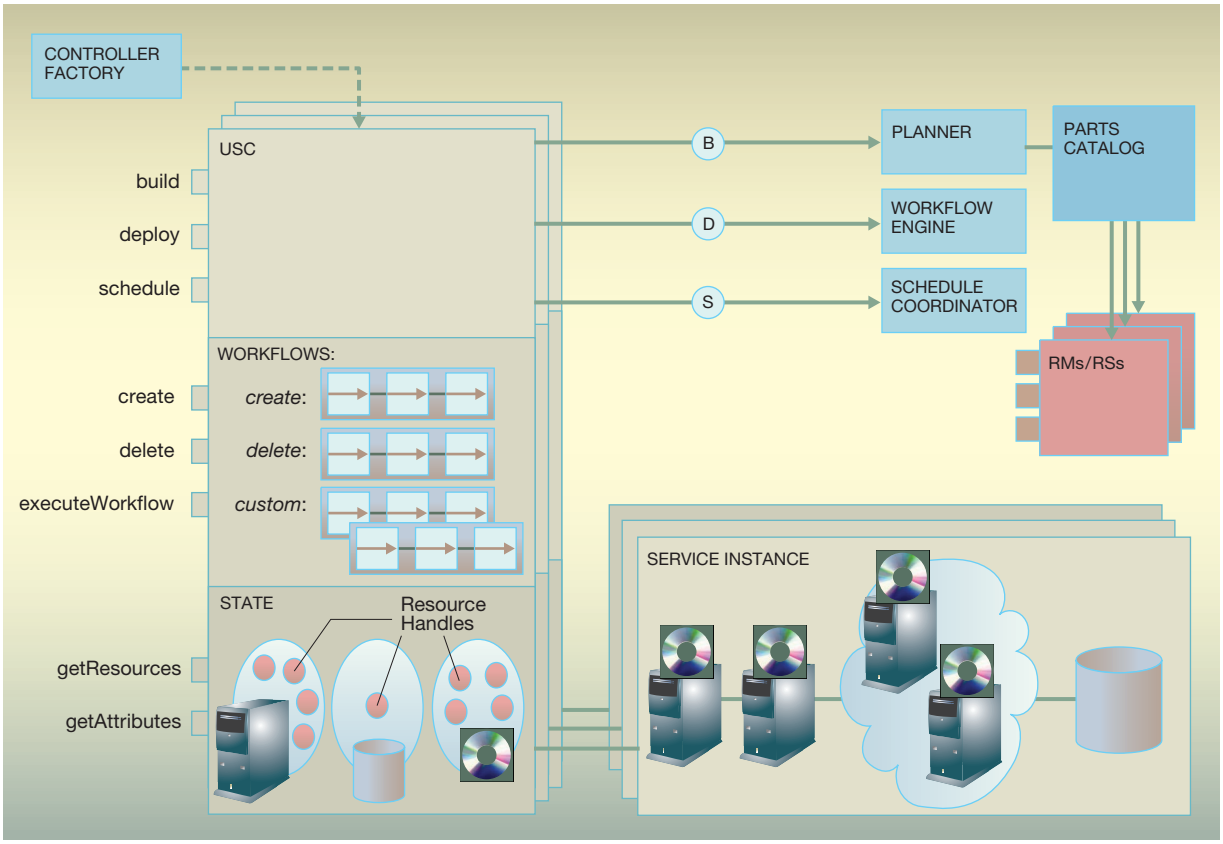
Figure 3 illustrates the operation of the USC and the components it interacts with. Upon instantiation by the framework the USC is empty, with no associated workflows. Then, the build operation on the USC invokes the Planner in order to generate the workflows (see the section "Planner"). This is depicted in Figure 3 as the arrow labeled B; the Planner returns the workflows. Next, the deploy operation is called to deploy the workflows on the workflow engine (see the subsection "Plan Executor" later in this paper). This is depicted in Figure 3 as the arrow labeled D. Finally, the operation schedule schedules the creation of the service instance using a Schedule Coordinator (see the subsection "Schedule Coordinator"). This is depicted in Figure 3 as the arrow labeled S. Note that the service instance itself has not been created yet. When the create operation is invoked, at the time determined by the Schedule Coordinator, the *create* workflow is launched and the service instance is created; using the resource controllers, the required resources are allocated and configured. Handles to these resources are kept in the USC and can be retrieved using the getResources operation.

Additional operations on the USC include executeWorkflow, which is used to invoke by name any of the custom workflows, and the getAttributes operation, which returns values of attributes from an execution of custom workflows.

It is important to understand that the USC does not have to be re-implemented when a new type of utility service instance is defined. It is a generic component that is instantiated automatically by a call to a controller factory (an OGSA factory described in Reference 13) and customized later by the Planner, based on the definition of the utility service.

Parts and topology trees. Resources and their capabilities are defined and represented as parts. Service instances are represented using a graphical concept known as a *topology tree*, whose nodes are parts.

Figure 3 The operation of the utility service controller (USC)



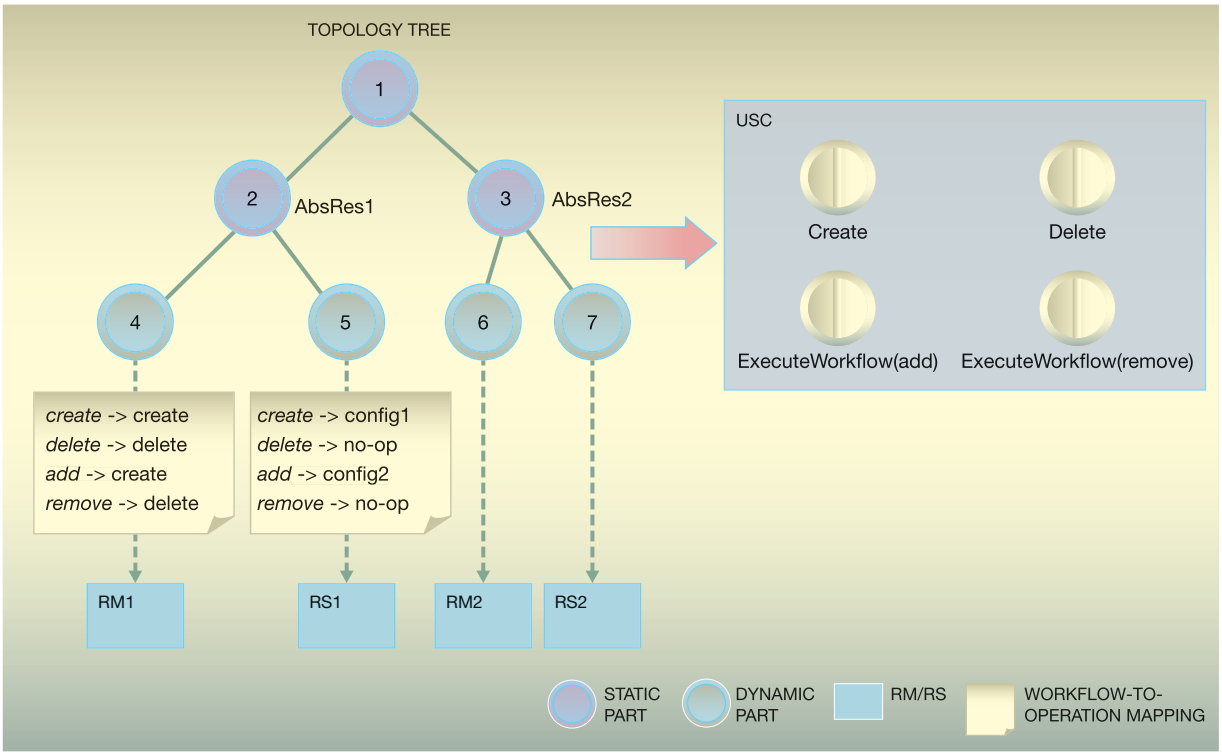
Parts catalog. The parts catalog is a collection of parts that can be accessed programmatically by the Planner, or manually by an expert working on defining a new service or developing a new part. The mechanism for accessing parts is not covered in this paper. A part is an XML-based definition describing a resource, a combination of resources, or operations on resources.

There are currently two types of parts: static parts and dynamic parts. The static parts represent an aggregation of lower-level parts, static or dynamic. The "dynamic" aspect of dynamic parts originates in components with running code, specifically RM or RS operations. Dynamic parts provide a mapping from global operations (workflows) on service instances to local operations, that is, operations on RMs and RSs. Specifically, if workflow X is mapped to RM/RS operation Y then an invocation of Y is included in workflow X. An example of a global operation is the *create* workflow to create a utility service instance. As part of the creation process, many local operations need to be performed. One such operation may be to configure a switch; in this case there will be a part with a mapping from the global operation create to a local operation config on a switch RS.

