Provisioning Voice Over Packet Networks: A Metadata Driven, Service Object Based Approach

Jung Tjong, Prakash Bettadapur and Alexander Clemm
Cisco Systems, Inc., 170 West Tasman Drive, San Jose, CA 95134-1706, USA
E-Mail : {jtjong, pbettada, alex}@cisco.com

Voice over Packet (VoP) technology has made significant inroads into service providers' infrastructure, leading to the convergence of voice and data networks. As voice functionality gets distributed across the network, VoP management systems need to address the challenges of coordinated provisioning across multiple devices. To be able to cope with the many flavors of VoP networks that exist and the multitude of different network elements that they consist of, it becomes a necessity for the provisioning system to be custom programmed. This paper presents a metadata driven approach, based on a concept of Service Objects, that allows a provisioning application to easily handle variations in network architecture, equipment types, and equipment versions. We describe our implementation experiences with a VoP management system that leverages our approach and its flexibility, utilizing XML definitions and Enterprise Java Beans as implementation technologies.

Keywords: Network management, Provisioning, Open Packet Telephony, VoIP, EJB, XML.

1. Introduction

Voice over Packet (VoP) technology in general and, with the dominance of IP, Voice over IP (VoIP) in particular has made significant inroads into service provider infrastructures. It provides an excellent value proposition, allowing service providers to integrate voice services with many value added services on the same infrastructure as data for significantly lower costs, leading to the convergence of voice and data. A wide variety of relatively inexpensive and simple (hence easy to manage) network elements (NEs) provide a high degree of flexibility to deploy high-bandwidth, carrier-class network services. Operations of the converged network is simplified over having to maintain separate, dedicated networks for different services. However, at the same time new challenges for voice network management are introduced. Compared to networks of traditional, more monolithic TDM switches, voice functionality has now become much more distributed across multiple devices, and multiple types of devices, across the network.

One aspect of particular importance is provisioning, which is critical to successful deployment of voice services on packet infrastructure. Due to VoP's distributed nature, provisioning now requires a greater deal of coordination than it used to. Just as importantly, many different types of network elements can participate. For instance, in principle any router that implements MGCP can participate as Media Gateway in an MGCP-based VoIP network. Routers can be of different versions, different types, and different vendors. This leads to an unprecedented flexibility in terms of network engineering, but also unprecedented heterogeneity of devices to be accounted for by provisioning systems for voice services.

This paper discusses experiences gained in the development of a VoP management system that includes the capability to provision VoP networks. An overview of the system can be found in [1]. We will focus specifically on provisioning and how the challenges relating to the distributed and highly heterogeneous nature of VoP are addressed. One of the design goals was to be able to easily integrate management applications with various types and versions of network devices, network architectures and control protocols, with initial focus on MGCP. To this end, we propose an approach based on a concept that we call Service Objects, driven by metadata, to provision the network. This concept allows to deal with the dynamic, heterogeneous and distributed nature of VoP networks rather nicely, making it easy to adapt the system to different network architectures and rapidly incorporate new types and variations of devices. We describe our implementation experience building a provisioning application as part of a VoP management system, which confirmed the flexibility of our approach. Our system leverages XML [9] and Enterprise Java Beans (EJB) [4] as implementation technologies whose value in building management applications has been documented before (e.g. [2, 5]). No inferences about Cisco's product direction or product features should be made.

Section 2 gives an overview over the managed VoP network architectures and outlines some management considerations. Section 3 presents our VoP management system, which includes the provisioning subsystem, and its architecture. Section 4 elaborates challenges in building VoP provisioning applications, including issues related to object model granularity and versioning. Section 5 introduces the concept of a Service Object (SO) and discusses a supporting architecture that allows SOs to be meta-driven, as well as some implementation considerations. Section 6 describes our implementation experience. Finally, section 7 presents some conclusions.

2. VoP Networks and Management Considerations

Different types of Voice over Packet networks exist, distinguished along the lines of the control protocols they are based on. As mentioned, we were initially concerned with management and provisioning of MGCP-based networks, although we expect our concepts to apply analogously to networks based on H.323 or SIP. (Ku et al have previously described a system for H.323 provisioning [3].) Great flexibility exists in engineering those networks, which therefore come in many different variations. However, they all underlie the same common theme, which is the separation of voice functionality into planes:

The bearer plane deals with transport of the payload. The devices at the edge of the bearer plane are referred to as Media gateways (MGs). They constitute the entities through which the bearer traffic enters the VoP network. When interconnecting VoP and TDM networks, it is the MGs who provide for the TDM to packet (and vice versa) conversion. Routers between MGs provide for the actual bearer fabric (the "data cloud"), shuffling data packets back and forth.

The control plane is responsible for signaling and call control and provides the actual "intelligence" of the network. Its components are commonly referred to as Media Gateway Controllers (MGCs). MGCs control MGs through the MGCP protocol, instructing them when to set up or tear down connections, requesting notification of specific events for further processing, etc.

