Network Virtualization Architecture. Concepts

Purpose is to define an Architecture, not to define or bless an implementation of that architecture.

The “architecture” we are addressing in this document is a technical and theoretical treatment of network (and cyber-infrastrcutre generally) design and function. This is \*NOT\* a business plan or a business architecture.

The document will address “services” in terms of how virtualized capabilities are implemented within this virtualized architecture, but this document is not a business case, or value proposition, for particular services. The document will try to describe how virtualization can deliver certain types of services with greater flexibility, agility, responsiveness. We will touch on opex cost efficiencies of virtualized facilities. But these are primarily meant as illustrative examples.

 The Network Virtualization Architecture document does not address specific implementations – how one service provider might implement these principles in their existing facilities nor does it explain how this architecture integrates with existing operational practice. These will likely be forthcoming, but as separate documents.

It should be noted that many existing business or technical paradigms and service models change substantially with the introduction of virtualization. This document does not address how providers should roll out or deploy such a capability. Much of this may be obvious from the discussion in this doc, but it should be understood that deployment of a conforming architecture does not have existing best practice, accepted engineering design rules, effective monitoring or valid capacity planning approaches at this time. And legacy rules and best practice rarely apply in the virtualized architecture. And so, short of involving that small community of software or systems engineers that have advanced and deep experience with the virtualization architecture and components, that there is no established market place or skills sets available to just hire to implement this environment. In a rare recommendation, this document recommends that service providers begin virtualization pilots immediately and aggressively – there is a very long and steep learning curve and organizations will need to develop the staff in-house through hands-on experimental alpha and beta pilots to develop the needed skills sets. *Any starting point will be productive* if it requires the staff to work through the relevant issues – and there will be many issues. Develop the collaborative relationships that will promote communication among those few experts to assist in early experimental deployments. And there will be resistance to throwing out old “conventional wisdoms” that are not applicable to virtualized services in order to properly implement new concepts required in a virtualized cyber-infrastructure universe. So virtualization requires both a new mind set \*and\* new technical skills.

This document does try to be clear and start top-down with high level concepts and use those to introduce more detailed or technical discussions and principles. But this is not a primer. The authors invite you to join the working group and ask questions and discuss the issues, and to help devise solutions to aspects not well worked out.

Terms and Definitions:

**What is “virtualization”?**

1. Virtualization is the *abstraction* of a [network] function or service component in order to define the behavior of that function or component independently of any underlying implementation.
2. Virtualization then *maps* the abstracted service object to a suitable set of infrastructure that, when configured or provisioned appropriately, will exhibit the specified behavior.

We include the mapping process as part of the virtualization architecture, and the realization of virtual objects, because these are crucial aspects of the “virtualization” process. It can be argued that just abstracting the service objects does not deliver something that is “virtually” the same as the thing upon which the abstraction is modeled. That is, abstracting the qualities of a virtual machine or a firewall function is not very useful if there is no way to materialize that abstracted object to actually deliver a real world service. Virtualization architecture therefore includes the mapping and realization in terms of concept, even if it does not require or go into details about specific implementation aspects of any particular abstracted objects.

“Virtualization” lifts a function or service out of the physical plane and into the “*concept plane”* where virtual objects can be manipulated without regard to their ultimate physical implementation and using graphical computer aided design tools to construct sophisticated new custom services. Automated software agents can then map those newly composed virtualized services to appropriate infrastructure.

**Why a Virtualization Architecture?**

From Merriam-Webster: **architecture** (noun)

1. the art or science of building
2. **a:**formation or construction resulting from or as if from a conscious act

**b:**a unifying or coherent form or structure

1. [architectural](https://www.merriam-webster.com/dictionary/architectural) product or work
2. a method or style of building
3. the manner in which the components of a computer or computer system are organized and integrated

(I especially like 2a: “as if from a conscious act.”)

An “*architecture*” describes how things function and work together.

We all have a hint of what is meant by a *virtual* thing. We hear of virtual machines, virtual circuits, virtual storage, virtual private networks, virtual Local area networks, virtual switching, virtual routing and forwarding, virtual memory, virtual addressing. But these are difficult to discern any common thread that promotes these working together.

