Part III

Cloud technologies

A few technologies have been crucial in enabling the development and use of cloud platforms. Web services allow applications to communicate easily over the internet: Composite applications are easily assembled from distributed web-based components using 'mashups.' If there is one technology that has contributed the most to cloud computing, it is virtualization. By decoupling the software platform from hardware resources, virtualization enables massive cloud data centers to function seamlessly in a fault-tolerant manner. Similarly, multi-tenancy allows the same software platform to be shared by multiple applications, and can thus be looked upon as application-level virtualization. Multi-tenancy is critical for developing software-as-a-service applications and Dev 2.0 platforms.

Chapter 7

Web services, AJAX and mashups

The internet is based on a universally accepted set of protocols, HTTP, DNS, and TCP/IP, that provide the foundation for web-based cloud computing offerings. In this chapter we examine three critical web-based technologies at the next level of granularity that have been instrumental in improving the usability of web-based applications: Web services are used to request for and access infrastructure services in the cloud; AJAX-based user interfaces allow web-based applications to be user friendly; finally mashups bring a new dimension to software as a service by enabling users to compose multiple SaaS applications into a single user interface.

7.1 Web services: SOAP and REST

We have discussed the genesis and evolution of web services briefly in Chapter 2. Here we give a brief technical overview of both SOAP/WSDL and REST-based web services, and also compare these in the context of their utility in building cloud-based offerings. For a more detailed description of web services protocols, see [30].

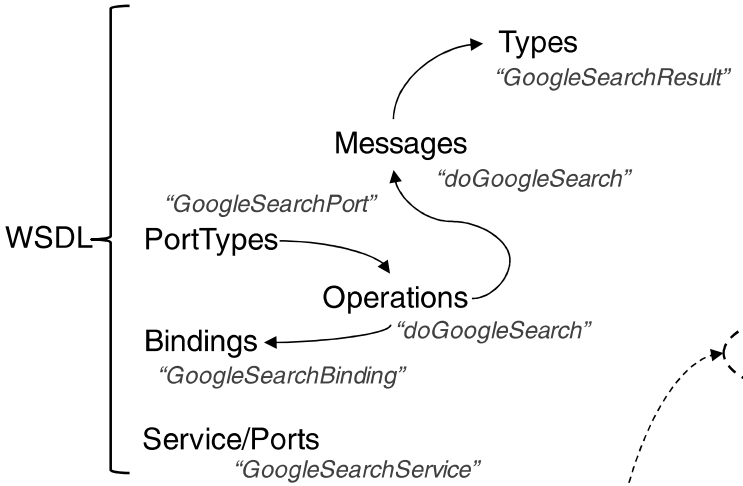
7.1.1 SOAP/WSDL Web services

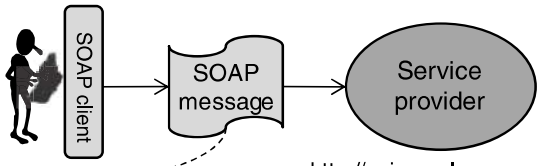
SOAP/WSDL web services evolved from the need to programmatically interconnect web-based applications. As a result SOAP/WSDL web services are essentially a form of remote procedure calls over HTTP, while also including support for nested structures (objects) in a manner similar to earlier extensions of RPC, such as CORBA; we shall return to this point later.

The elements of a SOAP/WSDL web service are illustrated in Figure 7.1, using as an example the service provided by Google for searching the web. A client application can invoke this web service by sending a SOAP request in XML form, as illustrated at the bottom left of the figure, to the designated service URL. The specifications of the service, including the service URL and other parameters, are made available by the service provider (in this case Google) as another XML file, in WSDL[1] format, as illustrated in the rest of the figure. The WSDL file specifies the service endpoint, i.e. the URL that responds to SOAP requests to this web service, as shown in the bottom right of the figure. Above this are a number of *port types*, within which are listed the *operations* (functions, methods) that are included in this service, along with their input and output parameter types; for example the operation `doGoogleSearch` has input and output messages *doGoogleSearch* and *doGoogleSearchResponse* respectively. The types of these messages are also specified in detail in the WSDL file, as XML schemas. For example in the case of a `doGoogleSearch` operation, the input messages are composed of simple types (i.e. strings, etc.), whereas the output, i.e. search result, is a complex type comprising of an array of results whose schema is also specified in the WSDL (not shown in the figure). Finally, the WSDL *binding* links these abstract set of operations with concrete transport protocols and serialization formats.

SOAP documents, i.e. the XML messages exchanged over HTTP, comprise of a *body* (as shown in bottom left of the figure) as well as an optional *header* that is an extensible container where message layer information can be encoded for a variety of purposes such as security, quality of service, transactions, etc. A number of WS-* specifications have been developed to incorporate additional features into SOAP web services that extend and utilize the header container: For example, WS-Security for user authentication, WS-Transactions to handle atomic transactions spanning multiple service

[1] WSDL: web service definition language.






```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelopexmlns:SOAP-ENV=http://  ../>
 <SOAP-ENV:Body>
  <ns1:doGoogleSearch xmlns:ns1="urn:GoogleSearch"  ….>
    <keyxsi:type="xsd:string">……</key>
   <qxsi:type="xsd:string">Cloud Computing</q>
   …
  </ns1:doGoogleSearch>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