This decoupling allows the bearer plane to be shared with other services, which might be controlled through their own control planes, utilizing the same transport infrastructure. Please refer to the literature (e.g. [6, 8]) for a more detailed description and further refinements on this subject. Collectively, bearer and control plane can provide functionality analogous to that of traditional PSTN switches, and thus logically replace them. Therefore we term a set of associated MGs and MGCs and the network connecting them as "virtual switch". (This is not to be confused with the term soft switch, which is often used synonymously with MGC but does not include the associated MGs.)

As far as management is concerned, some of the very properties that make VoP technology so attractive also make its management a challenge, including the distribution of functionality, openness, and flexibility. To deploy voice services effectively, all components of the virtual switch must be provisioned in a coordinated way. Please consider the following example borrowed from [1], which lists the steps that are required in establishing a PRI service. The example refers to signaling backhaul, i.e. the concept of transporting signaling payload between the MG and the MGC, as illustrated in figure 1. Signaling backhaul becomes necessary in the event that signaling physically terminates on the MG – the MG, not able to process signaling, needs to forward it to the MGC.

1. Add a line with TDM endpoints and a CCS channel on a Media Gateway;
2. Instruct the MGC to add a new trunk group and associate it with the customer;
3. Instruct the MGC to add the trunks;

4. Associate them with the according MG endpoint;
5. Verify a signaling backhaul connection has been set up (also, depending on the OPT solution, primary and secondary need to be dealt separately for reliability purposes);
6. Set up signaling backhaul connection if required (which involves adding signaling backhaul terminations at both MG and MGC, as well as possibly setup of a layer 2 connection to carry the signaling backhaul connection (e.g. an AAL5 PVC in VoATM deployments);
7. Set up a cross connect between CCS channel and signaling backhaul connection at the MG if required (depends on the type of MG).

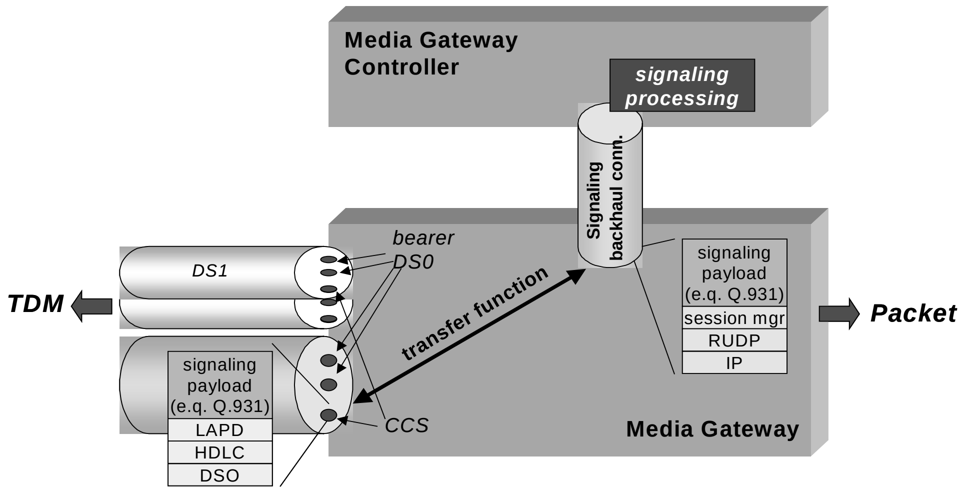


Figure 1 - *Signaling backhaul and PRI*

This illustrates that many steps can be required to complete the provisioning of a single service, which makes it inefficient and error prone to provide manually. If any of these steps fail, the management system should be robust to retry intermediate steps, or roll back partly completed steps.

3. OMS and its Architecture

In order to deal with the management challenge of VoP networks, we have built a management system, which we will dub “OMS” (Open packet telephony Management System). At its core, OMS provides a set of functions that allows users to operate in the more meaningful context of a virtual switch, which in the above example would allow the operator to simply request “add PRI service on the specified line for a specified customer”. So, instead of requiring individual operations to be performed at each of the network elements, OMS offers functions such as the following:

- Turn up/tear down/modify service for a customer
- Add/remove/modify a customer
- Create/delete/modify trunks, trunk groups, routes, route lists
- Associate or disassociate an MG from a virtual switch

All these functions are provided as if they were single operations, hiding the underlying operational complexity of having to deal with multiple operations across multiple network elements from the user, thus greatly improving operations efficiency and accuracy. In the following we will focus on the provisioning application that is provided by the system.

The basic OMS architecture as it pertains to provisioning is shown in Figure 2. It consists of the following components:

- The Managed Object (MO) Repository realizes the conceptual object model required by the applications.
- The Discovery Modules interacts with the underlying EMSs and populates the MO Repository.
- The Behavior Modules implement specific MO operations.
- The Provisioning Subsystem consists of modules for configuration management.