Topology trees. Utility services are defined using a topology tree whose nodes are parts. The root of the tree is a static part that is associated with the utility service template. The descendants in the tree are either static parts, in which case they are intermediate nodes (non-leaves), or dynamic parts, in which case they are leaves of the tree and are associated with specific RMs and RSs.

The Planner maps a service template to a set of workflows for provisioning and managing the utility service instances corresponding to the template. The process of workflow generation consists of two steps.

Figure 4 An example of a topology tree and the corresponding USC



First, the topology tree is created and then the topology tree is used to generate the set of workflows (see the following section, "Planner"). The generated set of workflows includes the mandatory *create* and *delete* workflows, and possibly some custom workflows. As previously mentioned, all of these workflows are encapsulated in the USC and will be launched by invoking the corresponding USC operations.

Figure 4 shows an example of a topology tree and the corresponding USC. The topology tree includes static parts 1, 2, and 3, and dynamic parts 4 through 7. Parts 2 and 3 represent lower-level parts AbsRes1 and AbsRes2 (AbsRes1 stands for "abstract resource 1"). Part 4 references RM1, used for controlling a concrete resource (not shown in the figure) that AbsRes1 represents, Part 5 references RS1, used for configuring the same concrete resource. Four workflows are defined and shown in the component labeled USC: the mandatory *create* and *delete* workflows used to create and destroy the service instance, and the custom *add* and *remove* workflows used to change the allocation of the AbsRes1 resource.

Workflows are defined by providing mappings from workflow names to operations on RMs and RSs. To avoid clutter only the mappings for RM1 and RS1 are shown in the figure. Note that the *create* workflow is mapped to a create operation on RM1 and a config1 operation on RS1. This implies that resource instances of type AbsRes1 are created and configured as a part of creating a service instance. Whereas the *add* workflow is mapped to operation create on RM1, it is mapped to a different operation config2 on RS2, which implies that a different configuration operation takes place when adding resources of type AbsRes1 to an existing service instance. Note that for RS1 the workflow *delete* is mapped to the special no-op. This implies that no configuration operation should take place when removing resources of type AbsRes1 from the service instance. The USC contains four knobs (conceptually) that control the service instance: create and delete operations to trigger the execution of the *create* and *delete* workflow and an executeWorkflow operation that may take two parameter values: add to trigger the execution of the *add* workflow, and remove to trigger the execution of the *remove* workflow.

Planner. The Planner transforms a service template into a concrete utility service definition that includes a set of workflows and other necessary information to be included in the USC. First, the topology tree is generated. The process starts by identifying the root node, a part referenced in the service template. The rest of the topology tree is generated by a recursive algorithm that starts with a set of nodes consisting of the single root node and expands it to the next level—its immediate descendants—by identifying the parts that are "aggregated" by it. The algorithm terminates when the set of parts to be expanded next are all dynamic parts and thus cannot be further expanded.

The second step includes the analysis of the topology tree and the generation of the *create*, *delete*, and custom workflows. For every workflow referenced in the topology tree, the operations on RMs/RSs that should be included in it are collected by using the mapping from workflow names to RMs/RSs operations defined in the dynamic parts that constitute the leaves of the tree. Note that the actual operations may have names different from the name of the workflow.

After the operations for a particular workflow are collected, they have to be ordered (the description of this is beyond the scope of the paper). A verification step ensures that the workflow is valid and contains no loops by analyzing the parameter values for each operation.

Schedule Coordinator. Before a service instance can be created, its resources have to be reserved. Reserving of resources is complicated by the fact that the availability of various resource types required may be limited to certain time slots. The reservation process involves the RMs and the Planner and is coordinated by the Schedule Coordinator.

Recall, that every RM implements the findAvailability and commitResource operations. The findAvailability operation returns all time slots within a given range for which the required resources are available. The commitResource operation commits resources for a particular time slot and returns a reservation ticket to be used later in the create operation when the resources are actually allocated.

In addition to the *create*, *delete*, and custom workflows, the Planner also generates *findAvailability* and *commitResource* workflows. The *findAvailability* workflow invokes in sequence the findAvailability operations on all the RMs that are referenced from the leaves of the topology tree, and it returns the time slots for which resources are available. The *commitResource* workflow invokes the commitResource operation on all RMs referenced from the topology tree to commit the required resources for a particular time slot. It returns reservation tickets for the committed resources.

A schedule operation has to be invoked on the USC before the actual service instance is created. This operation, which receives as parameters a start range and duration, invokes the Schedule Coordinator to reserve the necessary resources for the service instance.

The Schedule Coordinator triggers the execution of the *findAvailability* workflow. Next, it analyzes the return values of the workflow to find a start time within the given start range for which all resources are available for the required duration. Finally, it triggers the execution of the *commitResource* workflow to actually reserve the resources. The reservation tickets returned from the *commitResource* workflow are used later as input for the *create* workflow.

The Schedule Coordinator may fail to find a time slot for which all required resources are available. Even if it found an appropriate time slot based on the results of the *findAvailability* workflow, due to the distributed nature of the system, the *commitResource* workflow may fail if one or more of the commitResource operations on the RMs fail (possibly due to a competing asynchronous process of reservation for another utility service). In every one of these cases, a compensation action (not covered here) is taken by the Schedule Coordinator.

Plan Executor. Because plans are realized in the framework as workflows, the Plan Executor is a workflow engine. The workflow engine used must have a programmatic interface that allows dynamic deployment of workflows. Workflows for a particular service instance are deployed when the deploy operation is invoked on the USC. The execution of a deployed workflow by the workflow engine is triggered either by the Schedule Coordinator, which is the case for the *findAvailability* and *commitResource* workflows, or by the USC following a create, delete, or executeWorkflow invocation.

Workflow engines are particularly useful when dealing with long-running tasks or when some of the operations are performed manually, which is often the

case in utility systems. To learn more about workflow and workflow engines see References 17 and 18.

The Life Science Computing Service

Life sciences research companies developing new drugs need computing power to solve problems such as DNA sequencing or protein folding. These applications require intensive computation that uses hundreds of servers at a time. A typical computation lasts on the order of a few weeks. After the computation is complete, a laboratory phase that may last several months follows, during which the computing power is not needed. Thus, life-science companies face the cost and complexity of maintaining an IT infrastructure with a large computing capacity that is used only for a few weeks every few months.

The Life Science Computing Service (LSCS) can alleviate this problem and benefit the life-science customers. When a customer subscribes to the LSCS utility service, a base computation environment is created that includes dedicated servers, storage, a virtual private network (VPN) connection, and control software—all needed to perform computation. This environment is maintained for the duration of the customer subscription. At the start of a computation phase, the service provider upgrades the environment with a specified number of computation servers for the required duration. Upon completion, the service provider reclaims the servers for other uses. Because of the idle periods, the service provider can support several customers with the same infrastructure. The customers avoid the cost of acquiring and maintaining the computing infrastructure, and pay only for the computing resources they use.

In this section we show how we use the framework to create an LSCS utility system. The LSCS is a relatively simple utility, as it requires neither resource performance monitoring nor resource reallocation based on load and is thus suitable as a first implementation.