```
<definitions name="GoogleSearch" targetNamespace="urn:GoogleSearch“  ….>
<!--Types for search -result elements, directory categories -->
<types>
<xsd:schemaxmlns= …>
<xsd:complexTypename="GoogleSearchResult">
 <xsd:all>
  <xsd:elementname="resultElements"  type="typens:ResultElementArray" />
  <xsd:elementname="searchQuery" type="xsd:string" />
  ….
 </xsd:all>
….
</types>
<!--Messages for Google Web APIs -….-->
 <message name="doGoogleSearch">
  <part name="key" type="xsd:string" />
  <part name="q" type="xsd:string" />
  ….
<!--Port for Google Web APIs, "GoogleSearch" -->
 <portTypename="GoogleSearchPort">
  <operation name="doGoogleSearch">
   <input message="typens:doGoogleSearch" />
   <output message="typens:doGoogleSearchResponse" />
  </operation>
 </portType>
 ….
<!--Binding for Google Web APIs -RPC, SOAP over HTTP -->
<binding name="GoogleSearchBinding" type="typens:GoogleSearchPort">
 <soap:bindingstyle="rpc" transport="http://schemas.xmlsoap.org/soap/http"  />
 <operation name="doGoogleSearch">
  <soap:operationsoapAction="urn:GoogleSearchAction" />
 <input>
   <soap:bodyuse="encoded" namespace="urn:GoogleSearch" encodingStyle= …/>
  </input>
 …
 </operation>
….
</binding>
<!--Endpoint for Google Web APIs –>
 <service name="GoogleSearchService">
  <port name="GoogleSearchPort" binding="typens:GoogleSearchBinding">
   <soap:addresslocation="http://api.google.com/search/beta2"  />
  </port>
</service>
</definitions>
```

FIGURE 7.1. SOAP/WSDL Web Service

requests across multiple service providers, WS-Resource Framework enabling access to resource *state* behind a web service (even though each web service is inherently stateless) and WS-Addressing to allow service endpoints to be additionally addressed at the messaging level so that service requests can be routed on non-HTTP connections (such as message queues) behind an HTTP service facade, or even for purely internal application integration.

The origin of the rather complex structure used by the SOAP/WSDL approach can be traced back to the RPC (remote procedure call) standard and its later object oriented variants, such as CORBA. In the original RPC protocol (also called SUN RPC), the client-server interface would be specified by a `<..>.x` file, from which client and server stubs in C would be generated, along with libraries to handle complex data structures and data serialization across machine boundaries. In CORBA, the `.x` files became IDL descriptions using a similar overall structure; Java RMI (remote method invocation) also had a similar structure using a common Java interface class to link client and server code. SOAP/WSDL takes the same approach for enabling RPC over HTTP, with WSDL playing the role of `.x` files, IDLs or interface classes.

7.1.2 REST web services

Representational State Transfer (REST) was originally introduced as an architectural style for large-scale systems based on distributed *resources*, one of whose embodiments is the hypertext driven HTML-based web itself. The use of REST as a paradigm for service-based interaction between application programs began gaining popularity at about the same time as, and probably in reaction to, the SOAP/WSDL methodology that was being actively propagated by many industry players at the time, such as IBM and Microsoft.

REST web services are merely HTTP requests to URIs,[2] using exactly the four methods GET, POST, PUT and DELETE allowed by the HTTP protocol. Each URI identifies a resource, such as a record in a database. As an example, consider accessing a customer record with the REST service `http://x.y.com/customer/11998`, which returns the record in XML format. In case the record contains links (foreign keys) to related

[2] See Chapter 2.

records, such as the customer's accounts or address, links to these are embedded in the returned XML, such as `http://x.y.com/account/334433`. Alternatively, these links might be directly accessed via a REST service `http://x.y.com /customer/11998/accounts`. The client application merely accesses the URIs for the resources being managed in this 'RESTful' manner using simple HTTP requests to retrieve data. Further, the same mechanism can allow manipulation of these resources as well; so a customer record may be retrieved using a GET method, modified by the client program, and sent back using a PUT or a POST request to be updated on the server.

Figure 7.2 illustrates REST web services with the above example as well as two real-life examples using Yahoo! and Google, both of whom also provide a REST web service interface to their core search engine. Notice that the URLs of these search services include parameters (*appid* and *query* for Yahoo!, *ver* and *q* for Google); strictly speaking these service definitions deviate from the 'strong' REST paradigm, where resources are defined by pure URIs alone. In principle, such purity could have easily been maintained: Note that *version* is part of the URI in the Yahoo! service while it is a parameter in the case of Google, which need not have been the case; the input URL would simply need to have been processed differently. In practice however, the use of parameters in REST services has now become widespread.

Note that while the Yahoo! service returns XML, the Google Service returns JSON (JavaScript Serialized Object Notation). A JSON string is simply a piece of JavaScript code that defines a 'map'[3] data structure in that language. The advantage of using JSON is that XML parsing is avoided; instead, the response string is simply evaluated by client-side JavaScript code (e.g. `res=eval(response)`). In the case of our Google service, this would allow the results to be accessed directly from JavaScript, so that `res["responseData"]["results"][0]["url"]` returns the first result URL, etc. As far as REST is concerned, this is perfectly legal since in theory *any* allowable internet media types, such as HTML, XML, text, pdf or doc, can be exchanged via a REST service. Finally, we mention in passing that client and server authentication is easily handled in REST just as with normal HTML web pages by using SSL (i.e. HTTPS).

[3] A set of key-value pairs, for example {'a':1, 'b':2}.

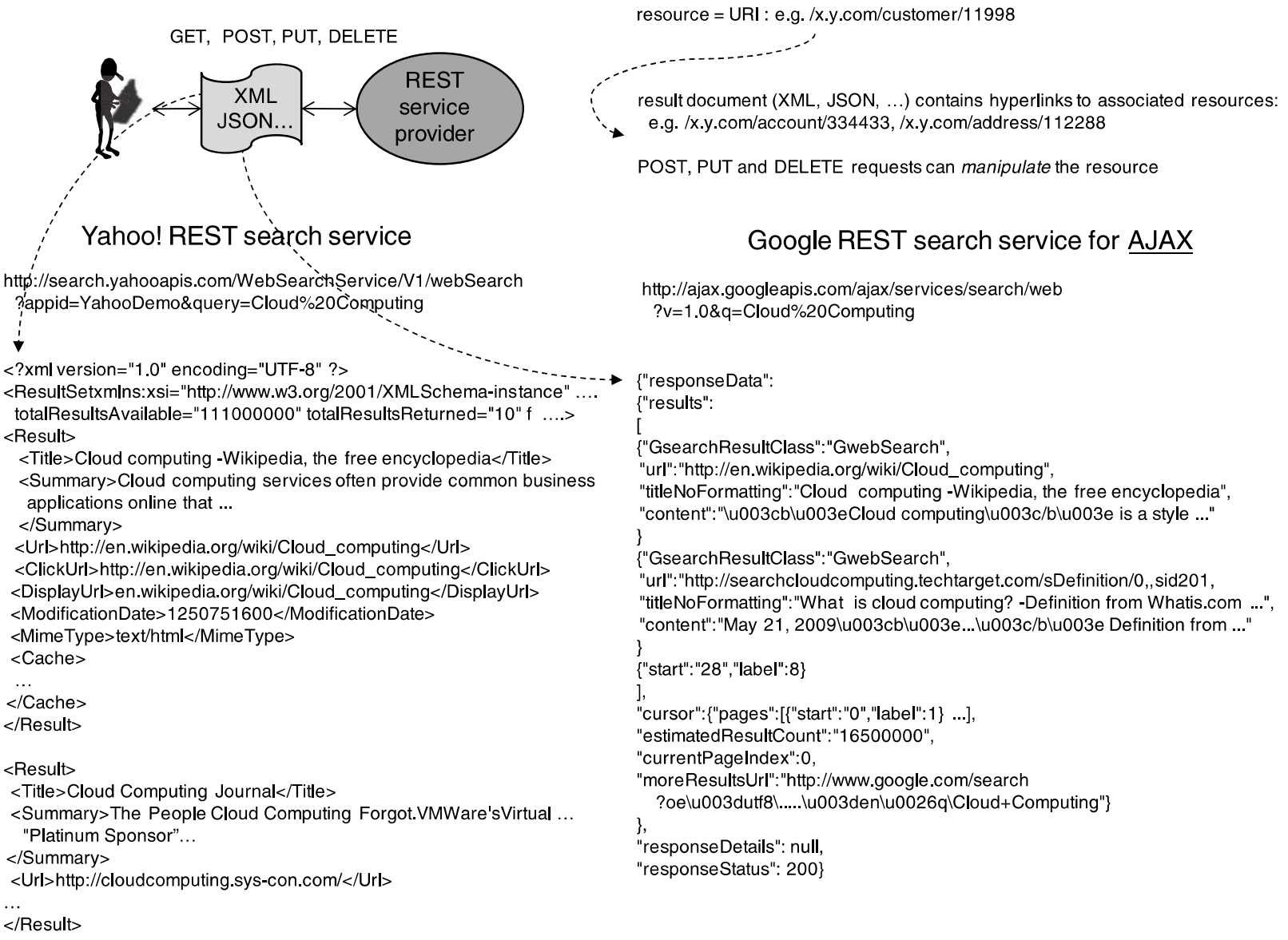


FIGURE 7.2. REST web services

7.2 SOAP versus REST

Many discussions of SOAP versus REST focus on the point that encoding services as SOAP/WSDL makes it difficult to expose the semantics of a web service in order for it to be easily and widely understood, so that many different providers can potentially offer the same service. Search is a perfect example. It is abundantly clear that the SOAP/WSDL definition of Google search does not in any way define an 'obvious' standard, and it is just as acceptable for an alternative API to be provided by other search engines. However, in the case of REST, there is the *potential* for such standardization: If for example, the REST standard for search were `http://<provider-URL>/<query-string>`, multiple providers could make this available; the response documents in XML could be self-describing by referring to provider specific name spaces where needed but adhering to a publicly specified top-level schema. We do not take a view on this aspect of the SOAP vs. REST debate, since standardization and reuse are difficult goals. As is apparent from the two very different REST APIs for web search, it is not SOAP or REST that drives standardization. Nevertheless, the relative simplicity of creating and using REST-based services as compared to the more complex SOAP/WSDL approach *is* immediately apparent from our examples. Further, REST can avoid expensive XML parsing by using alternatives such as JSON. So our view is that the case for using SOAP/WSDL needs to be explicitly made depending on the context, with REST being the option of choice from the perspective of simplicity as well as efficiency.

To examine when SOAP services may in fact be warranted, we now compare the SOAP and REST paradigms in the context of programmatic communication between applications deployed on different cloud providers, or between cloud applications and those deployed in-house. In Table 7.1 we compare these along six dimensions: The *location* where servers providing the service can reside; how *secure* the interaction is; whether *transactions* can be supported; how dependent the protocol is on HTTP *technology*; the extent of development *tools* and support required; the *efficiency* of the resulting implementations; and finally the software development *productivity* to be expected using each. We conclude from this analysis that for most requirements SOAP is an overkill; REST interfaces are simpler, more efficient and cheaper to develop and maintain. The shift from SOAP to REST especially in the cloud setting is apparent: The Google SOAP service is now deprecated, and essentially replaced by the REST API using JSON. While Amazon web services publish both SOAP as well as REST APIs, the SOAP APIs are hardly used

TABLE 7.1 SOAP/WSDL versus REST

	SOAP/WSDL	REST	Comments
Location	Some endpoints can be behind corporate networks on non-HTTP connects, e.g. message queues	All endpoints must be on the internet	Complex B2B scenarios require SOAP
Security	HTTPS which can be augmented with additional security layers	Only HTTPS	Very stringent security needs can be addressed only by SOAP
Efficiency	XML parsing required	XML parsing can be avoided by using JSON	REST is lighter and more efficient
Transactions	Can be supported	No support	Situations requiring complex multi-request / multi-party transactions need SOAP
Technology	Can work without HTTP, e.g. using message queues instead	Relies on HTTP	REST is for pure internet communications and cannot mix other transports
Tools	Sophisticated tools required (and are available) to handle client and server development	No special tools required especially if using JSON	REST is lighter and easier to use
Productivity	Low, due to complex tools and skills needed	High, due to simplicity	REST is faster and cheaper for developers to use

(15 percent is a number quoted on the web). In our opinon REST web services will gradually overtake SOAP/WSDL, and it is likely that mechanisms to address more complex functionality, such as transactions, will also be developed for REST in the near future.

7.3 AJAX: ASYNCHRONOUS 'RICH' INTERFACES

Traditional web applications interact with their server components through a sequence of HTTP GET and POST requests, each time refreshing the HTML page in the browser. The use of client-side (i.e., in-browser) JavaScript is limited to field validations and some user interface effects such as animations, hiding or unhiding parts of the page etc. Apart from any such manipulations, between the time a server request is made and a response obtained, the browser is essentially idle. Often one server request entails retrieving data from many data sources, and even from many different servers; however requests are still routed through a single web server that acts as a gateway to these services.

We described the historical evolution of the AJAX paradigm in Chapter 2. Using AJAX JavaScript programs running in the browser can make *asynchronous* calls to the server *without* refreshing their primary HTML page. Heavy server-side processing can be broken up into smaller parts that are multiplexed with client-side processing of user actions, thereby reducing the overall response time as well as providing a 'richer' user experience. Further, client-side JavaScript can make REST service requests not only to the primary web server but also to other services on the internet, thereby enabling application integration within the browser.

From a software architecture perspective, AJAX applications no longer remain pure thin clients: Significant processing can take place on the client, thereby also exploiting the computational power of the desktop, just as was the case for client-server applications. Recall that using the client-server architecture one ran the risk of mixing user interface and business logic in application code, making such software more difficult to maintain. Similar concerns arise while using the AJAX paradigm.

Figure 7.3 illustrates how AJAX applications work; these are also called 'rich internet applications' (RIA), in comparison to traditional web applications. A base HTML page is loaded along with JavaScript code that contains the remainder of the user interface. This JavaScript program renders a 'rich' user interface that can often look like a traditional client-server application. When data is required from the server, asynchronous requests are made via REST

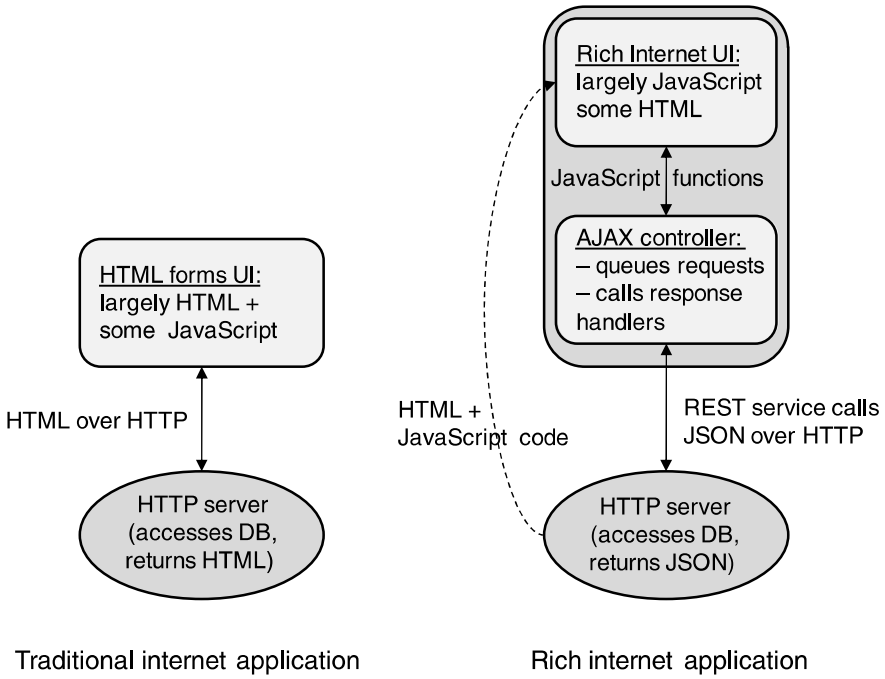


FIGURE 7.3. Rich internet applications with AJAX

web services, which return JSON structures that are directly used by the JavaScript code running in the browser.

Because of the nature of the HTTP protocol, a web server expects that an incoming request from a single client session will not be followed by another until the server has responded to the first request. If a client violates this protocol by sending many requests at a time, at best these will be ignored and at worst some poorly implemented servers may even crash! Therefore an AJAX controller is required to serialize the asynchronous requests being made to the server; each request is queued and sent to the server only after the previous request has returned with a response. Each response triggers a handler function which is registered with the controller when placing the request.

Using AJAX, highly interactive yet completely browser-based user interfaces become possible. Using this approach, software as a service application can begin to provide a user experience similar to thick client applications which typically run inside the enterprise, thereby making SaaS offerings more acceptable to enterprise users. Further, using AJAX, services from multiple cloud providers can be integrated within the browser, using JavaScript, instead of using more complex server-side integration mechanisms based on web services. Also, unlike server-side integration that needs to be performed

by corporate IT, simple JavaScript-based integrations can often be performed by business units themselves, in much the same manner as Dev 2.0 platforms allow simple applications to be developed by end-users. This has resulted in a proliferation of *user-driven* integration of services from multiple cloud-based services and SaaS applications, often without the knowledge and participation of corporate IT.

7.4 Mashups: user interface services

We have seen that using AJAX a JavaScript user interface can call many different web services directly. However, the presentation of data from these services within the user interface is left to the calling JavaScript program. Mashups take the level of integration one step further, by including the presentation layer for the remote service along with the service itself.

Figure 7.4 illustrates mashups, again using the Google search service that is also available as a mashup. In order to display a Google search box, a developer only needs to reference and use some JavaScript *code* that is automatically downloaded by the browser. This code provides a 'class' `google` that provides the AJAX API published by Google as its methods. (Strictly speaking this is a function, since JavaScript is not truly object orientated, but in colloquial usage such JavaScript functions are referred to as classes.) User code calls these methods to instantiate a search control 'object' and render it within the HTML page dynamically after the page has loaded. Notice that there is no