The Provisioning Subsystem functions by creating, modifying, deleting, or invoking actions on the MOs in the MO Repository. The MO Repository shields applications from having to interact directly with the network elements respectively their element management systems (EMSs), making transparent the specific way in which the NEs are accessed. Behavior modules and discovery modules implement MO operations and populate the MO repository respectively.

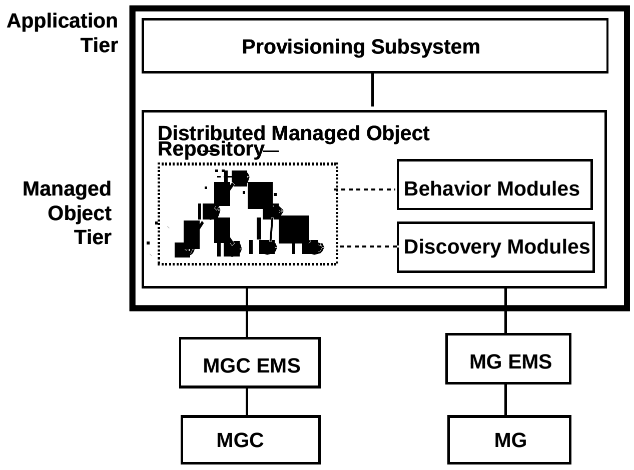


Figure 2 – System Architecture

There are two categories of MOs, as depicted in figure 3: Virtual Entity MOs provide the abstraction of the virtual switch as a whole and the components and services that it provides, aggregating information across network devices and coordinating between them as necessary. They essentially form a management hierarchy on top of Device MOs, which represent the actual network elements and their resources, such as MGs and MGCs. An M.3100 based object model [7] is used to model the Device MOs and the Virtual Entity MOs. It is generic enough to accomodate multiple network architectures and different flavors of MGs and MGCs. The model can also be extended to support multiple control protocols. As we shall see in the next section, despite the model's power, there are a lot of practical issues to utilize it directly for provisioning applications in an environment as dynamic and heterogeneous as VoP.

4. Challenges in Building a VoP Provisioning Application

Key to our provisioning application and indeed to our entire management system is the object model on which to operate. Providing the object model in a middle tier in an n-tier management system architecture provides a great advantage of allowing to separate the application logic from details of how to interact with the underlying systems and NEs. This makes the system more robust and easier to maintain.

In general, new variations of NEs that map to the same abstractions, i.e. MOs, will lead to extensions in the MO layer. For instance, extensions in the behavior and discovery modules will be required for those MO classes whose mapping is affected by the new variation, as it differs from before. In this case, no application layer modifications are necessary. However, when a new NE introduces some new properties that need to be considered in provisioning, the object model itself will have to be extended. The generic model that the management is based on provides the necessary flexibility to handle multiple flavors of MG and MGCs, and can accommodate the extensions. Of course, in addition to extensions in the MO layer, changes to the application itself become necessary in order to account for the new extensions in the object model.

In the VoP management domain, those kinds of scenarios occur constantly. For instance, new types of Media Gateways that are introduced into the network must be supported. A single network may also have multiple versions of the same components. For example, some of the MGs may be at version 1.1, and some others at release 1.2. It is not practical to expect that the network be upgraded to the same level at the same time. Even with a concept as compelling as that of an MO layer, this environment where multiple flavors of nodes and multiple versions of object instances have to be accommodated on an ongoing basis introduces challenges for the development of provisioning applications. Those challenges include the tradeoff between fine-grained and coarse-grained object models, i.e. flexibility and ease of extensibility vs. performance considerations, and the ability to keep up with new variations of devices that have to be mapped to.

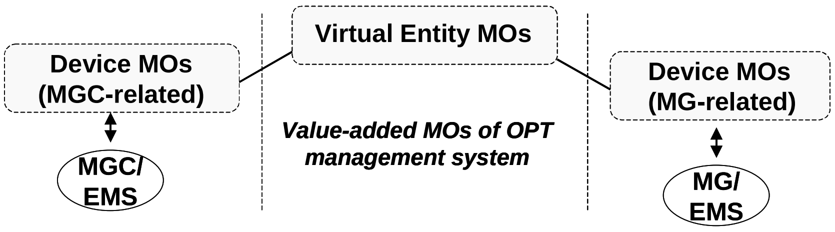


Figure 3 - MO categories

4.1. Object Model Granularity

An object model in the MO Repository allows the OMS applications such as the provisioning subsystem to work with multiple implementations of the network elements such as different types of MGC and MG by factoring out their common facets into common classes and placing their differences into subclasses. In general, fine-grained models are able to accommodate new types of devices much more easily than coarse-grained models, which have a tendency to break in such scenarios. This is because the fine-grained model tends to separate out the atomic concepts in a domain, which can then be combined at will in virtually any variation, while coarse-grained models need to make assumptions about how some of these concepts are combined in a particular domain. The ability to adapt to any kind of VoP network was a key goal for our system; hence we decided to go with a fine-grained model. This introduces a set of issues for our system to deal with:

- Minimizing application complexity. A fine-grained model can lead to issues of application complexity vs. a coarse-grained model, because the application has to worry much more about how to combine different concepts that are scattered around, linked by a multitude of relationships. Multiple objects per operation implies more complexity in getting, and especially creating, modifying and deleting the objects as transactional control, both across the objects and the network element entities that the objects represent. Certainly, a user or client application will not want to be exposed to these issues. However, they apply even if the transactional control is delegated to the Virtual Entity MO, which in turn handles its dependencies with other MOs, as opposed to an application.
- Ease of use. A fine-grained model introduces the issue of how to collect information that is required for operations that affect multiple objects in a way that does not make the system cumbersome to use. Let's consider the example of a "virtual switch trunk" MO, which in turn depends on a DS0 MO modeling the physical endpoint on the MG, and a trunk object on the MGC, modeling the MGC's properties of controlling that DS0. (In our system, the model is actually optimized not to model DS0s individually.) There are attributes that are specific only to the MG or to the MGC, which hence should be modeled as part of the MG or MGC object respectively. An example would be, for instance, echo cancellation parameters, which apply only to the line side. A user, however, may want to operate on the concept of a virtual switch trunk, without concern about what other objects the virtual switch trunk is composed of, since having to deal with MG and MGC objects separately would defeat the purpose of our management system. In presenting a virtual switch trunk to the user (or an application), the system will have to include those attributes which (internally) pertain only to the MG or the MGC side. There are different ways to deal with this from the modeling side, all of which have some drawbacks:
    - MG and MGC attributes could be modeled only as part of the Virtual Entity MO, not as part of the Device MOs. This would however compromise the model's precision and introduce issues with regards to synchronization and discovery.

- The Virtual Entity MO could duplicate, or "cache" relevant attributes of Device MOs that need to be exposed to the user. The drawback is that this introduces other practical issues with regards to maintainability (changes requiring extensions to MG or MGC objects would now also impact Virtual Entity MO) and object model integrity.
- The Virtual Entity MO could offer actions with parameters that map to the Device MO attributes, so the model information is implicitly included in action parameters. Again, it introduces issues with regards to maintainability and maintaining integrity of actions and object model.

In addition, users and applications will want certain operations to extend over multiple MOs, such as when requiring a list of all trunks in a virtual switch. The system needs to provide object services to allow for these types of operations.

All this means while for system implementation purposes we prefer a fine-grained object model, the user will want to operate on a coarse-grained object model. This difference has to be reconciled, and is key to how our system works. At the provisioning subsystem level, applications such as a GUI or an OSS could be greatly simplified if the services are already presented at this coarse-grained concept. Using virtual switch trunks as example, these high level applications should be able to get virtual switch trunks, modify virtual switch trunks, create a new virtual switch trunk and delete virtual switch trunks, not the individual fine-grained objects. Our approach presented in section 5 provides just that: it allows to retain the benefits of a fine-grained object model at the managed object layer in terms of maintainability and extensibility, yet offers users and applications the benefits of a coarse-grained object model which is exposed through SOs that are driven by meta-data to require minimum development or customization effort.

4.2. Variations in the Managed Objects

Having a generic object model in theory shields the application from the actual entities they are representing. In reality there are always variations among network element implementation or types and versions of the same element type, requiring different mappings as implemented for instance in the behavior modules. Some of the types and versions introduce slight modifications at the model level, such as varying ranges for a parameter depending on NE capacity (e.g. 1..100 or 1..1000). In some cases, a new parameter may need to be supported. Some attribute constraints may even have to be determined at run-time based on some formula, state information or current configuration.

In general, the client application should be aware of those variations so that input ranges can be checked at the client level and the right choices requested from the user, as opposed to having to engage in trial-and-error behavior. Extending and modifying the object model for all such cases can be impractical and requires changes not only in the MO layer but also the client that makes use of it. When applications have to simultaneously support multiple types and versions of the network element components, implementing those rules on the client application side quickly becomes unmanageable from a development perspective. A lightweight, dynamic extension capability on top of the existing object model that allows to account for minor deviations and variations of the object model and its behavior is much preferable. Introspection and dynamic invocation capabilities that allow clients to be generically implemented are addressed by our approach that will be presented in the next section.

5. The Service Object Concept

In this section, we introduce the concept of Service Objects (SOs) to address the provisioning challenges discussed earlier. SOs provide a particular view on the object model provided by the underlying MO layer. These views allow the provisioning application to be decoupled from the fine granularity and variations of the model provided by the MO layer. Because they are driven by meta-information, they are easily introduced dynamically. The following concepts are distinguished:

- The instance information exposed through this view is referred to as the *Service Object*. The abstract concept that a view represents is referred to as the *SO type*.