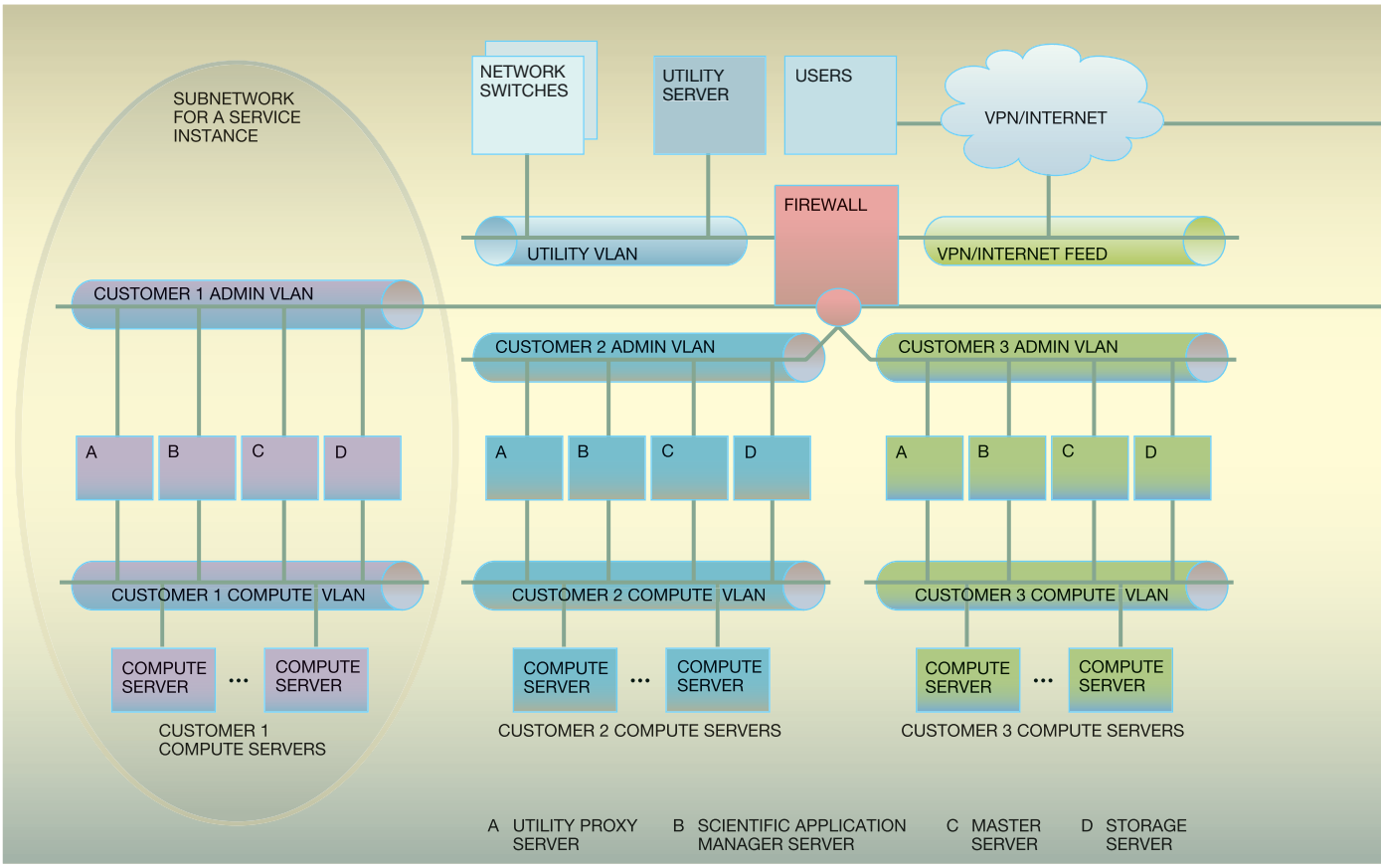
Overview of the LSCS utility. Every user is associated with a service instance that is created at the time the subscription takes effect. Using a GUI for this purpose, a user may request that additional computational servers be allocated to the environment for some length of time. Similarly, the user may request that computational servers be removed from the environment.

There are two types of resources used in LSCS: fixed resources and dynamic resources. The fixed resources are allocated to the service instance when it is created and deallocated when it is destroyed. They include four physical servers: a storage server, a utility Proxy, a master server, and an application server that is used to run the Scientific Application Manager (also known as *the manager*). The manager, which controls and manages the scientific computation on the computational servers available to it, is supplied by the customer; the LSCS does not mandate the use of a particular manager. It typically includes an internal scheduler that dispatches tasks to the computational servers and collects the results, and a data manager that caches data on the local disks of these servers as needed for the computation they perform. It is assumed that the manager automatically discovers newly allocated computational servers, and thus no interaction or reconfiguration is required. It is also assumed that the manager can deal with failures of computational servers (e.g., by redispatching the task), and therefore, computational servers can be deallocated from the service instance without notifying the manager. To accommodate more restrictive assumptions regarding the manager might require that the workflows that affect a service instance include some additional appropriate configuration operations. For example, an operation may be needed to notify the manager when a new computational server is added to a service instance.

In the future we envision a more sophisticated interaction between the manager and the utility system in which the manager may request more computational servers, based on the state of the computation. Moreover, when the utility system initiates an action to deallocate computational servers from the service instance, it might invoke a manager function to identify the best candidates for deallocation based on the state of the application. The design guiding principle is the separation of the two domains, the application domain from the utility system.

LSCS in more detail. In this section we describe the process of developing the sample LSCS utility using the framework. Our goal, aside from developing a running utility system, was to validate the idea of the framework as a system-building tool that involves reusable components, notably RSs, RMs, and parts. The design and implementation decisions that we made were shaped by this goal. For example, we used the Planner to generate the workflows included in the

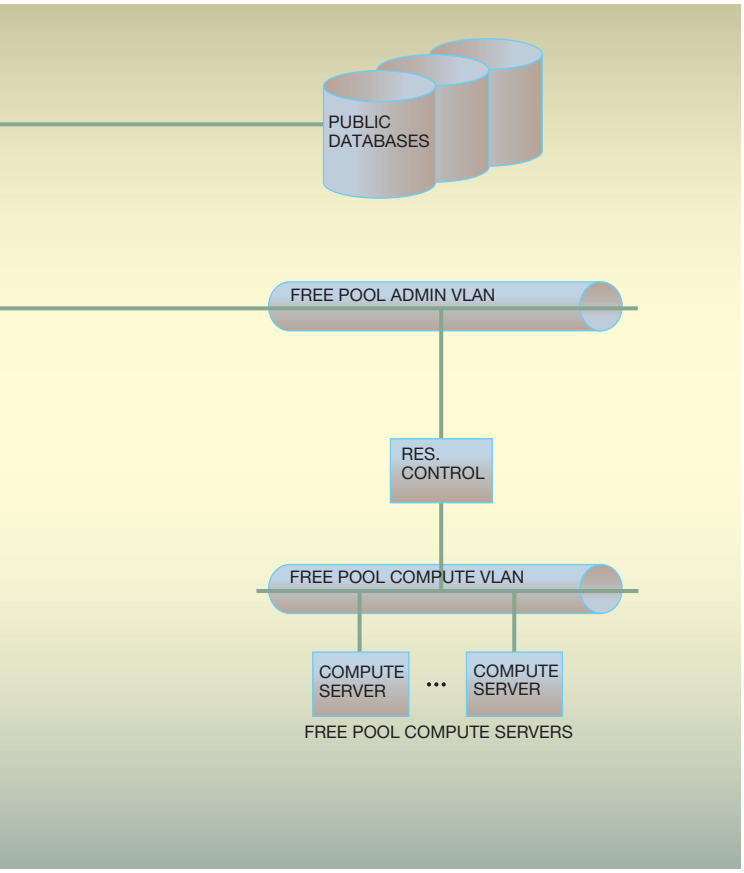
Figure 5 The infrastructure of an LSCS utility system



USC rather than writing them manually, this despite the fact that the workflows in this sample utility are rather simple. Specifically, we designed and implemented the code to control the resources as RSs and RMs, and we implemented the parts required to represent them. These parts were then used by the Planner to generate the workflow.

Due to time constraints, we took some shortcuts. For example, the function to create and destroy a service instance was done manually. Because the creation of a service instance involves configuration of firewalls and because an RS to automatically perform that operation was not available at the time, we decided to stay focused on our main goal—making the idea of a framework work—and invest our efforts in tasks that were directly related to this goal. Also, from an operational perspective, whereas a service instance is created once for each customer and typically remains unchanged for several years, resources such as the computational servers are allocated and deallocated at intervals of weeks. Thus, it is more cost effective to automate the process of allocating and deallocating computational servers for service instances rather than automating the process of service instance creation. As the framework mandates, we do generate *create* and *delete* workflows to provision the service instance after it is created.

The utility service controller. Concerning a service instance, two operations are mandatory in the framework: create and delete. When subscribing to the service, the user/subscriber can determine the initial computing capacity of the service instance. The user can also request to increase or reduce the computational capacity. For this, two operations are needed: addComputeServer and removeComputeServer. The actual operations that the USC provides are:



- create—triggers the execution of the *create* workflow
- delete—triggers the execution of the *delete* workflow
- executeWorkflow—a generic operation that receives as a parameter a name of a custom workflow and triggers its execution. Since *addComputeServer* and *removeComputeServer* are custom workflows, they can only be executed by using this operation.

It should be emphasized that the USC, including its operational interface above, is generated automatically by the framework at the time the new subscription is created.

The infrastructure and service instance environment. An important component of the utility system is the infrastructure on which it runs. Figure 5 illustrates the infrastructure of an LSCS utility system. The design of the infrastructure includes the design of the overall network (types of network elements, network topology), the resources to be allocated to service instances, as well as the service instance structure (network topology and fixed and dynamic resources).

The design of the infrastructure is based on requirements from the utility system provider, the service offering provider (if different from the utility system provider), and the subscribers/users. The design has to consider the utility provider's existing network and possible constraints on its use. For life-science applications there is usually a strict security requirement: life-science companies require a strong guarantee that their data are kept secure. In some cases, even information concerning the allocation/deallocation of computing resources is considered sensitive and needs to be protected.