```
<html xmlns="http://www.w3.org/1999/xhtml">
 <head>
  <script src="http://www.google.com/jsapi"  type="text/javascript">
  </script>
  <script type="text/javascript">
  google.load('search', '1.0');
  function OnLoad() {
    var searchControl= new google.search.SearchControl();
    searchControl.addSearcher(new google.search.WebSearch());
    searchControl.draw(document.getElementById("searchcontrol"));
  }
  google.setOnLoadCallback(OnLoad, true);
  </script>
 </head>
 <body>
  <div id="searchcontrol">Loading</div>
 </body>
</html>
```

Figure 7.4. Mashup example

AJAX controller or REST service visible to the user; all this is hidden within the API methods. Recall that the purpose of an AJAX controller was to serialize HTTP requests from a running session to a *particular* web server: There is no need for serialization across calls to different service providers, and therefore it is perfectly okay for different mashup services to provide their own AJAX controllers within their APIs.

From a user perspective, mashups make it easy to consume web services. In fact, the actual service call need not even be a REST service, and may instead involve proprietary AJAX-based interaction with the service provider. In this sense, mashups make the issue of a published service standard using REST or SOAP/WSDL irrelevant; the only thing that is published is a JavaScript library which can be downloaded at runtime and executed by a client application.

At the same time, the fact that mashups require downloading and running *foreign* code is a valid security concern especially in the enterprise scenario. JavaScript code normally cannot access resources on the client machine apart from the browser and network, so it may appear that there is no real security threat, unlike say ActiveX controls which have essentially complete access to the desktop once a user installs them. However, this may no longer remain the case in the future: For example Google Gears is a framework that enables offline operation of applications by caching data on the client desktop. This presents a potential security threat, though not as serious as ActiveX controls: For example, if a user has installed Gears for some reason, such as accessing Gmail in offline mode, another site the user accesses may ask for permission to use Gears (note that such a prompt is always shown, making Gears a bit safer), and if granted store some executables on a user's disk, and present the user with a link which runs these as a side effect. As a result of such potential security risks, enterprise adoption of mashups has been slower than warranted by the technology's advantages.

Note that Google initially published a SOAP/WSDL service but later replaced it with an AJAX mashup API, and as a by product also made available the REST web service which we discussed earlier. Another popular mashup is Google Maps. It is becoming increasingly apparent that mashup-based integration of cloud-based services is easy, popular and represents the direction being taken by the consumer web service industry. Enterprise usage of mashup technology is only a matter of time, not only in the context of cloud-based offerings, but also for integrating internal applications with cloud services, as well as amongst themselves.

Chapter 8

Virtualization technology

If one had to choose a single technology that has been most influential in enabling the cloud computing paradigm, it would have to be virtualization. As we have seen earlier in Chapter 1, virtualization is not new, and dates back to the early mainframes as a means of sharing computing resources amongst users. Today, besides underpinning cloud computing platforms, virtualization is revolutionizing the way enterprise data centers are built and managed, paving the way for enterprises to deploy 'private cloud' infrastructure within their data centers.

8.1 Virtual machine technology

We begin with an overview of virtual machine technology: In general, any means by which many different users are able simultaneously to interact with a computing system while each perceiving that they have an entire 'virtual machine' to themselves, is a form of virtualization. In this general sense, a traditional multiprogramming operating system, such as Linux, is also a form of virtualization, since it allows each user process to access system resources oblivious of other processes. The abstraction provided to each process is the set of OS system calls and any hardware instructions accessible to user-level processes. Extensions, such as 'user mode Linux' [17] offer a more complete virtual abstraction where each user is not even aware of other user's processes, and can login as an administrator, i.e. 'root,' to their own seemingly

private operating system. 'Virtual private servers' are another such abstraction [36]. At a higher level of abstraction are virtual machines based on high-level languages, such as the Java virtual machine (JVM) which itself runs as an operating system process but provides a system-independent abstraction of the machine to an application written in the Java language. Such abstractions, which present an abstraction at the OS system call layer or higher, are called *process virtual machines*. Some cloud platforms, such as Google's App Engine and Microsoft's Azure, also provide a process virtual machine abstraction in the context of a web-based architecture.

More commonly, however, the virtual machines we usually refer to when discussing virtualization in enterprises or for infrastructure clouds such as Amazon's EC2 are *system virtual machines* that offer a complete hardware instruction set as the abstraction provided to users of different virtual machines. In this model many system virtual machine (VM) instances share the same physical hardware through a virtual machine monitor (VMM), also commonly referred to as a *hypervisor*. Each such system VM can run an independent operating system instance; thus the same physical machine can have many instances of, say Linux *and* Windows, running on it simultaneously. The system VM approach is preferred because it provides complete isolation between VMs as well as the highest possible flexibility, with each VM seeing a complete machine instruction set, against which any applications for that architecture are guaranteed to run.

It is the virtual machine monitor that enables a physical machine to be virtualized into different VMs. Where does this software itself run? A *host* VMM is implemented as a process running on a *host* operating system that has been installed on the machine in the normal manner. Multiple *guest* operating systems can be installed on different VMs that each run as operating system processes under the supervision of the VMM. A *native* VMM, on the other hand, does not require a host operating system, and runs directly on the physical machine (or more colloquially on 'bare metal'). In this sense, a native VMM can be viewed as a special type of operating system, since it supports multiprogramming across different VMs, with its 'system calls' being hardware instructions! Figure 8.1 illustrates the difference between process virtual machines, host VMMs and native VMMs. Most commonly used VMMs, such as the open source Xen hypervisor as well as products from VMware are available in both hosted as well as native versions; for example the hosted Xen (HXen) project and VMware Workstation products are hosted VMMs, whereas the more popularly used XenServer (or just Xen) and VMware ESX Server products are native VMMs.

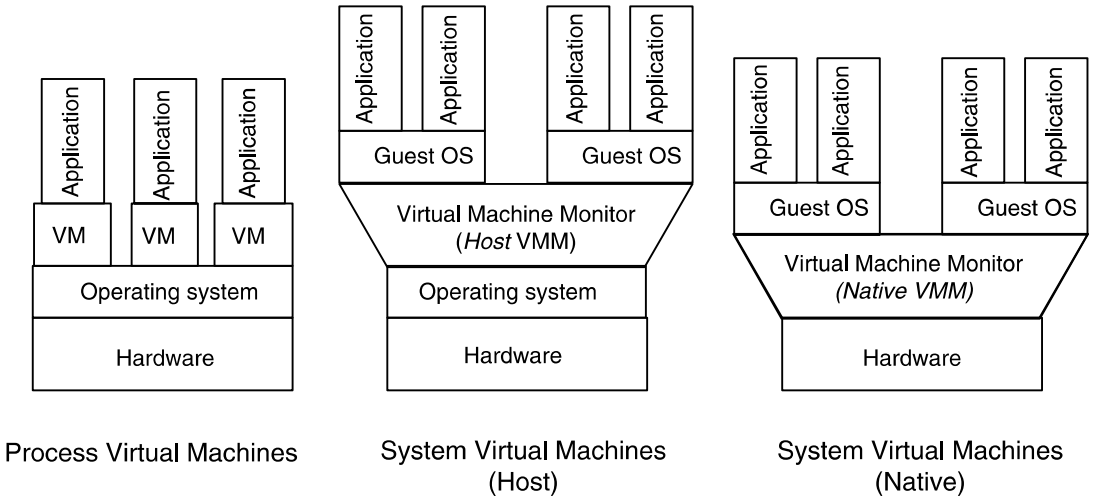


FIGURE 8.1. Virtual machines

In the next section we shall briefly describe how system virtual machines are implemented *efficiently* and how individual virtual machines actually run.

8.1.1 System virtual machines

A system virtual machine monitor needs to provide each virtual machine the illusion that it has access to a complete independent hardware system through a full instruction set. In a sense, this is very similar to the need for a time-sharing operating system to provide different processes access to hardware resources in their allotted time intervals of execution. However, there are fundamental differences between the 'virtual machine' as perceived by a traditional operating system processes and a true system VM:

1. Processes under an operating system are allowed access to hardware through system calls, whereas a system VMM needs to provide a full hardware instruction set for use by each virtual machine
2. Each system virtual machine needs to be able to run a full operating system, while itself maintaining isolation with other virtual machines.

Going forward we will focus our discussion on native VMMs that run directly on the hardware, like an operating system; native VMMs are more efficient and therefore the ones used in practice within enterprises as well as cloud platforms. One way a native system VMM could work is by *emulating* instructions of the target instruction set and maintaining the state of

different virtual machines at all levels of memory hierarchy (including registers etc.) *indirectly* in *memory* and switching between these as and when required, in a manner similar to how virtual memory page tables for different processes are maintained by an operating system. In cases where the target hardware instruction set and actual machine architecture are different, emulation and indirection is unavoidable, and, understandably, inefficient. However, in cases where the target instruction set is the same as that of the actual hardware on which the native VMM is running, the VMM can be implemented more efficiently.

An *efficient* native VMM attempts to run the instructions of each of is virtual machines natively on the hardware, and while doing so also maintain the state of the machine at its proper location in the memory hierarchy, in much the same manner as an operating system runs process code natively as far as possible except when required.

Let us first recall how an operating system runs a process: The process state is first loaded into memory and registers, then the program counter is reset so that process code runs from thereon. The process runs until a timer event occurs, at which point the operating system switches the process and resets the timer via a special *privileged* instruction. The key to this mechanism is the presence of privileged instructions, such as resetting the timer interrupt, which cause a *trap* (a program generated interrupt) when run in 'user' mode instead of 'system' mode. Thus, no *user* process can set the timer interrupt, since this instruction is privileged and always traps, in this case to the operating system.

Thus, it should be possible to build a VMM in exactly the same manner as an operating system, by trapping the privileged instructions and running all others natively on the hardware. Clearly the privileged instructions themselves need to be *emulated*, so that when an operating system running in a virtual machine attempts to, say, set the timer interrupt, it actually sets a *virtual* timer interrupt. Such a VMM, where only privileged instructions need to be emulated, is the most efficient native VMM possible, as formally proved in [45].

However, in reality it is not always possible to achieve this level of efficiency. There are some instruction sets (including the popular Intel IA-32, better known as x86) where some non-privileged instructions behave differently depending on whether they are called in user mode or system mode. Such instruction sets implicitly assume that there will be only one operating system (or equivalent) program that needs access to privileged instructions, a natural assumption in the absence of virtualization. However, such instructions pose a problem for virtual machines, in which the operating system is actually

running in user mode rather than system mode. Thus, it is necessary for the VMM to also emulate such instructions in addition to all privileged instructions. Newer editions of the x86 family have begun to include 'hardware support' for virtualization, where such anomalous behavior can be rectified by exploiting additional hardware features, resulting in a more efficient implementation of virtualization: For example, Intel's VT-x ('Vanderpool') technology includes a new VMX mode of operation. When VMX is enabled there is a new 'root' mode of operation exclusively for use by the VMM; in non-root mode all standard modes of operation are available for the OS and applications, including a 'system' mode which is at a lower level of privilege than what the VMM enjoys. We do not discuss system virtual machines in more detail here, as the purpose of this discussion was to give some insight into the issues that are involved through a few examples; a detailed treatment can be found in [58].

8.1.2 Virtual machines and elastic computing