Merriam-Webster is NO help: **virtual** (adjective)

1. being such in essence or effect though not formally recognized or admitted
2. being on or simulated on a computer or computer network
3. **a)** occurring or existing primarily online

**b)**of, relating to, or existing within a virtual reality

1. of, relating to, or using virtual memory
2. of, relating to, or being a hypothetical particle whose existence is inferred from indirect evidence

(M-W using circular definitions…ugh!)

Technopedia.com lists 130 technical terms that all begin with the word “virtual”.

This adjective usage of the term virtual implies that there is some common feature all of these things share…since they are all *virtual*… right? At best, the technopedia terms use the term “Virtual” simply to indicate a software implementation of something that used to be based upon some hardware product or device. It does not provide any insight into shared characteristics of “virtual” things, or how different “virtual” things work together, or a shared or common approach for manipulating or managing such things to create a system of virtual things that are working together.

We need to define the term “virtual” to represent a common set of characteristics for anything called virtual. We need to define an “architecture” that describes how virtual things function and work together.

# We need “*virtual*” to mean something specific: conforming to an architecture that employs a common concept driven design methodology, and scalable technology agnostic implementation principles to deliver custom and highly agile cyber-infrastructure applications and services.

#

**What is a Virtualization Architecture**

A “*virtualization* *architecture*” describes how virtual things are defined and function and work together to build custom application specific services.

A Virtualization Architecture does more than simply abstract functions or components. An architecture should:

1. Define the basic components of the architecture – the virtual objects.
2. Define how virtual objects are manipulated - created, destroyed, interrogated, etc. i.e. the virtual object life cycle.
3. How virtual objects express external behavior – How they interact with other entities.
4. Define the common principles by which these “virtual” objects are realized to deliver their abstracted behavior.

These key aspects define what virtual objects are, how we create or release them, and how they interact with each other.

**[VIMs vs VNFs as in ETSI NFV]**

**Key Concepts**

**Common Object Model**

The Common Object Model describes those aspects of virtual objects that are shared among all virtual objects…these are the common aspects of the model we use to define virtual objects.

1. Object Instance Identifier (OIID) – the Name of an object e.g. vm01-Houston, or computeNode02, etc.
2. The Class name – the type of object it is, e.g. ”ELine” := ethernet private line virtual circuit, “BareMetalServer” := Bare metal server, …
3. Attributes 1..n. – characteristics that define or reflect the state of the object instance. E.g. iso\_image= <filename>, memSize=64GB, Bandwidth=50Gbps, RecvErr=? Drops=? Set the image, memory size, or bandwidth; or query the receive errors, or transmit queue buffer congestion.
4. External ports 1..n – Interfaces to allow data to enter or exit the virtual object.
5. Description – what this object does.
6. Sub-Objects- Does this object incorporate other virtual objects?
7. Interconnection scheme – Which objects interact with which other objects, internally or externally.

**Common Life Cycle API**

These are basic functions that transition any virtual object through its life cycle.

1. Define(Class) – defines a new object class
2. Reserve(Class, OIID) - maps abstract functionality of the object to elements of the infrastructure, and holds those components for that object instance. Returns the provider’s resource Object Instance ID for the object.
3. Activate(OIID) – configure the resource and bring it into service to the user
4. Deactivate(OIID) – deconfigure the resource
5. Release(OIID) – Release reserved infrastructure components associated with an object
6. Query(OIID) – returns the state of an object instance. This returns all or specific attribute/value pairs for the instance
7. Undefine(Class) – deletes a class template.

The abstraction of a network function or component is captured in a textual specification that defines a “*Class*”, or “*Type*”, of virtual object. The Class specification defines an object in terms of its internal function and any attributes that may modify or reflect that function for a particular object instance of the class.

A “Class” defines a type of object whose instances all have a shared set of attributes. For instance, a “Host” Class might define an x86-based computenode, with attributes of BaseClockRate, MainMem, MinCache, ISOimage, NumCores. And a “Link” Class may define a point to point connection with attributes of Capacity, MaxFrameErrorRate, MaxJitter.

We define a Class and associate two things with that name: a) a set of relevant attributes that define and bound such objects, and b) a set of software functions that can take those attributes and map them into infrastructure to produce an instance of that class with the expected behavior.

One can think of a Class as a template that describes how a type of thing behaves and the knobs (attributes) that the user can set to tune specific instances of that class to their needs.