The design of the network and the service instance structure, illustrated in Figure 5 and further described next, was based on these requirements. Specifically, computational servers allocated to different service instances are isolated, and their local disks are scrubbed between allocations. The users communicate with the utility system over VPNs. Thus, even information on requests to allocate/deallocate servers is protected.

As shown in Figure 5, the network includes switches, VLANs, a firewall, and VPN connections. Servers, including computational servers, are allocated static fixed IP addresses. Every service instance has a subnetwork that includes two VLANs: an administrative VLAN, and a computational VLAN (one such subnetwork is highlighted in Figure 5). A set of designated servers (storage server, Scientific Application Manager server, utility proxy server, and master server) are connected to both VLANs. The number of these servers is fixed for the duration of the life of the service. The computational servers, which are dynamically allocated, are connected to the computational VLAN. The utility server (shown at the top of the figure) runs the framework code, which includes the RMs, RSs, the Planner, and so forth. The utility server, together with the network switches, is connected to the utility VLAN. This allows the RMs/RSs running on the utility server to configure the switches when computational servers are allocated/deallocated. The utility VLAN is connected to the various admin VLANs through the firewall. The free-pool subnetwork resembles a service instance subnetwork.

Users (shown attached to the VPN/Internet cloud in the figure) access the servers connected to their administrative VLAN on a dedicated VPN through the firewall. Computational servers cannot be accessed directly by the utility server or by the users. All servers allocated to a service instance are isolated from servers in other service instances.

Each service instance includes fixed and dynamic resources. In LSCS, the only dynamic resources are the computational servers. The following fixed resources are allocated to a service instance when it is created and deallocated when it is destroyed.

- *Storage server*—A shared file system is used in LSCS.
- *Scientific Application Manager server*—This is the server on which the manager application is installed.
- *Utility proxy server*—Because the utility system cannot access the computational servers directly, it needs "agents" to perform its functions inside the service instance. These agents are installed and started on the utility proxy. Some of the corresponding processes are part of the distributed implementation of various RMs/RSs. Others components include a Dynamic Host Configuration Protocol (DHCP) server and an "image manager" used to create new computational server images that can be installed on computational servers when they are allocated to the service instance.
- *Master server*—The master server is used by the user/subscriber to create a computational server image to be captured by the image manager. The user accesses the master server directly and configures it as desired. The user then captures the computational server image thus created with the image manager. When a new server is allocated, the captured image is installed on it.

RMs, RSs, and RICs. We describe here the main controllers in the LSCS system.

The *Compute Server RM* (computeServerRM) manages the pool of available computational servers (free pool). It provides the following operations:

- createComputeServer—accepts a description of a computational server and returns the handle of a computational server RIC that represents a particular computational server in the free pool. The service can be used to retrieve information needed to access the computational server in the free pool, such as its IP address.
- deleteComputeServer—the counterpart of createComputeServer, accepts the handle to a computational server RIC that identifies a computational server to be returned to the free pool.

The *Install RS* (InstallRS) performs much of the work needed to transfer a computational server from the free pool to a service instance (and back). It includes two main operations:

- preInstall removes the MAC address of the computational server from the DHCP server at the source (service instance or free pool) so that it will not respond to network boot requests from the server. In addition, it adds the MAC address of the server to the DHCP server at the destination. It then configures the server to boot from the network and reboots it.

- install completes the installation of the requested image on the server after it is moved to the destination environment. The disk scrubbing preceding an allocation is implemented by installing an in-memory Linux** image on computational servers whose destination environment is the free pool. This image is preloaded with a start-up script that writes zeroes to the disk. The operation returns only after the computational server is ready for use.

The *SwitchConfig RS* (SwitchConfigRS) configures the switch as part of the allocation/de-allocation process of computational servers. It provides an assign operation that assigns a computational server to the computational VLAN of the destination environment (service instance or free pool).

The order of operations to allocate a computational server to a service instance is important. First, the computational server is allocated by using the createComputeServer operation. Then, the preInstall operation is invoked to prepare the computational server (and the DHCP servers) for installation. Then, the computational server is located at the destination computational VLAN by configuring the switch by means of the assign operation. Last, the installation process is triggered by the install operation.

During the design of the LSCS utility system we realized that an additional entity, not previously accounted for, was needed. Our RMs/RSs were only capable of handling a single computational server at a time. The workflow that was generated from the

parts representing them also dealt with a single computational server. We needed an entity to manage a set of computational servers allocated to the same service instance. In particular, an iteration over a set of computational servers was needed that involved the execution of the *addComputeServer* and *removeComputeServer* workflows for every one of them. The same entity would also select a subset of the set of computational servers for deallocation based on a user request.

Our design treats this entity as an abstract computeServerGroup resource. A computeServerGroup resource is allocated (initially empty) to every service instance by using a ComputeServerGroupRM. It is realized as a computational server group RIC, a service instantiated by the RM when a computeServerGroup resource is conceptually allocated.

While going through this exercise, we realized that a resource group is a useful concept. Future versions of the framework will include it as a generic object that can be used to build higher-level parts. For LSCS, we had to handcraft the computational server group RIC and its RM as follows.

ComputeServerGroupRM is the RM of the computeServerGroup abstract resource. It provides the commitResource, create, and delete operations. The create operation merely instantiated a new computational server group RIC, the delete operation destroys it.

ComputeServerGroupRIC represents a group of computational servers. It provides the following operations.

- selectAddComputeServers accepts as parameter the number num of computational servers to be allocated (and an identifier of the image to be installed on them). It executes the *addComputeServer* workflow to add a single computational server num times. This is done by invoking executeWorkflow on the USC.

- selectRemoveComputeServers selects computational servers for deallocation. The only parameter currently supported is the identifier of the image installed on the computational server (different computational servers may have different images installed on them). The *removeComputeServer* workflow is iteratively executed over the set of selected computational servers.

Parts and the topology tree. The Planner uses parts to generate the required workflows to be encapsulated in the USC; the dynamic parts provide the mapping from workflow names to RMs/RSs operations. In our case, there are four required operations: create, delete, addComputeServer, and removeComputeServer. Figure 6 shows the topology tree for LSCS, including the parts and the mappings that the dynamic parts define from workflows names to operations.

The topology tree consists of one static part and five dynamic parts, the latter representing two RMs and two RSs. Two parts are needed to represent two different operations, Install and PreInstall on Install RS (the current mapping mechanism supports only one operation per part/workflow pair). If there is no entry for a particular workflow in a mapping, then no operation on the respective RM/RS has to be included. Note that the only operation that is included in the *create* (*delete*) workflow is a create (delete) operation on the ComputeServerGroupRM. Indeed, the only automatic operation in provisioning an LSCS service instance is the instantiation of a computational server group RIC. Other operations are performed manually in this version of LSCS. The static LSCS part aggregates the five dynamic parts (the descendants of the root node in the figure) needed for the definition of the service.

One of the advantages of the framework is that the complete topology tree, or just a subtree, can be reused in other, more complex cases. Thus, using the framework, the work performed when building RMs and parts does not have to be repeated when building new utility systems.

The Planner automatically generates four workflows from the parts and RMs/RSs described earlier. See Table 1.

Customer interactions. Because UBS was not yet available when LSCS was implemented, we used instead a simple component to manage all interactions with the customer as subscriber. This component, the *customer representative*, handles subscription requests and requests to add or remove computational servers to or from existing environments by interacting with both the USC and the computational server group RIC (which are not accessible by the subscriber).