We have seen how virtual machine technology enables decoupling physical hardware from the virtual machines that run on them. Virtual machines can have different instruction sets from the physical hardware if needed. Even if the instruction sets are the same (which is needed for efficiency), the size and number of the physical resources seen by each virtual machine need not be the same as that of the physical machine, and in fact will usually be different. The VMM partitions the actual physical resources in time, such as with I/O and network devices, or space, as with storage and memory. In the case of multiple CPUs, compute power can also be partitioned in time (using traditional time slices), or in space, in which case each CPU is reserved for a subset of virtual machines.

The term 'elastic computing' has become popular when discussing cloud computing. The Amazon 'elastic' cloud computing platform makes extensive use of virtualization based on the Xen hypervisor. Reserving and booting a server instance on the Amazon EC cloud provisions and starts a virtual machine on one of Amazon's servers. The configuration of the required virtual machine can be chosen from a set of options (see Chapter 5). The user of the 'virtual instance' is unaware and oblivious to which physical server the instance has been booted on, as well as the resource characteristics of the physical machine.

An 'elastic' multi-server environment is one which is completely virtualized, with all hardware resources running under a set of *cooperating* virtual

machine monitors and in which provisioning of virtual machines is largely automated and can be dynamically controlled according to demand. In general, any multi-server environment can be made 'elastic' using virtualization in much the same manner as has been done in Amazon's cloud, and this is what many enterprise virtualization projects attempt to do. The key success factors in achieving such elasticity is the degree of *automation* that can be achieved across multiple VMMs working together to maximize utilization. The *scale* of such operations is also important, which in the case of Amazon's cloud runs into tens of thousands of servers, if not more. The larger the scale, the greater the potential for amortizing demand effciently across the available capacity while also giving users an illusion of 'infinite' computing resources.

Technology to achieve elastic computing at scale is, today, largely proprietary and in the hands of the major cloud providers. Some automated provisioning technology is available in the public domain or commercially off the shelf (see Chapter 17), and is being used by many enterprises in their internal data center automation efforts. Apart from many startup companies, VMware's VirtualCentre product suite aims to provide this capability through its 'VCloud' architecture.

We shall discuss the features of an elastic data center in more detail later in this chapter; first we cover virtual machine migration, which is a pre-requisite for many of these capabilities.

8.1.3 Virtual machine migration

Another feature that is crucial for advanced 'elastic' infrastructure capabilities is 'in-flight' migration of virtual machines, such as provided in VMware's VMotion product. This feature, which should also be considered a key component for 'elasticity,' enables a virtual machine running on one physical machine to be suspended, its state saved and transported to or accessed from another physical machine where it is resumes execution from exactly the same state.

Virtual machine migration has been studied in the systems research community [49] as well as in related areas such as grid computing [29]. Migrating a virtual machine involves capturing and copying the entire state of the machine at a snapshot in time, including processor and memory state as well as all virtual hardware resources such as BIOS, devices or network MAC addresses. In principle, this also includes the entire disk space, including system and user directories as well as swap space used for virtual memory operating system scheduling. Clearly, the complete state of a typical server is likely to be quite large. In a closely networked multi-server environment, such as a cloud data

center, one may assume that some persistent storage can be easily accessed and mounted from different servers, such as through a storage area network or simply networked file systems; thus a large part of the system disk, including user directories or software can easily be transferred to the new server, using this mechanism. Even so, the remaining state, which needs to include swap and memory apart from other hardware states, can still be gigabytes in size, so migrating this efficiently still requires some careful design.

Let us see how VMware's VMotion carries out in-flight migration of a virtual machine between physical servers: VMotion waits until the virtual machine is found to be in a stable state, after which all changes to machine state start getting logged. VMotion then copies the contents of memory, as well as disk-resident data belonging to either the guest operating system or applications, to the target server. This is the *baseline* copy; it is not the final copy because the virtual machine continues to run on the original server during this process. Next the virtual machine is suspended and the last remaining changes in memory and state since the baseline, which were being logged, are sent to the target server, where the final state is computed, following which the virtual machine is activated and resumes from its last state.

8.2 Virtualization applications in enterprises

A number of enterprises are engaged in virtualization projects that aim to gradually relocate operating systems and applications running directly on physical machines to virtual machines. The motivation is to exploit the additional VMM layer between hardware and systems software for introducing a number of new capabilities that can potentially ease the complexity and risk of managing large data centers. Here we outline some of the more compelling cases for using virtualization in large enterprises.

8.2.1 Security through virtualization

Modern data centers are all necessarily connected to the world outside via the internet and are thereby open to malicious attacks and intrusion. A number of techniques have been developed to secure these systems, such as firewalls, proxy filters, tools for logging and monitoring system activity and intrusion detection systems. Each of these security solutions can be significantly enhanced using virtualization.

For example, many intrusion detection systems (IDS) traditionally run on the network and operate by monitoring network traffic for suspicious behavior

by matching against a database of known attack patterns. Alternatively, host-based systems run within each operating system instance where the behavior of each process is monitored to detect potentially suspicious activity such as repeated login attempts or accessing files that are normally not needed by user processes. Virtualization opens up the possibility of building IDS capabilities into the VMM itself, or at least at the same layer, i.e. above the network but below the operating system. The Livewire and Terra research projects are examples of such an approach [24, 25], which has the advantage of enabling greater isolation of the IDS from the monitored hosts while retaining complete visibility into the host's state. This approach also allows for complete mediation of interactions between the host software and the underlying hardware, enabling a suspect VM to be easily isolated from the rest of the data center.

Virtualization also provides the opportunity for more complete, user-group specific, low-level logging of system activities, which would be impossible or very difficult if many different user groups and applications were sharing the same operating system. This allows security incidents to be be more easily traced, and also better diagnosed by *replaying* the incident on a copy of the virtual machine.

End-user system (desktop) virtualization is another application we cover below that also has an important security dimension. Using virtual machines on the desktop or mobile phones allows users to combine personal usage of these devices with more secure enterprise usage by isolating these two worlds; so a user logs into the appropriate virtual machine (personal or enterprise), with both varieties possibly running simultaneously. Securing critical enterprise data, ensuring network isolation from intrusions and protection from viruses can be better ensured without compromising users' activities in their personal pursuits using the same devices. In fact some organizations are contemplating not even considering laptops and mobile devices as corporate resources; instead users can be given the flexibility to buy whatever devices they wish and use client-side virtual machines to access enterprise applications and data.

8.2.2 Desktop virtualization and application streaming

Large enterprises have tens if not hundreds of thousands of users, each having a desktop and/or one or more laptops and mobile phones that are used to connect to applications running in the enterprise's data center. Managing regular

system updates, such as for security patches or virus definitions is a major system management task. Sophisticated tools, such as IBM's Tivoli are used to automate this process across a globally distributed network of users. Managing application roll-outs across such an environment is a similarly complex task, especially in the case of 'fat-client' applications such as most popular email clients and office productivity tools, as well some transaction processing or business intelligence applications.

Virtualization has been proposed as a possible means to improve the manageability of end-user devices in such large environments. Here there have been two different approaches. The first has been to deploy all end-client systems as virtual machines on central data centers which are then accessed by 'remote desktop' tools, such as Citrix Presentation Server, Windows Terminal Services (WTS), or VNC (Virtual Network Computer). At least theoretically this is an interesting solution as it (a) eases management of updates by 'centralizing' all desktops (b) allows easier recovery from crashes by simply restarting a new VM (c) enables security checks and intrusion detection to be performed centrally and (d) with all user data being central, secures it as well as enables better data sharing and potential reduction of redundant storage use. However, this approach has never really become popular, primarily because of the need for continuous network connectivity, which in spite of the advances in corporate networks and public broadband penetration, is still not ubiquitous and 'always on.' Additionally, this approach also ignores the significant computing power available on desktops, which when added up across an enterprise can be very costly to replicate in a central data center.

The second approach is called 'application streaming.' Instead of running applications on central virtual machines, application streaming envisages maintaining only virtual machine images centrally. An endpoint client, such as a desktop, runs a hypervisor that also downloads the virtual machine image from the server and launches it on the end point client. In this manner the processing power of the end point is fully exploited, a VM image can be cached for efficiency and only incrementally updated when needed, and finally user data, which can be large, need not be centrally maintained but mounted from the local disk as soon as the virtual machine boots. Such a solution is implemented, for example, in the XenApp product from Citrix (incorporating technology from Appstream, which was acquired by Citrix). Application streaming additionally allows the isolation of personal and corporate spaces for security purposes as mentioned in the previous section.

8.2.3 Server consolidation

The most common driver for virtualization in enterprise data centers has been to consolidate applications running on possibly tens of thousands of servers, each significantly underutilized on the average, onto a smaller number of more efficiently used resources. The motivation is both efficiency as well as reducing the complexity of managing the so-called 'server sprawl.' The ability to run multiple virtual machines on the same physical resources is also key to achieving the high utilizations in cloud data centers.

Here we explore some implications and limits of consolidation through a simplified model. Suppose we wish to consolidate applications running on m physical servers of capacity c onto one physical server of capacity nc. We assume that virtual machines are either perfectly efficient, or any inefficiency has been built into the factor n. We focus on a few simple questions: (i) whether the single server should have n processors (or cores), or a clock speed n times that of each original server; (ii) how much smaller than m (the number of physical servers) can we make n while retaining acceptable performance; and finally (iii) what is the impact on power consumption and whether this changes the preferred strategy.