A virtual object of a particular Class is *instantiated* when a user requests that an instance of a class be reserved. The attributes of that class have been set to specific values by the user to define a specific *instance* of the Class. The description of this instance is sent to the provider. An Object Instance Identifier (OIID) is generated when the request is issued from user to provider. This is when the object instance begins its existence.

(Note: The user could create an OIID for the request also since, as far as the user is concerned, the instance began its existence when the local [user] agent decided to create an instance and began fixing its attributes. Formally, in a symmetric protocol both parties pass their respective OIIDs to the other as part of the creation process. This allows the two parties to always reference the other party’s OIID when sending a message to the other party about a particular object instance. This “link local” type of OIID convention, by associating the local OIID with the remote OIID, prevents confusion should an OIID ever be duplicated – its association with a particular party (or domain) keeps it globally unique from the local agent’s perspective. This can also simplify internal local processes that need only deal with locally generated OIIDs. )

A virtual object instance is also called a *resource*.

The underlying facilities in which or to which a resource is mapped is called the *infrastructure*.

A virtual resource instance is *mapped* when the functional aspects of the instance have been assigned to appropriate components in the infrastructure that can produce the behavior for the instance. Those infrastructure components must then be held (reserved) for that resource instance. After successfully reserving the infrastructure components assigned to the resource instance, the resource is in a Reserved State. Note1: There may be many (!) possible valid mappings for a given object instance. A valid mapping (which may require a specific set of interdependent infrastructure components – not just one) is a mapping that will produce an object instance with the user’s requested performance attributes/bounds. The provider may then choose any valid mapping to actually reserve for the instance. The provider’s selection of a mapping may reflect some further set of provider specified attributes that will reduce the valid mappings further – perhaps to implement some optimization strategy or that may reflect the provider’s need to efficiently manage shared facilities across a large user base. Once the provider has applied all of provider’s constraints to the valid user mappings, then any remaining mappings can be selected and reserved. If selected mapping cannot be fully reserved, then it is released and the next random mapping is reserved, and so on. If no valid mapping can be successfully and completely reserved, then the Reserve() primitive must return a ReserveFailed status. And the provider instance is Released and logged. Note2: The architecture does not specify how the valid mappings are constructed or represented in any implementation, nor how comprehensive those mappings may given a domain infrastructure. The architecture simply describes the notions of valid user constraints, an intersection with provider constraints to identify valid mappings.

A virtual resource is *realized* when the infrastructure reserved for that object instance has been configured or otherwise provisioned to produce the functional behavior of the resource. The virtual object enters the Active state and is available to the user.

Composite virtual objects

A “composite” virtual object is simply a named grouping of one or more virtual objects. The composite description (Class description) also describes the group’s data flow interconnectivity – the topology. The objects in the group are called the “children” of the composite, the composite itself is the parent. A composite is realized by realizing its children.

Atomic virtual objects

An atomic virtual object is an object that is directly mapped into the infrastructure. Atomics have no internal children objects, they can only be realized by directly provisioning and configuration of the infrastructure to realize the object instance. Atomic classes must provide the software modules that do the infrastructure mapping and infrastructure reservation, and the modules that do the infrastructure configuration and/or provisioning.

****

Note: terminology “instantiation” (concept), “mapping” (concept & infrastructure), “realization” (infrastructure), “provisioning” (infrastructure).

**Insulation and Isolation**

Two of the most important principles of virtualization are closely related: insulation and isolation. First, we want all object instances to be protected from one another. This means an object is not affected by events or actions occurring outside of the object. This is *insulation*. Second, we do not want events or actions inside a virtual object to affect objects outside of that instance. This is *isolation*. These properties as stated are relative to a particular object instance. However, when we apply these properties universally across all object instances in a domain we find that insulation and isolation are in fact the same property. So we just refer to it as “isolation”, or *the Vegas Rule* – “what happens in a virtual object, stays in that virtual object.” The Vegas rule is one of the most important principles in virtualization and should be rigorously implemented for all object classes.