Subscription requests are handled by first instantiating a new USC to represent a new service instance (by calling a controller factory), then building and

Figure 6 The topology tree for LSCS

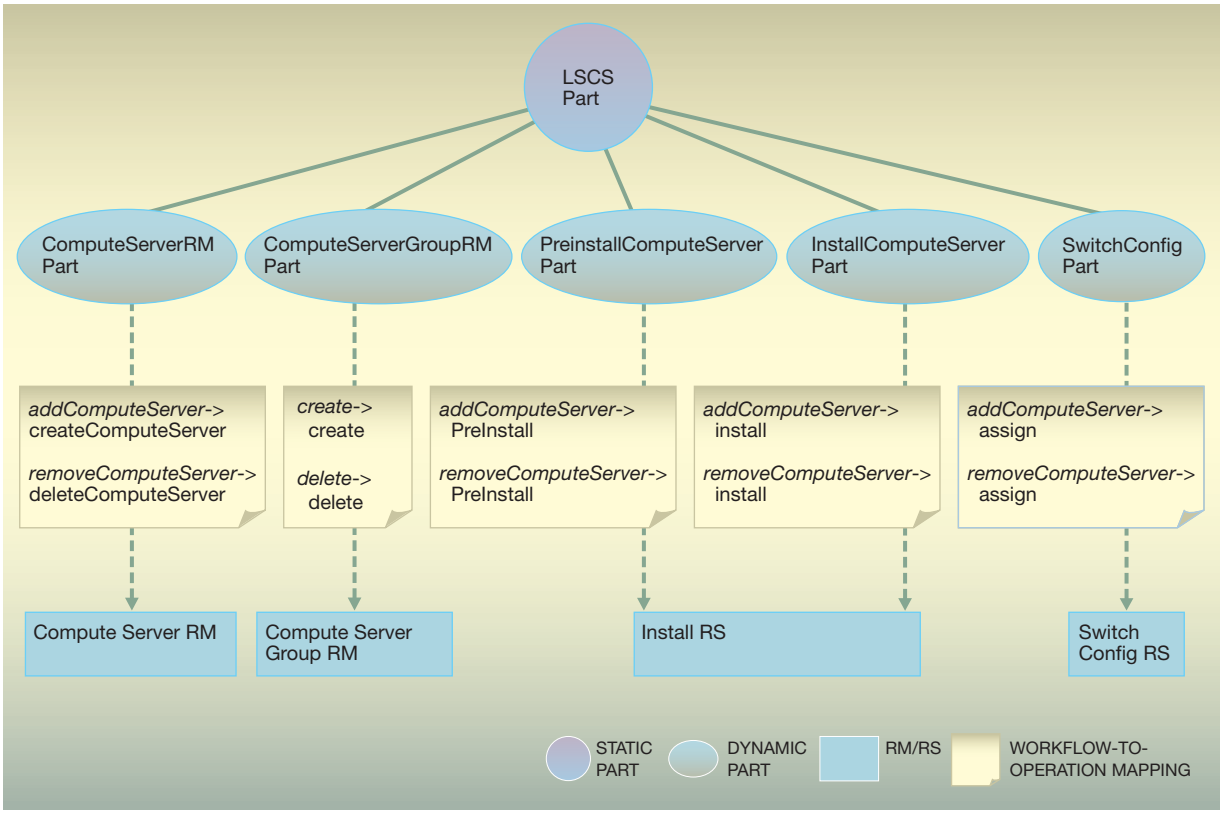


Table 1 Workflows generated from parts by the Planner

***create* Workflow**	***delete* Workflow**
ComputeServerGroupRM.create	ComputeServerGroupRM.delete
***addComputeServer* Workflow**	***removeComputeServer* Workflow**
computeServerRM.createComputeServer	InstallRS.PreInstall
InstallRS.PreInstall	SwitchConfigRS.assign
SwitchConfigRS.assign	InstallRS.Install
InstallRS.Install	ComputeServerRM.deleteComputeServer

deploying the workflows by calling build and deploy operations on the USC. Reserving resources by calling schedule is next, and finally, invoking create to trigger the execution of the *create* workflow. Note that the actual service instance was built in a manual process (as the *create* workflow merely instantiates an empty group instance service). In LSCS, the schedule operation merely reserves an abstract ComputeServerGroup resource and always succeeds.

Requests to add or remove compute servers are handled by interacting with both the USC and the ComputeServerGroupInstanceService. Basically, the operation selectAddComputeServer (or selectDelete ComputeServer for delete) on the ComputeServer-GroupRIC handles the request. The USC is called only to retrieve a handle to the ComputeServerGroupRIC. Note that ComputeServerGroupRIC was created by the *create* workflow; a handle (URL) to it was then

returned to the USC, which refers to it as a resource. Resource handles can be retrieved from the controller using a getResources operation. The interaction is further described in the next section.

Usage scenario. Figures 7 and 8 illustrate a usage scenario that covers all phases of the utility service life cycle.

a. A customer (a life-science company) subscribes to the service provided by LSCS. The process uses an interface provided by the customer representative component and may include human interaction.
b. The service instance corresponding to the new subscription is manually constructed. It includes an application manager server that runs the subscriber-selected manager application.
c. The customer representative component instantiates a new USC by invoking the controller factory (Step 0 and Step 1 in Figure 7).
d. The customer representative calls build on USC; the operation invokes the planner to generate the workflows (Step 2 in Figure 7).
e. The customer representative calls deploy on USC to deploy workflows on the workflow engine (Step 3 in Figure 7).
f. The customer representative calls schedule on USC to reserve an abstract ComputeServerGroup resource (Step 4 in Figure 7).
g. As soon as the workflows are generated and deployed, their execution can be triggered by invoking operations on the controller. The *create* and *delete* workflows can be executed by invoking create and delete respectively. The *addComputeServer* and *removeComputeServer* workflows can be executed by invoking the executeWorkflow operation.
h. The customer representative invokes the create operation on USC (Step 5 in Figure 7). This triggers the execution of the *create* workflow.
i. The *create* workflow invokes create on the computeServerGroup RM. A ComputeServerGroupRIC is instantiated and a handle to it is returned to the USC (Step 6 and Step 7 in Figure 7).

The scenario continues in Figure 8.

a. The subscriber requests the customer representative to add a number num of computational servers to its service instance.
b. The customer representative invokes getResources on USC to get the handle to ComputeServerGroupRIC allocated to it (Step 0 in Figure 8).
c. The customer representative uses the handle to invoke selectAddComputeServers on the ComputeServerGroupRIC with a parameter value num (Step 1 in Figure 8).
d. ComputeServerGroupRIC invokes executeWorkflow (*addComputeServer*) on the controller (Step 2 in Figure 8). Each such invocation triggers the execution of the *addComputeServer* workflow, which adds one computational server to USC from the free pool (Step 3, Step 4, and Step 5 in Figure 8). Note that only the beginning of the workflow is described in the figure; the complete workflow is found in Figure 1. The handle to the computeServerRIC representing the computational server is returned to USC from the workflow and kept as an attribute.
e. ComputeServerGroupRIC retrieves the handle by invoking the getAttribute operation on the USC in order to keep track of the computational servers in the group. It adds the handle to its internal data structures (Step 6 in Figure 8).

This process (items d and e in this description) is repeated num times.

Figure 7 Usage scenario (part 1)

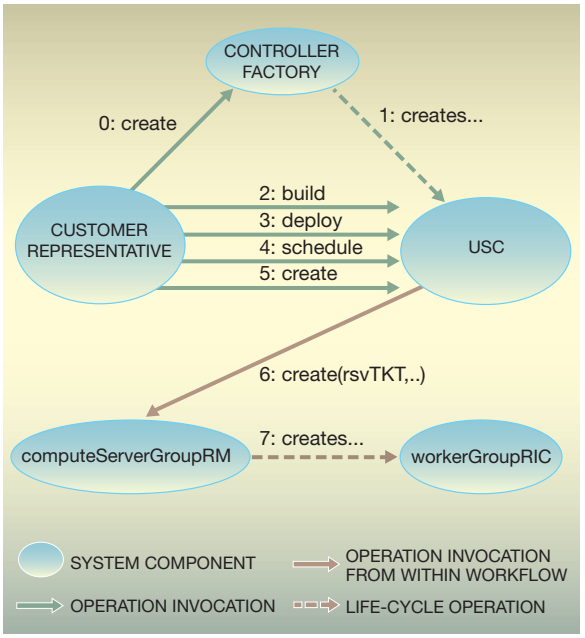
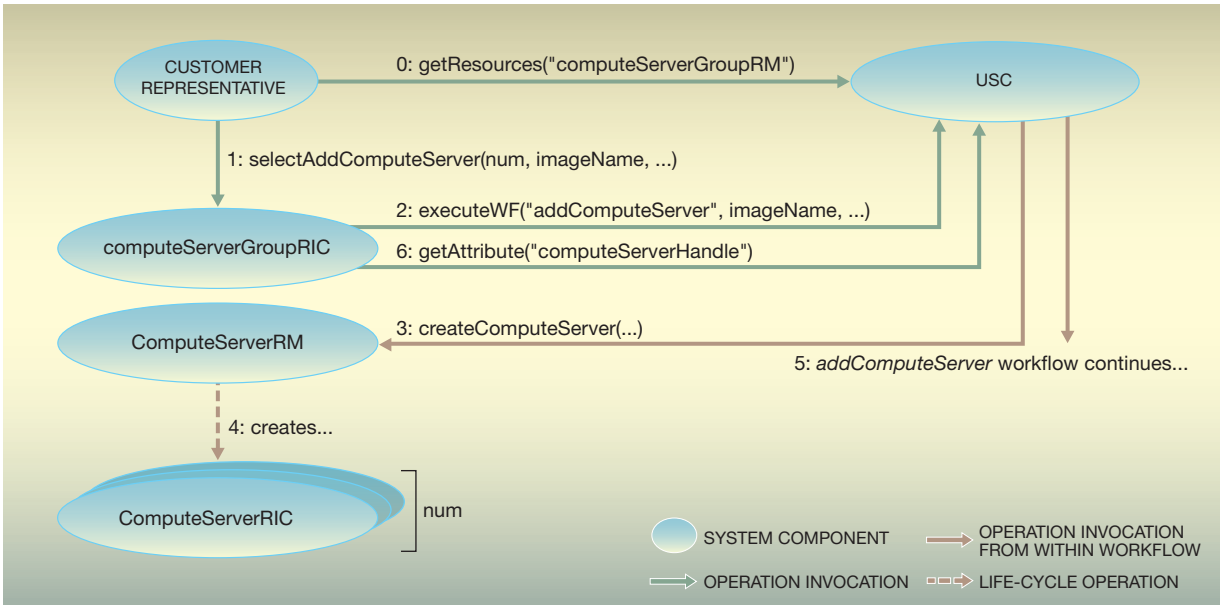