A simple model using basic queuing theory provides some insight: A server running at an efficiency of e can be thought of as a single server queuing system where, for whatever reason, either light load or inefficient software, the arrival rate of requests (instructions to be processed) is e times less than that which can be served by the server. In queuing theory terminology, $e = \lambda/\mu$, where λ is the arrival rate and μ the service rate. We define the average 'normalized response time' as the average time spent in the system T normalized by average time between requests, $1/\lambda$, as $r = T\lambda$. (Note: response time is a good measure of performance for transactional workloads; however it may not be the right measure for batch processing.)

Using standard queuing theory [7] we can compute r_o, the normalized response time using m physical servers as

$$r_o = \frac{e}{1-e} \tag{8.1}$$

for each of the original servers. Now consider consolidating these servers into one server with m processors, wherein the queue becomes one with m servers working at the same rate μ, servicing an arrival rate of $m\lambda$. Queuing theory

yields r_p, normalized response time using one server with p processors as

$$r_p = me + \frac{P_Q}{1-e}. \tag{8.2}$$

(Here P_Q is the 'queuing probability', which is small for light loads.[1]) If, on the other hand, we have a single server that is m times faster, we once again model it as a single server queue but with service rate $m\mu$. Since e remains unchanged, the normalized response time in this case (r_c) remains the same as r_o in (8.1).

Thus we see that for light loads, i.e., underutilized servers where $e \ll 1$, the consolidation onto a multi-processor machine versus one with faster clock speed can result in significant degradation in performance, at least as measured by average normalized response time. For heavy loads, on the other hand, the second term in (8.2) dominates, and response time is poor (large) in both cases.

Now consider the case where the single server onto which we consolidate the workload is only n times faster than the original servers. In this case we find that the normalized response time r_n is

$$r_n = \frac{me}{n-me}. \tag{8.3}$$

Using this we can see that it is possible to use a server far less powerful than the aggregate of the m original servers, as long as n/m remains reasonably large as compared to e; and if indeed $n \gg me$ then the average normalized response time degrades only linearly by the factor of n/m.

Thus we see that a simple queuing theory analysis yields some natural limits to server consolidation using virtualization. The theoretical maximum benefit, in terms of a reduction in number of servers, is $n/m = e$, at which point the system becomes unresponsive. In practice it is possible to get fairly close to this, i.e. if $n/m = e(1+\epsilon)$, then the average normalized response time becomes $1/\epsilon$. In effect, whatever the initial inefficiency, one can decide on an acceptable average normalized response time and plan the consolidation strategy accordingly.

It is instructive to bring into consideration another factor in this analysis, namely power consumption, an issue which is becoming increasingly important in data center management. Power consumption of chips is related to the voltage at which a chip operates, in particular power P grows as the square of

[1] The formula for computing P_Q can be found in [7].

the voltage, i.e. $P \propto V^2$. It is also a fact that higher clock speeds require higher voltage, with almost a linear relationship between the two. Thus, a system that runs at a clock speed n times faster than a 'base' system, will consume n^2 the power of the base system, whereas the n core system will consume only n times the power. In fact this is one of the reasons for the shift to 'multi-core' CPUs, with systems having four to eight cores per CPU being commonplace as of this writing, and CPUs with dozens of cores expected to be the norm in the near future.

Revisiting our queuing model, in the case of consolidation onto an n processor/core server, instead of one that is n times faster, we can compute the average normalized response time, call it r_P, as:

$$r_P = me + \frac{P_Q}{1 - \frac{m}{n}e}. \tag{8.4}$$

Notice that the response time remains the same as r_p in (8.2) for *light* loads, i.e., when P_Q is small. Thus the response time still degrades by a factor of m, independent of n, as compared to the faster clock speed case (8.1). However, in the case of heavy load, where the second term dominates, there is a marked degradation in performance in the multi-processor case if $n \ll m$, as compared to the $m = n$ case, i.e. (8.2).

Thus there is a trade off, at least theoretically, between reducing power consumption by consolidating onto multi-processors or multi-core CPU systems, versus improved performance on systems with faster clock speeds but at the cost of non-linear growth in power consumption per server. In practice this trade off is less significant since there are limits on how far clock speed can be increased, for both power as well as due to fundamental physical constraints. Lastly, apart from consolidation, it is important to note that individual applications implemented using multi-threaded application servers can also exploit multi-core architectures efficiently. Therefore, both enterprise as well as cloud data centers today rely almost exclusively on multi-core, multi-processor systems.

8.2.4 Automating infrastructure management

An important goal of enterprise virtualization projects is to reduce data center management costs, especially people costs through greater automation. It is important to recognize that while virtualization technology provides the

ability to automate many activities, actually designing and putting into place an automation strategy is a complex exercise that needs to be planned. Further, different levels of automation are possible, some easy to achieve through basic server consolidation, while others are more complex, requiring more sophisticated tools and technology as well as significant changes in operating procedures or even the organizational structure of the infrastructure wing of enterprise IT.

The following is a possible roadmap for automation of infrastructure management, with increasing sophistication in the use of virtualization technology at each level:

1. **Level 0 – Virtual images**: Packaging standard operating environments for different classes of application needs as virtual machines, thereby reducing the start-up time for development, testing and production deployment, also making it easier to bring on board new projects and developers. This approach is not only easy to get started with, but offers significant reduction in infrastructure management costs and saves precious development time as well.
2. **Level 1 – Integrated provisioning**: Integrated provisioning of new virtual servers along with provisioning their network and storage (SAN) resources, so that all these can be provisioned on a chosen physical server by an administrator through a single interface. Tools are available that achieve some of these capabilities (such as VMware's VirtualCenter integrated suite). In the majority of enterprises such tools are currently in the process of being explored and prototyped, with only a few enterprises having successfully deployed this level of automation on a large scale
3. **Level 2 – Elastic provisioning**: Automatically deciding the physical server on which to provision a virtual machine given its resource requirements, available capacity and projected demand; followed by bringing up the virtual machine without any administrator intervention; rather users (application project managers) are able to provision virtual servers themselves. This is the automation level provided by Amazon EC2, for example. As of this writing, and to our best knowledge, *no* large enterprise IT organization has deployed this level of automation in their internal data center at any degree of scale, though many projects are under way, using commercial products or the open source Eucalyptus tool (see Chapter 17).
4. **Level 3 – Elastic operations**: Automatically provisioning new virtual servers or migrating running virtual servers based on the need to do so, which is established through automatic monitoring of the state of all

virtual physical resources, and which can arise for a number of reasons, such as:

1. *Load balancing*, to improve response time of applications that either explicitly request for, or appear to need more resources, and depending on their business criticality.
2. *Security*, to quarantine a virtual machine that appears to have been compromised or attacked.
3. *Collocation*, to bring virtual machines that are communicating with each other physically closer together to improve performance.
4. *Fault tolerance*, to migrate applications from physical machines that have indicated possible imminent failure or need for maintenance.
5. *Fault recovery*, to provision a new instance virtual machine and launch it with the required set of applications running in order to recover from the failure of the original instance, so as to restore the corresponding business service as soon as possible.

While tools such as VMotion provide the underlying capability to migrate virtual machines 'in-flight,' as we have described in the previous section, exploiting this capability to achieve this level of automation of operations is really the *holy grail* for virtualization in enterprises, or even in infrastructure cloud platforms such as Amazon.

Virtualization projects in enterprises today are either at Level 0 or 1. Level 2 is available in Amazon EC2 in the cloud, whereas Level 3 automation has hardly ever been achieved in totality, at least with system virtual machines. Even in Amazon EC2, while monitoring and auto-scaling facilities are available, in-flight migration of virtual machines is not available, at least as of this writing.

If, however, one considers process virtual machines, such as Google App Engine, or efficient software-as-a-service providers, one can argue that to a certain extent the appearance of Level 3 is provided, since an application deployed in such a platform is essentially 'always on,' with the user not needing to be aware of any infrastructure management issues. Taking a process VM or even application virtualization (i.e. Dev 2.0) route may enable enterprises to provide pockets of services that can appear to achieve nearly Level 3 elastic automation, whereas achieving this degree of automation at a lower level of abstraction, such as system virtual machines is likely to be much harder to deploy at a large scale.

Where to start? An enterprise virtualization strategy needs to systematically plan which classes of applications should be moved to a virtual environment as well as whether and when the progression to increasing levels of automation should be attempted. Often the best place to start a virtualization exercise is

within the IT organization itself, with the 'test and dev' environments that are used by developers in application development projects. Developers regularly require many of the capabilities enabled by virtualization, such as being able to manage project-specific sets of standard operating environments, re-create and re-start servers from a check pointed state during functional testing, or provision servers of different capacities for performance testing. As an additional benefit, having developers experience virtualization during application development also makes supporting applications in a virtualized production environment much easier. Finally, exactly the same argument holds for cloud computing as well; using a cloud data center for development is a useful first step before considering production applications in the cloud.

8.3 Pitfalls of virtualization

As our discussion so far has revealed, virtualization is critical for cloud computing and also promises significant improvements within in-house data centers. At the same time it is important to be aware of some of the common pitfalls that come with virtualization:

1. Application deployments often replicate application server and database instances to ensure fault tolerance. Elastic provisioning results in two such replicas using virtual servers deployed on the same physical server. Thus if the physical server fails, both instances are lost, defeating the purpose of replication.
2. We have mentioned that virtualization provides another layer at which intrusions can be detected and isolated, i.e., the VMM. Conversely however, if the VMM itself is attacked, multiple virtual servers are affected. Thus some successful attacks can spread more rapidly in a virtualized environment than otherwise.