Important object characteristics such as predictability and repeatability require proper isolation among object instances. The infrastructure represents a finite quantity – it is not unlimited. In order to deliver predictable performance to all virtual objects in an infrastructure domain, each individual virtual object must be mapped so as to insure that sufficient infrastructure is available and allocated to the object instance to deliver the requested performance for the instance. Also, the performance of the resulting instance must not be able to exceed the performance limit assigned to that instance. The instance must be able to perform up to the bounded limit, and no further. This is a well bounded object – i.e. their range of functioning is has a fixed limit and it is enforced so that the object can consume all of the capabilities within its boundaries, but no more.

There are nuances to provisioning and enforcing performance limits (insulation and isolation) in different types of infrastructure. However, since the architecture does not dictate the infrastructure or specific implementations of a virtual class, this document will not explore these issues further. The architecture simply requires that virtual objects be rigorously isolated in order to provide performance assurance for any object that requests or requires such predictability.

**How virtual objects interact with one another**

Virtual objects are “well bounded”, they are isolated. No information is able to ingress or egress the virtual object. This is not limited to conventional data, but includes any actions that may create interference with other objects – for example a well bounded VM cannot exceed the number of vCPUs it is assigned. To do so would interfere other VMs by starving them of cycles and so would be communicating beyond its bounds. Similarly, a virtual switch should not be able to see packets or frames that belong to other virtual objects or environments. A well bounded object allows no outside object to perceive its presence beyond the defined characteristics that were configured as its boundaries. Thus a well bounded object has its boundaries explicitly defined and rigorously enforced to prevent internal actions from exceeding those bounds and interfering with other entities. From an outside observer, a virtual object is an opaque black box that offers no internal signs of life. And from an internal observer, the outside world is likewise an opaque black universe that is completely impenetrable.

This definition allows a virtual object to function within its virtual bubble, but it is completely unable to communicate with any external entity, or vice versa. In the real world, this is a problem. The impenetrable boundary means that nothing performed inside the bubble can be expressed outside – ever. And this also means that a user has no means to reach into the bubble to control or monitor the activities inside the virtual object. We need a means of allowing information to flow into or out of a virtual object such that we can explicitly characterize the flow and we can be assured it will not impact other virtual objects. Put more simply, the ingress or egress of information to the bubble must itself be well bounded so that we do not break the “well bounded” predictable and deterministic principles of virtual objects.

To do this, we define “ports” on a virtual object. Ports are interfaces on the boundary that allow information to flow into or out of an object. These ports are well bounded also. By default, virtual ports are explicitly defined as to their data flow characteristics – the format of the data ingressing or egressing, the amount of data that can transit a port in a given period of time, and some measure of characteristic of the flow – its burstiness or shape. A virtual port is not a transport link – it is not a circuit or LSP or fiber… It can be thought of as a hole in the wall of the well-bounded bubble that is the virtual object.

So virtual objects communicate by transmitting or receiving information via virtual ports. But since the virtual ports are simply ingress and egress “holes” in the virtual object’s boundaries, we still require a means of indicating where the data transiting one of these ports should be going. The data must flow into or out of another virtual port on another virtual object. Thus the virtualization architecture requires a means of indicating which virtual ports on which virtual objects are connected to, or are *adjacent* to, some other virtual port on some specified virtual object. Data exiting one hole in a bubble must immediately enter another hole in a bubble. These port-to-port mappings are called “adjacencies” and consist of pairs of 2-tuples: Virtual object A/virtual port N “is adjacent to” Virtual object Z/virtual port M.

Thus, the virtualization architecture must be able to specify virtual ports as part of a virtual object instance, and it must be able to indicate which virtual ports are connected to, or are “adjacent to”, which other virtual ports. These adjacency relations are not virtual objects – they are *pure relationships* and have no physical analog.

Groups, or composite virtual objects

By defining multiple virtual object instances, each with a set of virtual ports, we create a grouping called a *composite* virtual object. And by specifying the adjacency relations among the virtual ports within the composite group, we define the topology of that grouping – the data flow graph. If the virtual objects are infrastructure objects (e.g. VMs, or virtual circuits) the adjacencies are typically said to form the virtual network topology. If those objects in the group are a more abstract service or function, the adjacencies are said to form a *functional service graph*, or a *service chain*. There is no actual difference between these graphs or how they are managed.

