UCS15E08 – CLOUD COMPUTING

UNIT 2

Study of Hypervisors

A hypervisor is an operating system, which means that it knows how to act as a traffic cop to make things happen in an orderly manner. The hypervisor sits at the lowest levels of the hardware environment.

Because in cloud computing you need to support many different operating environments, the hypervisor becomes an ideal delivery mechanism. The hypervisor lets you show the same application on lots of systems without having to physically copy that application onto each system. One twist: Because of the hypervisor architecture, it can load any (or many) different operating system as though it were just another application. Therefore, the hypervisor is a very practical way of getting things virtualized quickly and efficiently.

Scheduling access

You should understand the nature of the hypervisor. It's designed like a mainframe OS rather than like the Windows operating system. The hypervisor therefore schedules the amount of access that guest OS have to everything from the CPU; to memory; to disk I/O; and to any other I/O mechanisms. With virtualization technology, you can set up the hypervisor to split the physical computer's resources. Resources can be split 50–50 or 80–20 between two guest OS, for example. Without the hypervisor, you simply can't do that with Windows.

The beauty of this arrangement is that the hypervisor does all the heavy lifting. The guest operating system doesn't care (or have any idea) that it's running in a virtual partition; it thinks that it has a computer all to itself.

Types of Hypervisiors

Different hypervisors support different aspects of the cloud. Hypervisors come in several types:

1. **Native hypervisors**, which sit directly on the hardware platform are most likely used to gain better performance for individual users.
2. **Embedded hypervisors** are integrated into a processor on a separate chip. Using this type of hypervisor is how a service provider gains performance improvements.

3. **Hosted hypervisors** run as a distinct software layer above both the hardware and the OS. This type of hypervisor is useful both in private and public clouds to gain performance improvements.

WEB SERVICES: SOAP AND REST

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.

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. 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.

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 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. 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.

3. 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. 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.

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. In our opinion 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.

	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	No special tools required especially if using JSON	REST is lighter and easier to use

	client and server development		
Productivity	Low, due to complex tools and skills needed	High, due to simplicity	REST is faster an cheaper for developers to use