- The definition of a view is referred to as the *SO definition*. It is specified using XML. There can be multiple SO definitions for an SO type, reflecting different variations of the same concept as will be explained below.
- SOs are not "objects" that are exposed directly. Rather, they are views that are accessed and operated upon through a dedicated set of server objects, called *SO components*. There is one set of SO components per SO type. SO components operate on the XML-based SO definitions to provide the specified views, i.e. they are meta-driven.

The following subsections elaborate these concepts in more detail.

5.1. Service Objects

A Service Object is a view that maps to one or more managed objects. It has parameters that map to attributes of the associated MOs, as specified in the SO definition. It addresses the issues outlined in section 4 as follows:

- Object model granularity. Because an SO can pertain to multiple MOs, it can effectively aggregate them into a more coarse-grained model to better address the users' and applications' needs. The SO not only exposes information to applications, it also allows applications to operate on this view as if it were a single entity.
- MO variations. To address the variability of the MOs, the dependencies between the SO parameters and the MO variations resulting from type and versions of the network element are specified declaratively as part of the SO definition. There can be multiple SO definitions per SO type. Only one SO definition is valid for any particular SO, depending on certain predefined properties of the MO that the SO maps to. Those properties are the containing network element type or version. This allows us to specify such dependencies and constraints only at the SO level without requiring any changes in the MO model. By building on top of SOs and interfacing to SO Handlers rather than the MOs directly, no modifications in client applications are necessary as they make use of the introspection and dynamic invocation capabilities that the SO components provide.

Again, SOs constitute just a view. We are not defining a new object model layer and we are not mediating MOs into a new model. No new persistent or transient object are created.

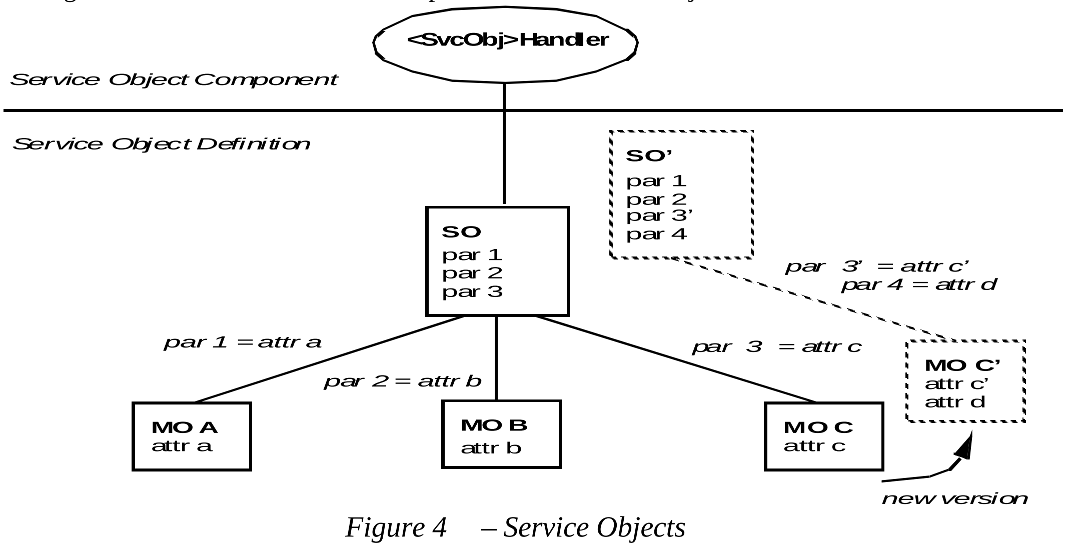
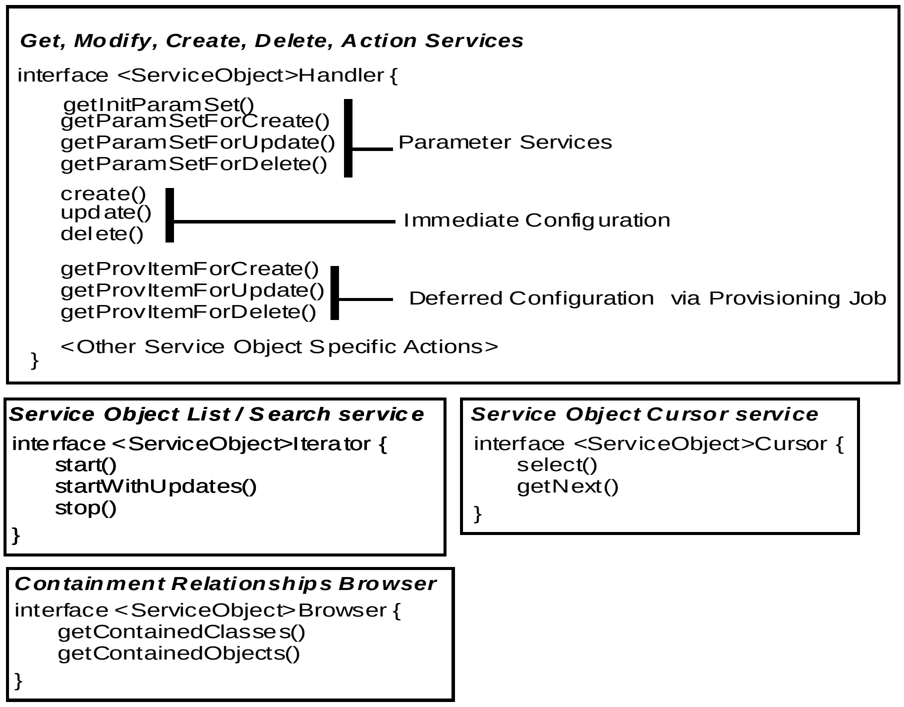