Finally, a composite virtual object inherits the characteristic of its constituent virtual objects and so the composite is itself a well bounded object. And it too can fall victim to the same problem as the atomic virtual object of total black box behavior to external entities. So the composite virtual object needs a means of ingressing or egressing information as well. Like other virtual objects, a composite object may define external ports that allow information in or out. Externally, these virtual ports on the composite object look and act just like the ports described above.

However, internally things are a bit different. Internally, there is a sense of the well bounded bubble and everything outside the bubble is nothing but the dark universe of virtual space. If from the internal perspective we were to morph the external boundary, to invert it to show the internal components as if they were outside and the dark universe of virtual space now inside the boundary, then the boundary would appear as just another opaque virtual object. And we can manipulate it similarly by defining ports and interact with that external domain object just like any other virtual objects in the composite. And we can define internal adjacencies that link internal virtual object ports to the external virtual object ports allowing information to ingress and egress the composite object using the same derived resource graphing technique used for the internal topology. Indeed, the composite object’s external boundary is defined by the emergent bounding of the internal virtual instances. The external ports inherit their boundary constraints from the internal ports they are associated with (adjacent to).

It should be obvious, but we note it here anyway, that a composite virtual object may contain other composite virtual objects as its children. Thus we can view virtual objects as hierarchical tree structures – the top most composite being the root virtual environment containing other objects – which may be composites as well, or that may be atomic virtual objects, or a mix of both. This ability to use composites to recursively compose more complex or sophisticated virtual objects or virtual services, is a very powerful feature of a well thought out virtual architecture. This allows complex service constructs to be saved as virtual object templates in a library or portfolio, and those virtual object templates then used in other more complex services or environments.

These construction rules allow virtual composite objects to exhibit the same well bounded predictable and deterministic performance characteristics that accrue from the well bounded properties of the internal virtual objects. Thus these architectural principles allow us to construct recursively larger and more sophisticated composites that have the same predictable deterministic performance.

**How virtual objects are realized**

In general, the virtualization architecture does not dictate how virtual objects are actually realized in the infrastructure. *How* is left to a local service provider. The service provider’s only and overarching responsibility is to provide a virtual object that meets the specific bounds of the user request. Some of those boundaries are innate in the description of the virtual object class – e.g. an object that behaves like an x86 architecture computer. And others are user specified attribute values or defaults for the object class. If we take the example of a class of objects that are hardware analogs, that is virtual objects that are modeled after hardware devices like the x86 virtual machine or a virtual openflow switch, then the provider’s job is to deliver an object whose behavior is indistinguishable from the original object – it is virtually the same thing.

The virtualization architecture is technology agnostic. This means that the architecture does not care what type of technology is used to produce the virtual object, in fact there may be multiple underlying technologies used even by a single provider. For instance, the provider can implement an x86 VM using real x86 hardware running a hypervisor, or they may elect to implement the VM using a software x86 emulator that behaves like an x86 device. As long as the user cannot distinguish between the two implementations, and as long as a VM implemented with one or the other meets the specification requested by the user, then they are valid and conforming implementations in the eyes of the virtualization architecture.

Technology agnostic also underscores the notion that a virtual object can be defined to be anything. There is no required implementation, or rather, the provider can implement the virtual class however they see fit as long as the class specifications are met. Therefore, there are no constraints imposed by the infrastructure on what a class may be defined to be or do.

**Deterministic performance, predictability, and repeatability**

Deterministic performance is a key property of the generic virtualization architecture. Deterministic means that given an effective isolation implementation (well bounded objects), and the same initial conditions, then a virtual object will behave exactly the same each time it is realized. Predictability and repeatability are thus possible.

However, in the real world, complete isolation and complete control of all initial conditions is very difficult to achieve. A realistic virtualization model is formally quasi-deterministic, where performance variance from one case to the next may vary, but such variance is vanishingly small hopefully to be unnoticed, but below some quantifiable threshold. But if the virtual implementation is done rigorously to provide a high degree of isolation (tight bounding), and the initial conditions are similarly tightly controlled each time, then the variance in the resulting performance of two separate realizations will be predictable and minimal. The smaller the resulting variance, the more deterministic and repeatable is the resulting behavior. Even in composite virtual objects, deterministic mapping of each constituent virtual instance means the performance of combinations of constituent instances can be inferred and is predictable within some range defined by the combination of the deterministic variability of each component instance. Thus a composite of any arbitrary size or data flow topology can, in theory, exhibit a degree of predictability as an emergent property of its constituent components.