3. If the 'server sprawl' that motivated the building of a virtualized data center merely results in an equally complex 'virtual machine sprawl,' the purpose has not been served, rather the situation may become even worse than earlier. The ease with which virtual servers and server images are provisioned and created can easily result in such situations if one is not careful.
4. In principle a VMM can partition the CPU, memory and I/O bandwidth of a physical server across virtual servers. However, it cannot ensure that these resources are made available to each virtual server in a synchronized manner. Thus the fraction of hardware resources that a virtual server is actually able to utilize may be less than what has been provisioned by the VMM.

Chapter 9

Multi-tenant software

Applications have traditionally been developed for use by a single enterprise; similarly enterprise software products are also developed in a manner as to be independently deployed in the data center of each customer. The data created and accessed by such applications usually belongs to one organization. As we discussed earlier in Chapter 3, hosted SaaS platforms require a single application code to run on data of multiple customers, or 'tenants'; such behavior is referred to as multi-tenancy. In this chapter we examine different ways to achieve multi-tenancy in application software.

Before proceeding it is important to also note that virtualization, as discussed in the previous chapter, is also a mechanism to achieve multi-tenancy at the system level. In a virtualized environment, each 'tenant' could be assigned its own set of virtual machines. Here we examine alternatives for implementing multi-tenancy through application software architecture rather than at the system level using virtual machines. Thus, such multi-tenancy can also be termed *application-level* virtualization. Multi-tenancy and virtualization are both two sides of the same coin; the aim being to share resources while isolating users from each other: hardware resources in the case of system-level virtualization and software platforms in the case of multi-tenancy.

9.1 Multi-entity support

Long before ASPs and SaaS, large globally distributed organizations often needed their applications to support multiple organizational units, or 'entities,' in a segregated manner. For example, consider a bank with many branches needing to transition from separate branch specific installations of its core banking software to a centralized deployment where the same software would run on data from all branches. The software designed to operate at the branch level clearly could not be used directly on data from all branches: For example branch-level users should see data related only to their branch and branch-wise accounting should consider transactions segregated by branch. If there was a need to enhance the system, say by introducing a new field, such a change would need to apply across all branches; at the same time, sometimes branch specific extensions would need to be supported as well. These requirements are almost exactly the same as for multi-tenancy! In a multi-entity scenario there are also additional needs, such as where a subset of users needed to be given access to data from all branches, or a subset of branches, depending on their position in an organizational hierarchy. Similarly, some global processing would also need to be supported, such as inter-branch reconciliation or enterprise-level analytics, without which the benefits of centralization of data might not be realized. Such advanced features could be implemented using 'data access control' as covered later in Section 9.4. We first focus on basic multi-entity support as it will lead us naturally to understand how multi-tenancy can be implemented.

Figure 9.1 depicts the changes that need to be made in an application to support basic multi-entity features, so that users only access data belonging to their own units. Each database table is appended with a column (`OU_ID`) which marks the organizational unit each data record belongs to. Each database query needs to be appended with a condition that ensures that data is filtered depending on the organizational unit of the currently logged-in user, which itself needs to be set in a variable, such as `current_user_OU`, during each transaction. An exactly similar mechanism can be used to support multi-tenancy, with `OU_ID` now representing the customer to whom data records belong. Note that the application runs on a single schema containing data from all organizational units; we shall refer to this as the *single schema* model.

Many early implementations of SaaS products utilized the single schema model, especially those that built their SaaS applications from scratch. One advantage of the single schema structure is that upgrading functionality of the

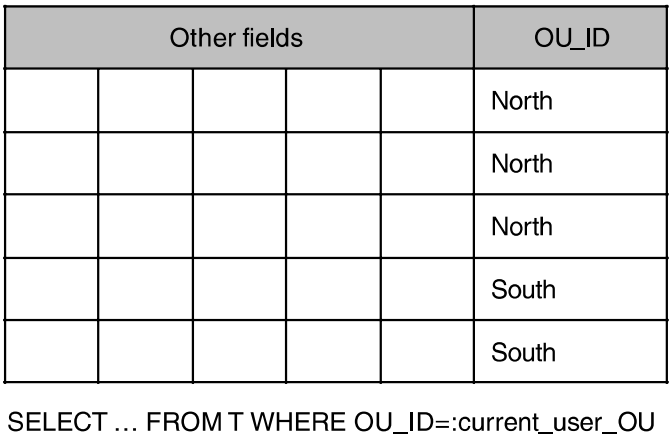


FIGURE 9.1. Multi-entity implementation

application, say by adding a field, can be done at once for all customers. At the same time, there are disadvantages: Re-engineering an existing application using the single schema approach entails significant re-structuring of application code. For a complex software product, often having millions of lines of code, this cost can be prohibitive. Further, while modifications to the data model can be done for all customers at once, it becomes difficult to support customer specific extensions, such as custom fields, using a single schema structure. Meta-data describing such customizations, as well as the data in such extra fields has to be maintained separately. Further, it remains the responsibility of the application code to interpret such meta-data for, say, displaying and handling custom fields on the screen. Additionally, any queries that require, say, filtering or sorting on these custom fields become very complex to handle. Some of these issues can be seen more clearly through the example in Figure 9.2 that depicts a multi-tenant architecture using a single schema model which also supports custom fields:

In the single schema model of Figure 9.2, a Custom Fields table stores meta-information and data values for *all* tables in the application. Mechanisms for handling custom fields in a single schema architecture are usually variants of this scheme. Consider a screen that is used to retrieve and update records in the Customer table. First the record from the main table is retrieved by `name`, suitably filtered by the `OU` attribute of the logged in user. Next, custom fields along with their values are retrieved from the Custom Fields table, for *this* particular record *and* the `OU` of the logged in user. For example, in `OU` 503, there are two custom fields as displayed on the screen, but only one in `OU`

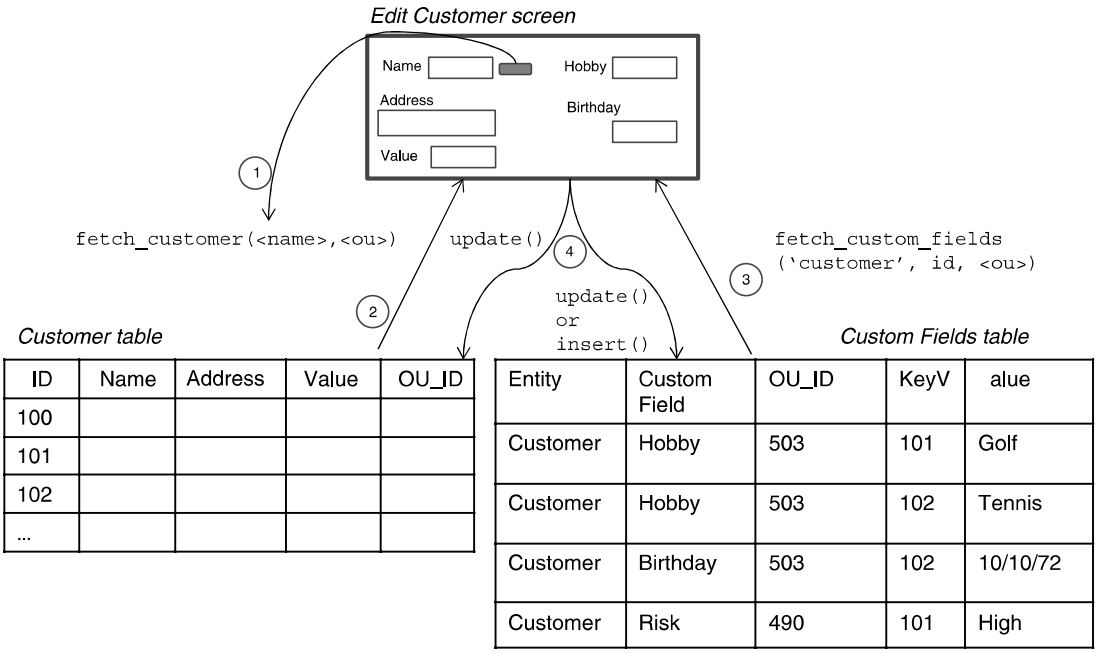


FIGURE 9.2. Multi-tenancy using a single schema

490, and none otherwise. Furthermore, some records may have missing values for these fields, so while saving the record care must be taken to appropriately either insert or update records in the Custom Fields table.

The above example is a simple case; more complex requirements also need to be handled, for example where a list of records is to be displayed with the ability to sort and filter on custom fields. It should be clear from this example that the single schema approach to multi-tenancy, while seemingly having the advantage of being able to upgrade the data model in one shot for all customers, has many complicating disadvantages in addition to the fact that major re-engineering of legacy applications is needed to move to this model.

9.2 MULTI-SCHEMA APPROACH

Instead of insisting on a single schema, it is sometimes easier to modify even an existing application to use multiple schemas, as are supported by most relational databases. In this model, the application computes which `OU` the logged in user belongs to, and then connects to the appropriate database schema. Such an architecture is shown in Figure 9.3

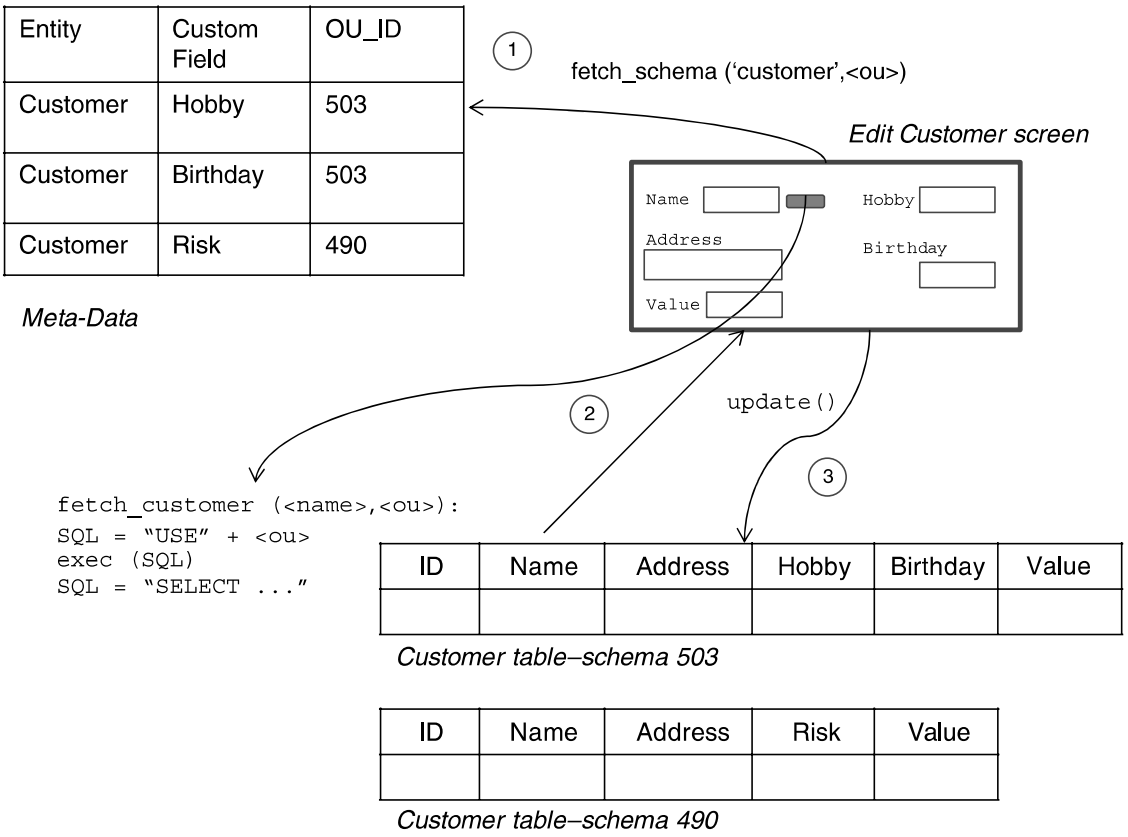


FIGURE 9.3. Multi-tenancy using multiple schemas

In the multiple schema approach a separate database schema is maintained for each customer, so each schema can implement customer-specific customizations directly. Meta-data describing customizations to the core schema is also maintained in a separate table, but unlike the Custom Fields table of Figure 9.2, this is pure meta-data and does not contain field values in individual records. As a result, the application design is simpler, and in case a legacy application needs to be re-engineered for multi-tenancy, it is likely that the modifications will be fewer and easier to accomplish.

Consider implementing the Edit Customer screen as discussed earlier using a multiple schema approach: The application renders the appropriate fields on the screen using information from the Meta-Data table. When making a database query, the application sets the database schema before issuing data manipulation (i.e. SQL) statements so as to access the appropriate schema. Note that supporting the multiple schema model involves incorporating elements of an *interpretive* architecture, very similar to the Dev 2.0 model discussed in Chapter 3, and which we shall return to in more

detail in Chapters 12 and 14. Thus, it is natural that SaaS offerings based on the multiple schema model are quite naturally able to morph into Dev 2.0 platforms.

We have described a rather simple implementation to illustrate the concept of using multiple schemas for multi-tenancy. In practice, web-application servers need to have schema names configured during deployment so that they can maintain database connection pools to each schema. Therefore, another level of indirection is usually required, where customer name (i.e. `OU`) is mapped to the actual schema name, so that customers can be added or deleted online without bringing the system down.

In the case of a multi-entity scenario within a single organization, the number of users was relatively small, probably in the thousands at most. For a SaaS application, the number of users will be orders of magnitude larger. Thus additional factors need to be considered for a multi-tenant SaaS deployment, such as how many applications server and database instances are needed, and how a large set of users are efficiently and dynamically mapped to `OU`s so as to be connected to the appropriate application server and database instance.

9.3 Multi-tenancy using cloud data stores

As discussed in the previous chapter, cloud data stores exhibit non-relational storage models. Furthermore, each of these data stores are built to be multi-tenant from scratch since effectively a single instance of such a large-scale distributed data store caters to multiple applications created by cloud users. For example, each user of the Google App Engine can create a fixed number of applications, and each of these *appears* to have a separate data store; however the underlying distributed infrastructure is the same for *all* users of Google App Engine, as we shall describe in more detail in Chapter 10.

Here we focus on a different problem: As a *user* (application developer) of a cloud platform , how does one create one's own multi-tenant application? In the case of Amazon EC2 the answer is straightforward; since this is an infrastructure cloud it gives users direct access to (virtual) servers where one can recreate exactly the same multi-tenant architectures discussed earlier using standard application servers and database systems.

However the situation is different using a PaaS platform such as Google's App Engine with its Datastore, Amazon's SimpleDB or even Azure's data services. For example, a single App Engine application has *one* data store name

space, or schema (so, if we create one 'Customer' model, then we cannot have another by the same name in the same application). Thus, it appears at first that we are constrained to use the inefficient single schema approach.

However, an interesting feature of the Google Datastore is that entities are essentially *schema-less*. Thus, it is up to the *language* API provided to define how the data store is used. In particular, the Model class in the Python API to App Engine is object-oriented as well as dynamic. As we have seen earlier in Chapter 5, the properties of all entities of a 'kind' are derived from a class-definition inheriting from the Model class. Further, as Python is a completely interpretive language, fresh classes can be defined at runtime, along with their corresponding data store 'kinds.'

Figure 9.4 shows one possible implementation of multi-tenancy using multiple schemas with Google App Engine, in Python. Separate classes are instantiated for each schema, at runtime. This approach is similar to simulating multiple schemas in a relational database by having table names that are schema dependent.

A similar strategy can be used with Amazon's SimpleDB, where *domains*, which play the role of tables in relational parlance and are the equivalent of