Figure 8 Usage scenario (part 2)



The on-line gaming service

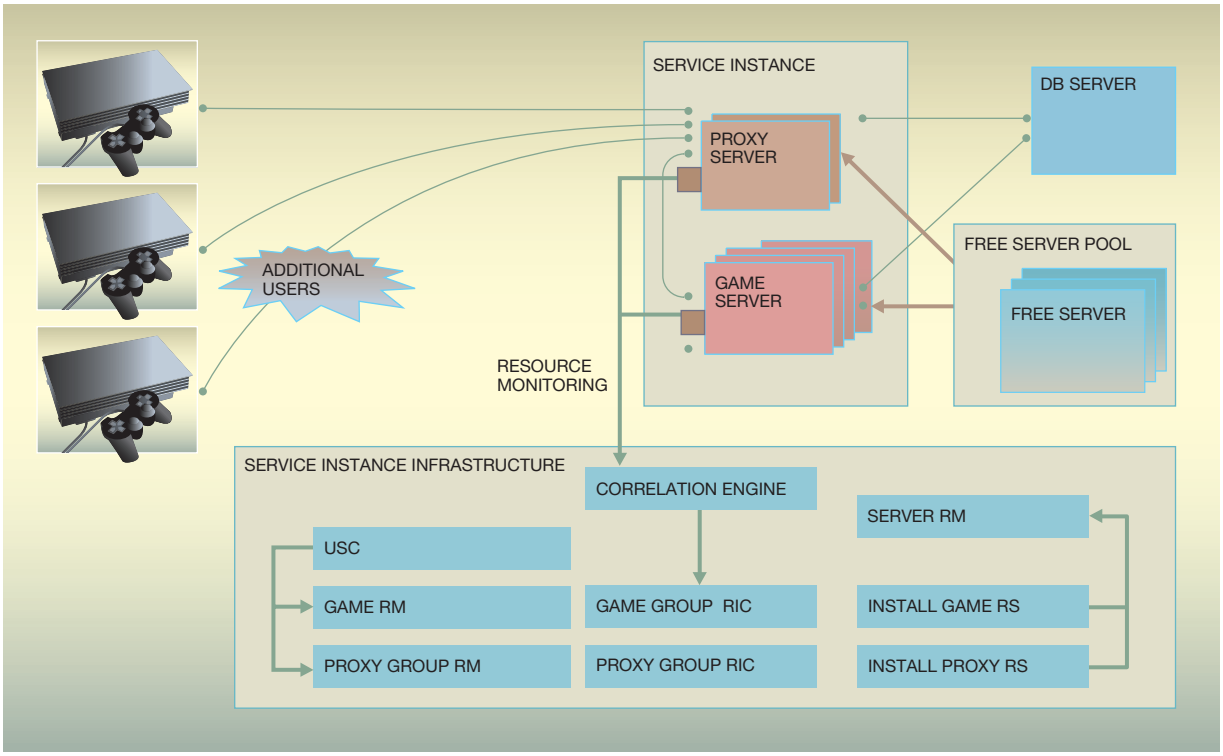
On-line games fall into two major categories: massively multiplayer persistent (MMP) games and session-based games. MMP games, which are frequently adventure-based, can host thousands of players, and a session can run indefinitely. Session-based games, which are typically shooter games, have typically less than 128 players and tend to be short lived, lasting less than one hour. These two game types are also differentiated by the required response time for state updates and by the number of locations (players) to which these updates must be distributed. These differences place unique requirements on the underlying resources.

Today's game developers and game publishers have separate infrastructures with dedicated resources for each stage (development, test, and production) in the life cycle of a game. These dedicated resources, such as management servers, file stores, database servers, firewalls, and DNS (domain name service) servers, are generally underutilized and are not dynamically reallocated. Moreover, the development, test, and production environments place different demands on infrastructure resources.

The on-line gaming service (gaming service, for short) offers environments to game developers and publishers that are customizable in terms of resources, allows sharing of resources, and can vary the resource allocation based on load. Thus, developers and publishers can concentrate on their core business, the creation and delivery of entertainment content, and leave the management and maintenance of the gaming infrastructure to the utility service provider. Customers are charged only for the resources actually used. For example, a developer may use a game service instance for a few months, while a publisher may require the transfer of resources from one game service to another as the popularity of games changes in the marketplace.

MMP games and session-based games both employ techniques to optimize resource usage while meeting their performance goals for game play events. The gaming service provides monitoring of performance data and a set of policies using that data to drive dynamic allocation of resources. Some resource types are uniformly managed across service instances and employ and thus are controlled by the same policies and parameter values. Other resource types require treatment that is specialized to the game and has custom rule sets. For these custom rules, the utility service makes no guarantees as to their correctness and applies them as long as there is no contention over resources. In case of contention, the

Figure 9 The on-line gaming system



applicable SLAs between the subscriber, the service provider, and the service supplier determine the actual allocation of resources.

The gaming service differs from LSCS in its requirements: the mix of resource types, the duration of resource allocations, the support for custom (game) software, and the support for dynamic resource allocation in response to changes in load. The security requirements, however, are not as stringent as those for LSCS.

Overview of the gaming service. The gaming service provides game environments for development or play. Customers subscribe to the service weeks or even months in advance. A subscription is usually for a specified game type, set of resources, activation time, and duration. The service provider can make use of rule sets that determine how resources are managed. Game service instances are created, managed, and destroyed on a continuous basis. The provisioned environment may contain development tools, various security features, and facilities for managing players. Game players can subscribe to individual games, log in and log out of games, and engage in peripheral activities such as chat. The active game environments acquire and release resources automatically, depending on player activity.

Like LSCS, game environments contain fixed resources and dynamic resources. The fixed resources are allocated when the service instance is created. These include a firewall, a DNS server, a network address translation (NAT) router, a file store (where the game images are stored), and a database server (on which game databases store the persistent game state). Although network isolation is not required, security is provided by the firewall, a packet filter on each server that discards traffic not directed at the port assigned to the game it hosts, and a NAT router that only routes traffic to gateway servers.

The dynamic resources include a set of game servers for game logic and state management, and a set of proxy servers where players log in and game sessions are maintained. Add-on services, such as chat,

are also hosted on the proxy servers. Figure 9, which shows the gaming service architecture, depicts the servers as part of the service instance infrastructure.

Design of the gaming service. We describe here the gaming utility service controller (USC), which represents and controls a service instance, and the base components, RMs, RSs, and RICs, that are used to build and support the gaming service. We also describe the parts, the topology tree, and the workflows that make this service work.

The USC. The USC operations include (1) create—triggers the execution of the *create* workflow, (2) delete—triggers the execution of the *delete* workflow, and (3) executeWorkflow—a generic operation that receives as a parameter a name of a custom workflow and triggers its execution. executeWorkflow is used to invoke the workflows that are generated for each one of the following four operations:

1. addGameServer—Add a game server to the service instance
2. addProxyServer—Add a gateway server to the service instance
3. delGameServer—Delete a game server from the service instance
4. delProxyServer—Delete a gateway server from the service instance

As in LSCS, we call the two *addXXXServer* workflows (*XXX* stands for either "Game" or "Proxy") through the use of the selectAddXXXServer call, which is carried out by XXXGroupInstanceServices immediately after the creation of the service instance to allocate the gaming environment's initial server sets. The custom workflows are invoked by using the executeWorkflow operation on the USC.

RMs, RSs, and RICs. *Server RM* is similar to the computational server group RM for LSCS.