Deterministic variance is a vector composed of all attributes as implemented vs the ideal as requested. For example, a VM requiring a 2.3 Ghz base clock, that is mapped to a 2.4 Ghz cpu, will exhibit a deterministic variance of 2.4/2.3 or 104.3% of what was specified. Such a minor difference may not translate into a major predictability delta if, for example the VM is not consuming 100% of available CPU cycles. Alternatively, it may be a CPU bound application and that delta could quickly accumulate to a significant performance improvement over the nominal expectation. From one instantiation to another, a VM instant that may have the requested 2.3 Ghz clock will not exhibit the exact repeatable performance of the 2.4 Ghz allocated instance. Even where the CPU is not saturated, the faster clock can result in faster responses to other components, which could potentially cause unexpected behavior in those streams. Over longer and longer periods of time the slightly faster (or slower) CPU can result in a wide divergence in repeatability. In reality, deterministic variance will occur for any object instance as the bounds and initial conditions cannot be perfectly controlled. And over time the deterministic variance will cause performance divergence from one experiment (or run) to the next. Characterizing the effect of multi-variable deterministic variance due to mapping on the predictability and repeatability of virtual object instances – especially for composites - is a topic for research. This topic would also consider the effect of data flow topology and application specific factors such as initial conditions, load clipping due to bounding, and hold times.

For purposes of specifying a deterministic virtualization architecture, we will assert that for purposes of bounding the virtual object, i.e. mapping instances to infrastructure, that the deterministic variance is negligible, and that the implementation of virtual objects does produce deterministic performance.

Realization

Virtual objects are realized in a two step process. First, the infrastructure necessary to produce the virtual instance must be identified, located, and held for the virtual object. This includes a start time and an end time for each piece of infrastructure required. This is the *reservation*. Second, the infrastructure components reserved for the object must be configured or provisioned to actually produce the behavior expected of the virtual object. This is *activation*. Once the object instance has been activated, it is said to have been *realized*.

When an object is requested to be reserved, a unique resource instance id is generated and assigned to represent the instance. This is called *instantiation.* Even if the instance does not successfully reserve all of the needed infrastructure components, it must be instantiated in some form to even search for those components. This identification of necessary infrastructure components necessary to implement the object is called *mapping* the object to the infrastructure. Mapping of two exactly duplicate user object requests may result in two very different infrastructure mappings depending on the implementation chosen to implement each. Indeed, a single object instance may generate multiple valid mappings that could produce the desired object. The provider chooses one such mappings as the candidate for reservation. The mapping of the components and the reserving of components may be performed in a dovetailed fashion depending on the mapping chosen and the infrastructure availability. So these are not prescribed by the architecture to happen in a particular order.

If the provider is able to find and hold all of the necessary infrastructure components for the time requested by the user, then the provider sends a ReserveConfirm response back to the user. If the provider tries but is unable to completely reserve any valid mapping, then the components that were successfully reserved are released and the reserveFailed is returned to the user. The instance (OIID) is Released and the OIID is not logged. The architecture recommends that the instance id be used only once so that such events can be logged and retained indefinitely and so that queries against such logs or the rdb can identify specific objects that were processed.

Object Instance Identifiers

Object instances identifiers should be globally unique and non-repeating. This can be accomplished by providers creating instance ids that are two-tuples: a globally unique network or domain id, and a locally unique instance id (locally unique means within the scope of the service provider’s domain.)

The globally unique component is the Domain Identifier assigned to the service provider. We refer the reader to the OGF NSI v2 protocol specifications for information regarding globally unique domain IDs. For purposes of the Virtualization Architecture, we assert that each provider has been assigned a globally unique network or domain ID. And that provider generates locally unique identifiers for all of it internally unique naming requirements – including for virtual object instances.

**Why Virtualization?**

Virtualization allows us to define our service objects as we wish. Thus a community of users or providers can define the types of objects they want or need to be offered as service objects, develop a consensus portfolio, and then each provider can implement those abstract virtualized objects in their preferred infrastructure. We are no longer simply promoting <a technology or product>as-a-Service.