Figure 4 – Service Objects

Figure 4 shows a conceptual diagram of a Service Object that maps to three managed objects. `SO` has three parameters which map to attributes `a`, `b`, and `c` of MOs `A`, `B`, and `C`, respectively. Suppose an extension to the MO layer's object model is introduced, leading to a new MO class `C'` which includes a new attribute `d`. (A concrete example would be `SO` representing a trunk view, with `C` representing the line side of the MG, with `d` being a new line attribute for echo cancellation, which we would like to include in our trunk view.) A new view `SO'` needs to be defined which takes the new mapping to `C'` with the new attribute `d` into account. Both `SO` and `SO'` represent variations of the same concept (in the example, an application would think of both simply as a trunk). Whether an application invokes `SO` or `SO'` is handled transparently by the Handler, which determines the applicable SO definition for its SO type.

5.2. Service Object Components

Transparency and flexibility to client applications is achieved by providing a set of services that allow provisioning applications to operate on SOs as if they were MOs, creating, retrieving, modifying, and deleting SOs, based on the declarative SO definitions. These services are provided by a set of application level server components, called SO components, that in our case are implemented as EJBs. These components are Handler, Iterator or Cursor, and Browser, as illustrated in figure 5. As mentioned before, there is one set of components for each SO type. SO components are meta-driven by SO definitions.



Get, Modify, Create, Delete, Action Services

```
interface <ServiceObject>Handler {
     getInitParamSet()
     getParamSetForCreate()
     getParamSetForUpdate()
     getParamSetForDelete()
     create()
     update()
     delete()
     getProvItemForCreate()
     getProvItemForUpdate()
     getProvItemForDelete()
     <Other Service Object Specific Actions>
}
```

Service Object List / Search service

```
interface <ServiceObject>Iterator {
     start()
     startWithUpdates()
     stop()
}
```

Service Object Cursor service

```
interface <ServiceObject>Cursor {
     select()
     getNext()
}
```

Containment Relationships Browser

```
interface <ServiceObject>Browser {
     getContainedClasses()
     getContainedObjects()
}
```

Figure 5 Service Object Component Interfaces

The Handler is a component that allows for SO creation, modification and deletion, as well as Provisioning Job related services. (Provisioning Job services are to be used with Provisioning Job components that provide a batch/scheduled provisioning services. This paper will not discuss this aspect further.) In addition, the Handler provides services to retrieve the SO parameters of the SO definition, allowing client applications to discover how an SO has been defined. Since the Handler component is the most critical component, its services are described in more detail in section 5.3.

The Iterator is a component that provides list or search services to find SOs based on some criteria. This component only returns summary information about the SO and not the entire parameter values of the SO. The SO itself can be retrieved using the Handler component. A client will receive SOs asynchronously, and option to receive updates to keep the list up-to-date is available. The Cursor component is the synchronous version of the Iterator and provides database cursor like functionality.

The Browser is an optional component that can be used to display a containment hierarchy of the elements associated with a SO. For instance, the Browser component could return a physical containment of the equipment. One of the purposes of this component is for applications to be able to choose resources used in creating a new SO. Another is for selecting an element to be used for selection criteria in searching SOs.