Install Game RS performs the work needed to reallocate a server from the free pool and configure and install it in the gaming environment as a game server. It has two operations:

- Install—installs and configures the base operating system, the database connection libraries, and security software.
- ConfigureGS—configures the server for the role it will play in the service instance (namely, a game server). This involves the running of various scripts and installing the data collection agent (monitoring).

Game Group RM is similar to the LSCS ComputeServerGroup RM. When its create operation is invoked, it creates the GameGroupInstanceService, which provides the selectAddGameSvr and selectDelGameSvr operations. The selectAddGameSvr operation adds a set of num game servers to the game server group by executing the *addGameSvr* workflow num times. Similarly, the selectDelGameSvr operation removes game servers from the group by invoking *delGameSvr* workflow. Both of these workflows are executed by invoking the executeWorkflow operation on the USC.

Install Proxy RS and *Proxy Group RM* are the counterparts of the Install Game RS and the Game Group RM for the proxy servers.

Infrastructure RM performs the system configuration operations that are not server based. For each new service instance the RM (1) opens a game port on the firewall, (2) configures the DNS server, and (3) configures the NAT router.

Database RM, which is used to store the persistent state, creates the database for the game service instance. This RM has two operations: (1) create—creates the database instance by using a set of scripts that load database tables (the instance is created on a server node that is already configured as a database server), and (2) delete—deletes the database instance.

Parts and the topology tree. Figure 10 shows the parts and the topology tree for the gaming service, as well as an illustration of the mapping from workflows to RM/RS operations. In this figure, the subtrees rooted at the Game Server Part and the Proxy Server Part have the same structure; both employ group parts and install-server parts, and both use the Server RM as a common component (for clarity, only the Game Server Part subtree is shown in detail).

Using the game service template, the Planner builds the workflows and the rules that will be invoked at runtime to create and manage the gaming service instance. The workflows include:

create—Creates gaming service instance
delete—Deletes gaming service instance
addGameSvr—Reserves, creates, and adds *N* servers to the game server group

Figure 10 The topology tree for the gaming service

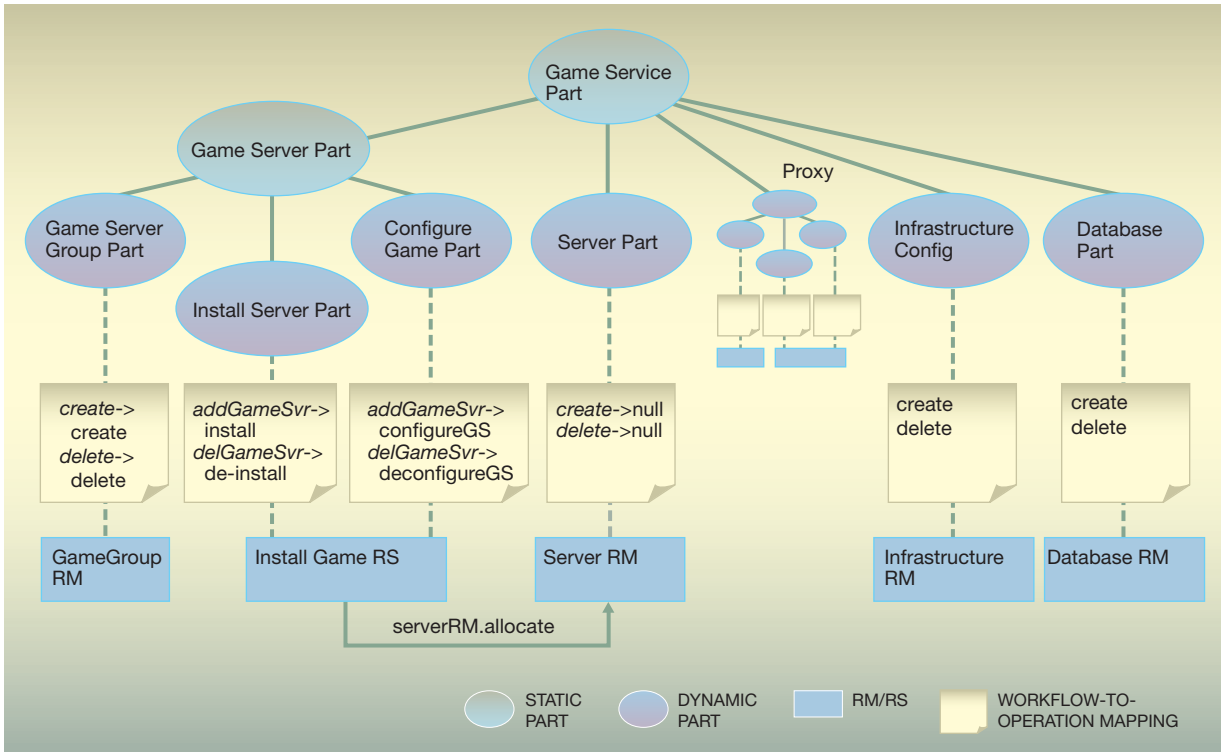


Table 2 Examples of workflows generated by the Planner and their corresponding operations

***create* Workflow**	***addGameSvr* Workflow**
Game-Grp.create	InstallGameRS.Install
Proxy-Grp.create	InstallGameRS.ConfigureGS
databaseRM.create	
infrastructureRM.create	

delGameSvr—Deletes a server from the game server group
addProxySvr—Reserves, creates, and adds *N* servers to the proxy server group

delProxySvr—Deletes a server from the proxy server group

A "meta-workflow" that invokes the *create* workflow and adds the initial server sets by invoking the *add* workflow for each of the server types is defined manually outside of the Planner. Examples of workflows generated by the Planner and their corresponding operations are shown in Table 2.

Event correlation. The gaming service uses an event-correlation engine to drive reallocations of resources for gaming service instances, and also for problem determination. In the current design only a few policies are supported, such as policies to aggregate and smooth performance data, threshold rules that monitor the aggregate metrics and detect performance violations, and availability monitors that periodically check server availability. When a server is determined to be overloaded or under loaded, the threshold policies trigger automatic resource rebalancing using the selectAddXXXSvr and selectDelXXXSvr operations. These policies are described in detail in Reference 19.

A data collection agent used for monitoring resides on each server resource. Agents are configured by the InstallXXXRS.ConfigureXX operation to provide sensor data for the appropriate type of resource. Sensor data is pushed to, or pulled from, the utility management node. The correlation engine gets notified when these events occur (the setup is performed by the InstallXXXRS.ConfigureXX operation). The correlation engine is kept apprised of which servers belong to which service instance by GroupInstanceServices. This basic configuration data is used in conjunction with the sensor data by the rule-based correlation engine.

Conclusions

In this paper we described a utility computing framework and two utility systems that were designed using it. The LSCS utility system has been implemented and is operational. Our experience shows that the utility-computing framework facilitates the creation of new utility systems by providing:

- Common functions, such as resource reservation, generation and execution of workflows, and monitoring and correlation of events.
- Common interfaces for those functions that are resource-specific so that they can be easily developed and incorporated in the framework.
- A methodology and tools for defining the infrastructure needed to support a utility service and flexibly assembling existing provisioning and management functions based on the service requirements.

Our experience in using the framework for the sample utilities showed that components such as RMs and parts are often reused. For example, the LSCS server and server group components were reused to a large extent in the gaming utility system. We also discovered that patterns that were not initially part of the framework, such as the server group, appear to have broad applicability. This componentization of provisioning and management function is important not only for reuse, but also because it facilitates development by multiple vendors while preserving interoperability.

We found that the limitations of the current mechanism for mapping workflows to operations resulted in a larger than expected set of parts. Some of these parts represent configuration tasks that were mapped to resource services in an ad hoc fashion. In addition, these tasks were highly specific to a particular physical infrastructure, and it is unclear how reusable the parts and the corresponding implementation components will be. Overcoming these challenges, while preserving the benefits offered by the framework, is part of our ongoing work.

The general problem of building solutions and commercially deploying utility computing services is being tackled in the marketplace. Our future work is intended to help make utility computing a commercial success through the use of advanced technology, and more specifically by applying autonomic methods to the utility computing environment.

Acknowledgments

We had many interactions with researchers, architects, and developers while working on this project, and we thank everyone who contributed. We recognize Georg Ochs, Georg Bildhauer, and Andrea Schmidt for their contributions to the project as a whole and for their insight that affected this work. We thank Stefan Zink, Sebastian Bauer, Peter Taube, and Srirama Krishnakumar for their work on various aspects of the LSCS. We thank Germán Goldszmidt for his contributions to Océano, an early prototype from which this project benefited significantly. We also thank Yariv Aridor, Ofer Biran, and Srirama Krishnakumar for their contributions to Raquarium, another important early work in this space. We also thank Yariv Aridor and Ofer Biran for helpful discussions that shaped our ideas on the architecture of the framework. Kemal Ebcioglu contributed to our understanding of the requirements through his analysis of related work and its shortcomings. Various members of our team benefited from the time spent discussing gaming services with Boaz Betzler. Finally, we thank Andrew Laycock, Terence Wells, and Gordon Watson for introducing us to life-science computing and helping us vet our approach to this problem.