**Virtualization changes the traditional [network] service model.**

In the legacy model the service provider designs the service and deploys the hardware that presents the service to the user, e.g. an IP service. This is one service engineered across hardware dedicated to that one service. And users connect to and adapt to that service.

In the new [virtual] services model the customer designs their own virtual service environments and deploys them using automated software agents over the provider’s physical infrastructure. The provider delivers virtual service environments to their customers, and the customers deliver services to their users.

the provider delivers virtual service environments to the user.

, and the service provider delivers that custom service environment to the user.

In the old model, the provider ran a service that they build by hand. In the new model, the provider delivers virtual environments – and those virtual environments

Virtualization enables malleable customized services. Because of the common archsoftware driven mapping, these services can be manipAs CSPs do, the overhead of building a service once done manually is amortized across many users

(We need to elaborate here why a technical specification in an abstracted service object does not violate virtualization. For instance, an Ethernet Framed P2P Connection…. Why specifying “Ethernet” as the ingress and egress framing for a virtual Class of connection object does not violate virtualization? ) (… or does it?..)

(Consider how the JAVA Virtual Machine abstracted the “hardware” component of web based applications and provided portability)

**A Network Virtualization Architecture:**

A “network virtualization architecture” describes a set of abstracted service objects that together provide a complete set of virtual components that could function as a network, This is a weak definition in that the term “network” is not well defined.

**“Well bounded” virtual objects**

 “Well bounded” objects are objects that are insulated and isolated from other virtual objects. These conform to the “Vegas Rule”: What happens in a virtual object, stays in that virtual object. This means that a virtual object cannot interact with any other virtual object except through explicitly specified data flow interactions. This means that other virtual objects should never be able to determine if that

Thus a redesigned Virtual CI architecture is comprised of a set of virtual object “*types*” – or *classes* – that can be instantiated to cover

Notions of network automation and orchestration are dependent on a well rounded virtualization model.

**Virtualization can be applied to any cyber-infrastructure function or component.**

Any conventional notion of a CI device, application, function, or service can be abstracted away from the physical infrastructure and defined as an abstracted “*virtualized*” service object. And that abstracted object can then be instantiated and mapped to available infrastructure in a more flexible agile manner and in a more consistent manner that scales to manage these resources efficiently and effectively – and with little or no human intervention required.

SCRAP - SCRAP - SCRAP - SCRAP - SCRAP - SCRAP - SCRAP - SCRAP - SCRAP - SCRAP - SCRAP –

Interestingly, this performance assurance of virtual objects can allow for objects that have no performance guaranteed. That is, one could say a virtual circuit with no performance guarantee should not get any performance. But this is not quite correct. Such a virtual circuit would not have a performance *guarantee.* But it would be allowed to wait and to scavenge capacity if it is available. It would be a “best effort” circuit – it gets what is available. But this begs the question of “what is available”?. The first obvious limit on a best effort circuit would be the physical limit of the port or link it transits. The circuit cannot exceed that physical limit. There could be other limits as well. For example we could provision all best effort circuits within one guaranteed performance circuit. This outer circuit (the “tunnel” circuit) would have a guaranteed maximum capacity and all of the internal best effort circuits would fight each other for capacity within the limits of the tunnel circuit. And since the best effort circuits have no guarantees, the tunnel circuit capacity could be quite small – there was no performance guaranteed for those internal circuits,… so even poor performance would be - strictly speaking - compliant with the service request.

Other virtual circuits with performance guarantees could be provisioned along side the tunnel and they would get along fine since all have enforced limits. This tunneling of best effort “scavenger” circuits within a guaranteed capacity tunnel allows the provider to manage the quantity of both best effort and guaranteed traffic on a link.

In a generic virtualization model, it should be noted that components that form the underlying infrastructure for the virtual mapping process, could in fact be virtual resources created beforehand by some prior virtual mapping process. For example: A VM resource is mapped to a server by the VM allocation agent. The server could be a virtual resource that the VM allocation agent requested from the Bare-Metal-Server allocation agent. Similar examples can be made of Virtual Circuits acting as infrastructure trunks in which other Virtual Circuits are allocated. So the terms “resource” and “infrastructure” are not fixed or absolute roles in the virtualization process, but rather relationships of objects to the virtualization mapping process in a particular layer.