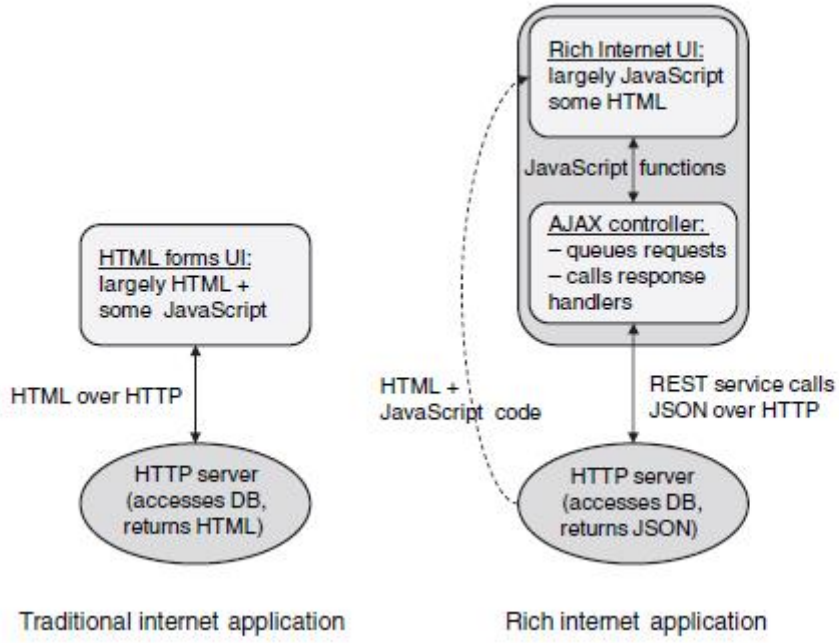
AJAX: ASYNCHRONOUS 'RICH' INTERFACES

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**.**

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

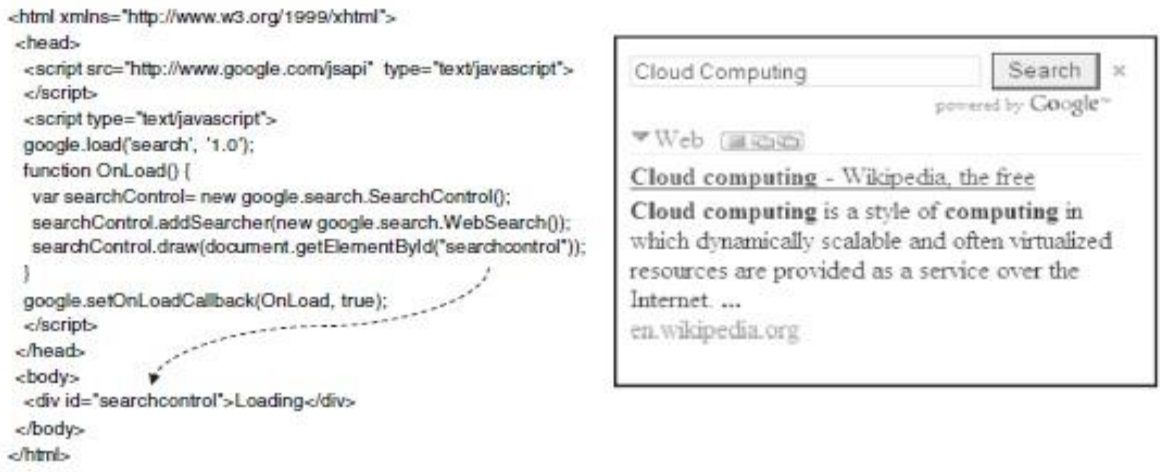
Incomparison to traditional web applications, base HTML page is loaded along with JavaScript code that contains the remainder of the user interface. This JavaScript program renders a 'rich' user interface that can often look like a traditional client-server application. When data is required from the server, asynchronous requests are made via REST 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.

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.



The above figure 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. User code calls these methods to instantiate a search control 'object' and render it within the HTML page dynamically after the page has loaded.

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.

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

Virtualization technology

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.

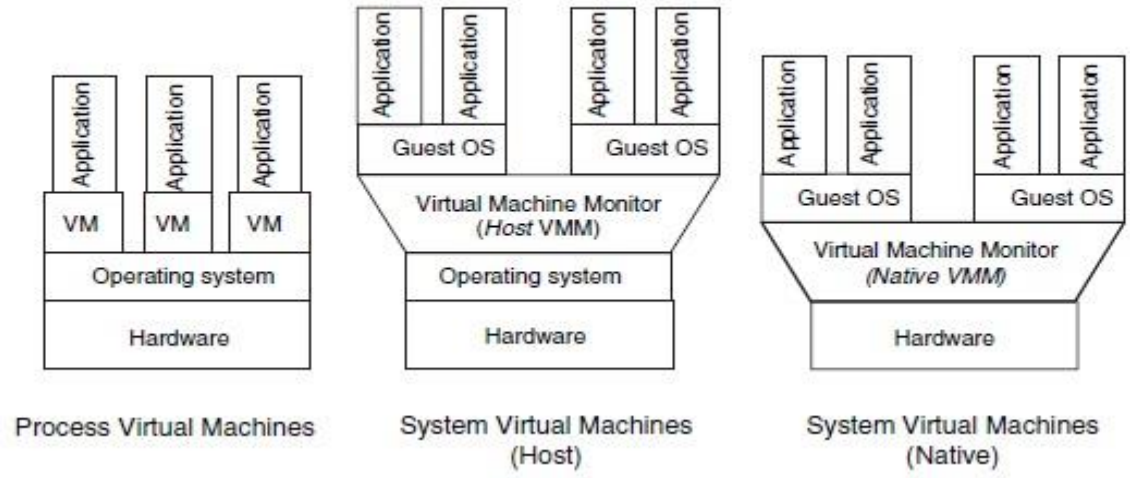
1. Virtual Machine technology

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.

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.



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.

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

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.

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.

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.

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.

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.

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.

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.

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.

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 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.

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.

1. 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.

2. The second approach is called 'application streaming.' Instead of running applications on central virtual machines, application streaming envisages maintaining only virtual machine images centrally

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.

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:

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.

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.

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.

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.

PITFALLS OF VIRTUALIZATION

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 resourcesaremadeavailabletoeachvirtualserverinasynchronizedmanner. 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.

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.

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.

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.

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.

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 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.

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.

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. 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 OUs so as to be connected to the appropriate application server and database instance.

3. MULTI-TENANCY USING CLOUD DATA STORES

Cloud data stores exhibit non-relational storage models. Furthermore, each of these data stores are built to be multitenant 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.

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. 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.

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

4. DATA ACCESS CONTROL FOR ENTERPRISE APPLICATIONS

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.

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 Data store or Amazon's Simple DB, joins are not supported.