5.3. Service Object Handler Services

All Handler components implement the same basic and generic service interface. Information about SOs is returned in the form of a set of parameters, which map in a specified way to the underlying MOs. To create or modify SOs, the Handler similarly takes the input as a set of parameters. Thus no SO specific interfaces are needed to perform the basic services of retrieval, creation, modification and deletion. Since a Handler component is provided per SO type, a customized SO specific interface can be provided if required. This approach is similar to the Attribute Value Assertion (AVA) concept in the CMIP protocol or the VarBind concept in SNMP but with a difference. The client of CMIP or SNMP services generally must know what parameters or attributes must be specified for the object as well as constraints for their values. Usually the knowledge about variations must be coded into the client applications. While some constraints are defined in the object definition, these definitions are static and not necessarily implemented by the specific object that the client is interacting with. On the other hand, the SO Handler component that processes the create, modify and delete requests allow clients to discover the required parameters and their rules using the parameter services interface (see *getInitParamSet(), getParamSetForCreate(), getParamSetForUpdate()* and *getParamSetForDelete()* in Figure 4.). The rules are returned as parameter properties and constraints. They are generated by the SO Handler component based on the static definition of the parameter properties and constraints in the SO definition and run-time behavior coded in the Handler component. For example, the list of allowed value may be constructed and retrieved at run-time from the network element itself.

In essence the interaction between the SO Handler and its clients resembles the interaction between an application and a human user. When a user creates a new object through a graphical user interface, typically the application would ask what type of object to be created and some initial parameters that are known to apply generically. Then it would prompt the user for the parameters that need to be specified through a series of forms. These forms usually perform value checking for the parameters. This contrasts with a (less friendly) interface in which a user would have to know a priori which parameters are required and what constraints apply, having to provide them without being prompted.

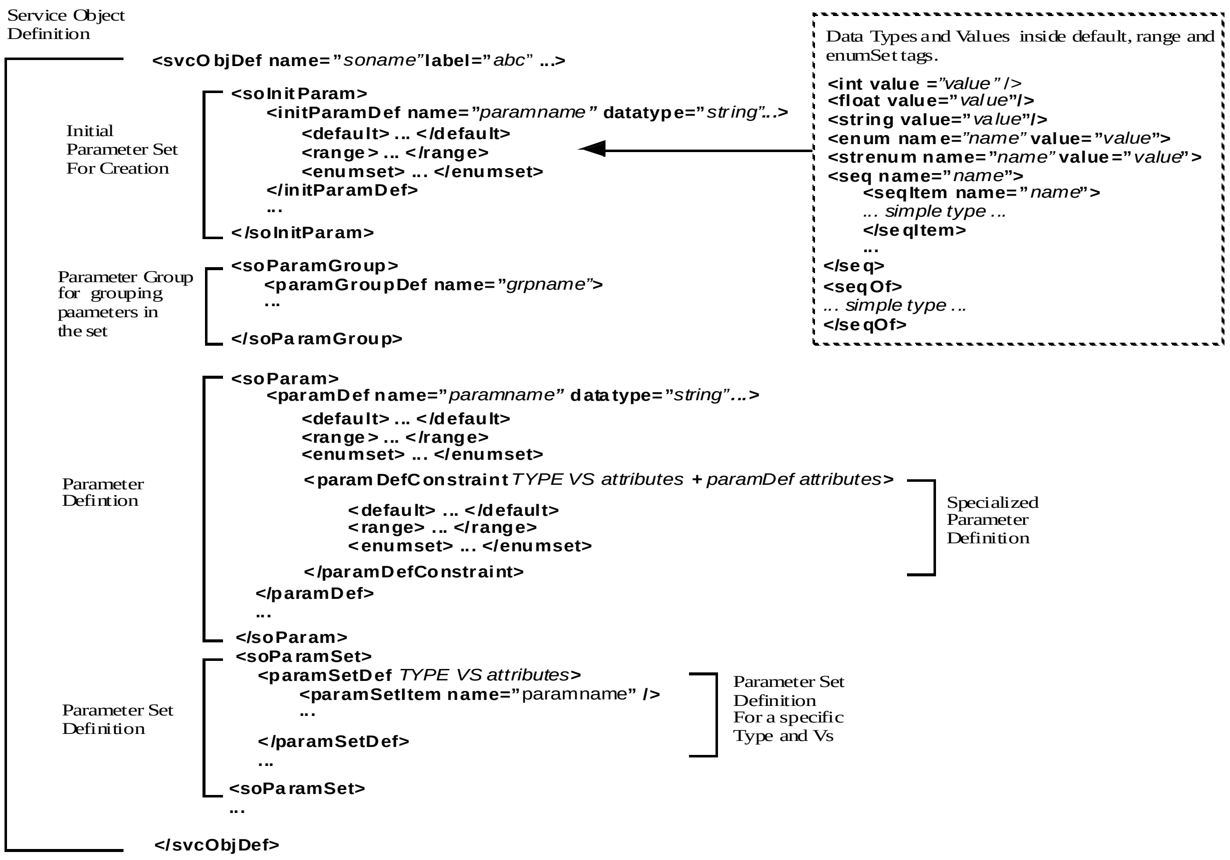
The Handler component is well suited to implement such applications. The parameter set and its rule can be used to render the form. Once the form is populated, the parameter set is sent to the Handler either by invoking the *create()* method directly or by invoking *getProvItemForCreate()* to get provisioning items that can be added into a provisioning job. The Handler component also contains business logic that will perform any checking and validation that go beyond parameter checking.