Cited references

1. K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. P. Pazel, J. Pershing, B. Rochwerger, "Océano—SLA Based Management of a Computing Utility," *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*, IEEE, New York (2001).

2. J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle, "Managing Energy and Server Resources in Hosting Centers," *Proceedings of the 18th Symposium on Operating Systems Principles*, ACM, New York (2001).
3. A. Abbondanzio, Y. Aridor, O. Biran, L. L. Fong, G. S. Goldszmidt, R. E. Harper, S. M. Krishnakumar, G. Pruett, and B. Yassur, "Management of Application Complexes in Multitier Clustered Systems," *IBM Systems Journal* **42**, No. 1, 189–195 (2003).
4. J. Moore, D. Irwin, L. Grit, S. Sprenkle, and J. Chase, *Managing Mixed-Use Clusters with Cluster-On-Demand*, Technical Report, Department of Computer Science, Duke University (November 2002).
5. *HP Utility Data Center*, Technical White Paper, Hewlett-Packard Company (October 2001).
6. *ThinkDynamics: On Demand Provisioning*, IBM Corporation (2003).
7. M. Lovington, *Intelligent Storage Platform*, Sun Microsystems, Inc. (2002).
8. *N1 Provisioning Server 3.0 Blades Edition*, Sun Microsystems, Inc. (2003).
9. *The I-Fabric Architecture*, Sun Microsystems, Inc. (2002).
10. *Op-Force 2.0—IT Automation Suite*, Jareva Technologies, Inc. (2002).
11. P. Goldsack and P. Toft, "SmartFrog: a Framework for Configuration," *Large Scale System Configuration Workshop*, National e-Science Centre, UK (2001).
12. R. Braynard, D. Kostic, A. Rodriguez, J. Chase, and A. Vahdat, "Opus: an Overlay Peer Utility Service," *Proceedings of the 5th International Conference on Open Architectures and Network Programming (OPENARCH), IEEE*, New York (2002).
13. The Globus Alliance, http://www.globus.org.
14. I. Foster, C. Kesselman, J. Nick, S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," *Global Grid Forum*, Open Grid Service Infrastructure WG (June 22, 2002).
15. A. Roy and V. Sander, "GARA: A Uniform Quality of Service Architecture," *Grid Resource Management: State of the Art and Future Trends*, J. Nabrzyski, J. M. Schopf, and J. Weglarz, Editors, Kluwer Academic Publishers, Norwell, MA (Fall 2003).
16. Open Grid Services Infrastructure Working Group (OGSI-WG), Global Grid Forum, http://www.ggf.org/ogsi-wg.
17. D. Georgakopoulos, M. F. Hornick, A. P. Sheth, "An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure," *Distributed and Parallel Databases* **3**, No. 2, 119–153 (1995).
18. F. Leyman and D. Roller, *Production Workflow: Concepts and Techniques*, Prentice Hall, Upper Saddle River, NJ (2000).
19. K. Appleby and S. Calo, "Policy-Based Automated Provisioning," *IBM Systems Journal* **43**, No. 1, 97–120 (2004, this issue).

Accepted for publication October 11, 2003.

Tamar Eilam *IBM Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 (eilamt@us.ibm.com).* Dr. Eilam is a research staff member in the Distributed Middleware Department at the Watson Research Center. She received her Ph.D. degree in computer science in 2000 from the Technion, the Israel Institute of Technology, where she worked on trade-offs between space and efficiency in routing protocols. Dr. Eilam's areas of interests include systems management, distributed systems, and networking. She is currently working on the dynamic generation of automation plans for systems configuration.

Karen Appleby *IBM Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 (applebyk@us.ibm.com).* Karen Appleby is a member of the Network and Systems Management Department at the Watson Research Center. She holds an M.S. degree in computer science from New York University. Karen Appleby's main research interests are in the areas of event correlation, problem determination, e-commerce, and policy-based management.

Jochen Breh *IBM Development GmbH Germany, Schoenaicher Strasse 220, 71032 Boeblingen, Germany (breh@de.ibm.com).* Mr. Breh is a software engineer in the e-Utility Department at the IBM Development Laboratory in Boeblingen. His current activity is designing an on demand solution in the SAP space. Previously, he was involved in porting message-based infrastructures to the IBM eServer zSeries platform. Mr. Breh obtained a Master's degree in electrical engineering from the University of Stuttgart in 1997.

Gerd Breiter *IBM Development GmbH Germany, Schoenaicher Strasse 220, 71032 Boeblingen, Germany (gbreiter@de.ibm.com).* Gerd Breiter is a Senior Technical Staff Member working with IBM Fellow Jeffrey Nick's team on the architecture for the infrastructure for on demand computing. Prior to this project he worked on integrating rich media into WebSphere/J2EE business applications and on the architecture and design of transaction and recognition systems in the banking industry. His technical interests are in advanced Web technologies, especially in Web services and the grid environment.

Harald Daur *IBM Development GmbH Germany, Schoenaicher Strasse 220, 71032 Boeblingen, Germany (harald_daur@de.ibm.com).* Mr. Daur is a software engineer in the e-Utility Department at the IBM Development Laboratory in Boeblingen. He is currently working on the architecture for the infrastructure for on demand computing, focusing on the provisioning of IT resources and its underlying model. Previously, he was involved in the architecture and design of J2EE extensions to integrate audio and video data into business applications. Mr. Daur obtained a B.S. degree in computer science from the University of Esslingen in 1986.

Sameh A. Fakhouri *IBM Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 (sameh@us.ibm.com).* Sameh Farkhouri is an advisory software engineer in the Distributed Middleware Department at the Watson Research Center. He received a Master's degree in 1983 from New York University. His research interests include the discovery, monitoring, and management of network topology, as well as distributed messaging systems.

Guerney D. H. Hunt *IBM Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 (gdhh@us.ibm.com).* Dr. Hunt is currently Senior Manager of the Distributed Middleware Department at the Watson Research Center. He is also on the editorial board of the IEEE Pervasive Computing magazine. He received his Ph.D. degree in 1995 from Cornell University, working with Ken Birman in the area of multicast flow control for fault-tolerant distributed computing. His research interests include dis-

tributed systems, autonomic provisioning, on demand infrastructures, fault tolerance, pervasive computing, context, privacy, and security.

Tan Lu *IBM Systems Group, 2455 South Road, Poughkeepsie, New York 12603 (tanlu@us.ibm.com).* Mr. Lu is a core architect working on the design of IBM's on demand computing initiative. He received his M.S. degree in computer engineering in 1999 from Carnegie Mellon University. Previously, he was responsible for leading the architecture design for IBM's eServer zSeries z/900 I/O subsystem. His interests include the application of on demand technologies in emerging businesses in American and Chinese markets.

Sandra Miller *IBM Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532.* Sandra Miller is an Advisory Software Engineer in the Distributed Middleware Department at the Watson Research Center. She received her M.S. degree from Duke University in 1990. She has worked in the areas of network communications and distributed computing. She is currently on a leave of absence from IBM, residing in Winchester, England with her family.

Lily Mummert *IBM Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, New York, 10532 (lily@us.ibm.com).* Dr. Mummert is a research staff member at the Watson Research Center. She received her Ph.D. degree in 1996 from Carnegie Mellon University in the area of distributed file systems. Her research interests include distributed systems, systems management, fault tolerance, and performance evaluation.

John Pershing *IBM Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 (pershng@us.ibm.com).* Mr. Pershing is a research staff member in the Internet Infrastructure and Computing Utilities Department at the Watson Research Center. He received his B.S. and M.S. degrees in computer science from Massachusetts Institute of Technology. He has been involved in computer systems work for over 30 years, primarily in the areas of computer networking, large-scale clustering, and systems management.

Hendrik Wagner *IBM Development GmbH Germany, Schoenaicher Strasse 220, 71032 Boeblingen, Germany (xxhewa@de.ibm.com).* Mr. Wagner is currently with the IBM Development Laboratory in Boeblingen, Germany. After his graduation in computer science from the Berufsakademie in Stuttgart, Germany in 1991, Mr. Wagner worked on various IT projects for the automobile industry and telecommunication customers and on middleware-related product development on different platforms. His main interests are in object-oriented architectures and component-based systems.