```
# Normal schema definition (not used)
#Class Customer(db.Model):
#  ID   = db.IntegerProperty()
#  Name = db.StringProperty()
  ... ...
# Dynamic OU specific classes for 'Customer'
for OU in OUList:
  #Gets ALL fields from meta-data
  schema=fetch_schema('Customer' OU)
  # Create OU specific class at run-time
  OUclass=type('Customer'+OU,(db.Model,), schema)
```

ID	*Name*	*Address*	*Hobby*	*Birthday*	*Value*

Customer 503

ID	Name	Address	Risk	Value

Customer 490

FIGURE 9.4. Multi-tenancy using Google Datastore

'kind' in the Google Datastore, can be created dynamically from any of the provided language APIs.

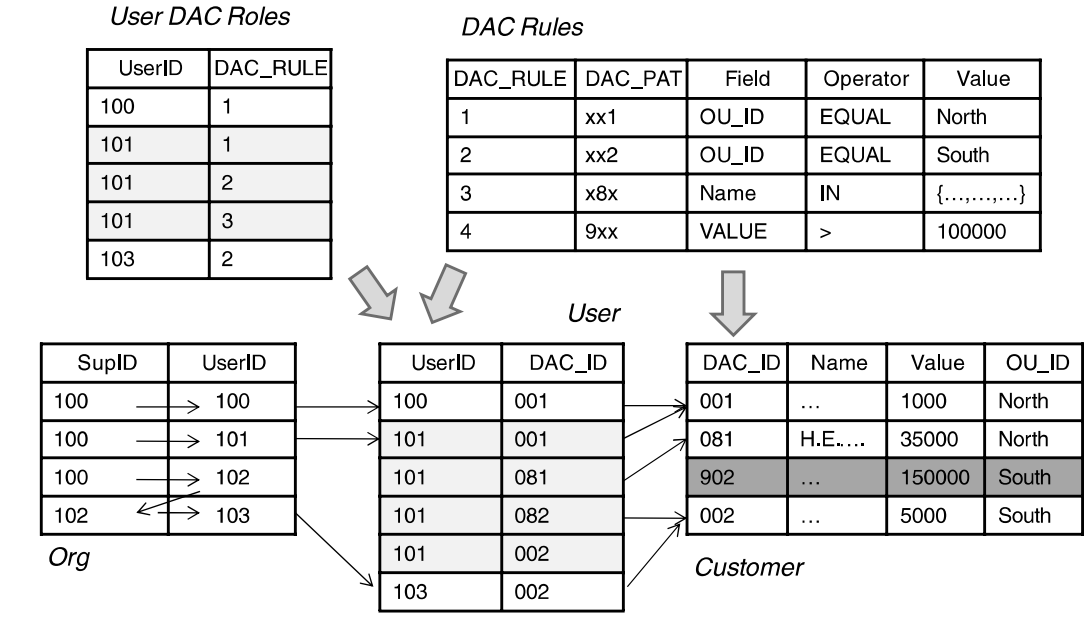
9.4 Data access control for enterprise applications

So far we have covered the typical strategies used to achieve multi-tenancy from the perspective of enabling a single application code base, running in a single instance, to work with data of multiple customers, thereby bringing down costs of management across a potentially large number of customers.

For the most part, multi-tenancy as discussed above appears to be of use primarily in a software as a service model. There are also certain cases where multi-tenancy can be useful within the enterprise as well. We have already seen that supporting multiple entities, such as bank branches, is essentially a multi-tenancy requirement. Similar needs can arise if a workgroup level application needs to be rolled out to many independent teams, who usually do not need to share data. Customizations of the application schema may also be needed in such scenarios, to support variations in business processes. Similar requirements also arise in supporting multiple *legal* entities each of which could be operating in different regulatory environments.

As we mentioned earlier, in a multi-entity scenario a subset of users may need to be given access to data from all branches, or a subset of branches, depending on their position in an organizational unit hierarchy. More generally, access to data may need to be controlled based on the *values* of any field of a table, such as high-value transactions being visible only to some users, or special customer names being invisible without explicit permission. Such requirements are referred to as *data access control* needs, which while common, are less often handled in a reusable and generic manner. Data access control (or DAC) is a generalization of multi-tenancy in that the latter can often be implemented using DAC. In Figure 9.5 we illustrate how data access control can be implemented in a generic manner within a single schema to support fairly general rules for controlling access to records based on *field values*.

Each application table, such as Customer, is augmented with an additional field `DAC_ID`. The DAC Rules table lists patterns based on value ranges of arbitrary fields using which the values of the `DAC_ID` in each Customer record are filled through a batch process. Users are assigned privileges to access records satisfying one or more such DAC rules as specified in the User DAC Roles table. This information is expanded, via a batch process, to data



Using RDBMS

```
SELECT … FROM CUSTOMER AS C, USER AS U, ORG AS O1 OUTER JOIN ORG AS O2 WHERE
  O1.SupID = :user AND                    /*  current user
  O1.UserID = O2.SupID AND                /*  current user and supervisors
  O2.UserID=   U.UserID AND               /*  DAC_IDs of these users
  U.DAC_ID = C.DAC_ID                     /*  filter Customer table by these DAC_IDs
```

Using GQL queries on Google Datastore:

```
USER_LIST = fetch_team_for_user(:user)    /* execute self join on ORG in memory
DAC_LIST  = SELECT  DAC_ID  from USER WHERE UserID in USER_LIST
SELECT … FROM  CUSTOMER  WHERE  DAC_ID  IN DAC_LIST
```

FIGURE 9.5. Data access control

in the User table where there is a record for each value of DAC_ID that a user can access. For example, the user 101 has access to three DAC rules, which translate to five records in the User table. This calculation involves computing the complete set of mutually exclusive and unique DAC range combinations based on the DAC Rules and thereafter which subset of these a particular user has access to based on the User DAC Roles information; note that this computation is independent of the actual DAC_ID values in the Customer or other application tables.

It is straightforward to limit access to records of the Customer table to only those a particular user is permitted, as specified in the User table using a join. In the illustration of Figure 9.5, we introduce an additional complication where users are also given access to the DAC permissions of all their direct reports, as specified in the Org table.

In a traditional relational database, SQL queries on the Customer database can be modified to support data access control by introducing a generic join,

including a self-join on the Org table to find all direct reports of a user, which is then joined to the User table and the Customer table. However, in a cloud database, such as Google Datastore or Amazon's SimpleDB, *joins are not supported*. Therefore the same functionality must be implemented in code as shown in the figure: The self-join on Org is done in memory giving a list of reportees, including the user; this is used as a filter to get the permissible `DAC_ID`s from the User table. Finally this list is used to filter the application query on the Customer table.

It is important to note that when adding or modifying Customer records the `DAC_ID` needs to be recomputed based on the DAC Rules; this computation also needs to be optimized, especially if there are a large number of DAC Rules. Adding new DAC Rules or modifying existing ones will also require re-computation and updates to the `DAC_ID` values in the Customer table. Care also needs to be taken when filling the DAC Rules table to ensure that DAC ranges on the same field are always non-overlapping.

We thought it fit to cover data access control here, as part of our treatment of multi-tenancy, first because these requirements are closely related, but also to bring out the complexities of real enterprise applications even for incorporating a generic requirement such as data access control. Beyond the example itself the lesson to be learnt is that migrating applications to a multi-tenancy model, especially using cloud databases, is not a trivial task.