5.4. Service Object Definition

The SO Definition serves as a meta-data for the SO Handler component in providing its parameter services. Thus the rules are not hard-coded but data-driven. XML is used to define SOs because of the hierarchical nature of the content structure and the wide availability of the XML parser. This standard has become widely used recently as a format for exchanging data between applications and for configuration files. An SO definition contains two basic parts:

- The SO parameter definition specifies the properties of SO parameters, such as data type, constraints, associated Managed Object class and attribute, and presentation information like label and a short help text. These properties can be overridden based on the type and version of the MGC, MG and MG card components in the definition itself. This facility can also be used to specify default, range or enumeration values or even parameter labels that are sensitive to type and version of MGC, MG and MG Card. All parameter properties can be specialized this way.
- The parameter set definition specifies the SOs themselves, i.e. their parameters and appropriate subsets for various types and versions of the MG, MGC and MG card components of the Virtual Switch.

Each SO type is associated with a configuration file. The structure of the configuration file is shown in Figure 6. A configuration file contains one or more SO definitions. Each SO definition contains four basic sections:



```
<svcObjDef name="soname"label="abc" ...>
  <soInitParam>
    <initParamDef name="paramname" datatype="string"...>
      <default> ... </default>
      <range> ... </range>
      <enumset> ... </enumset>
    </initParamDef>
    ...
  </soInitParam>
  <soParamGroup>
    <paramGroupDef name="grpname">
    ...
  </soParamGroup>
  <soParam>
    <paramDef name="paramname" datatype="string"...>
      <default> ... </default>
      <range> ... </range>
      <enumset> ... </enumset>
      <paramDefConstraint TYPE VS attributes + paramDef attributes>
        <default> ... </default>
        <range> ... </range>
        <enumset> ... </enumset>
      </paramDefConstraint>
    </paramDef>
    ...
  </soParam>
  <soParamSet>
    <paramSetDef TYPE VS attributes>
      <paramSetItem name="paramname" />
      ...
    </paramSetDef>
    ...
  <soParamSet>
  ...
</svcObjDef>
```

Data Types and Values inside default, range and enumSet tags.

```
<int value ="value" />
<float value="value"/>
<string value="value"/>
<enum name="name" value="value">
<strenum name="name" value="value">
<seq name="name">
    <seqItem name="name">
    ... simple type ...
    </seqItem>
    ...
</seq>
<seqOf>
... simple type ...
</seqOf>
```

Service Object Definition
Initial Parameter Set For Creation
Parameter Group for grouping paameters in the set
Parameter Defintion
Specialized Parameter Definition
Parameter Set Definition
Parameter Set Definition For a specific Type and Vs

Figure 6 - Service Object Definition XML Structure

- The initial parameter set defines the parameters that user need to specify in creating a new SO. The information in the type and version information is used to select the parameter set.
- The parameter group definition is to define a logical grouping of parameters in the parameter set. This can be used, for example to group parameters together in the user interface form.
- The parameter definition specifies properties associated with the parameters. A parameter definition constraint section allows to specify or restrict the properties to certain NE types and versions. For example, the TYPE VS attributes could be `mgc_type`="vsc" and `mgc_vs`="8.1" to indicate that the particular parameter only applies for MGCs of type VSC with version 8.1.
- The parameter set definition specifies the parameters that are applicable to a particular NE type and version. It includes two attributes `moClass` and `moAttr` that specify which Managed Object class and Managed Object Attributes are associated with the parameter.

6. Implementation Experience

In this section we will describe how the SO concepts have been realized in the OMS Provisioning Subsystem. We will also present interactions between a client and the SO components to further illustrate their usage.

6.1. Provisioning Subsystem

The OMS Provisioning Subsystem provides functions for configurations associated with day-to-day VoP service provisioning. Among the frequently performed configurations are adding, modifying and deleting PRI-Backhaul, Trunk Group, Trunk and Route List. These entities constitute SO types and are represented in the MO repository as multiple MOs.

We choose to implement the SO components using Java and EJB for ease of implementation and Java GUI integration (by allowing Java objects to be passed between the component and the GUI). For each SO type we introduce its own Handler and Iterator components. For example, the Trunk-Handler and TrunkIterator implement the Handler and Iterator components for Trunk SO. The Browser components are shared among the SOs since the resources needed to provision them can be presented with the same set of objects. We followed the SO component pattern to also create a provisioning job handler component that provides services for creating, modifying and deleting provisioning jobs, which control batches of prescheduled provisioning activities. Because of the transactional nature of provisioning jobs, they are persisted in a repository.

Using the SO components we are able to create a Java based GUI that is largely generic (SO specific behavior and business logic is provided by the Handler component). We design a GUI frame, shown in Figure 7, that displays a tree view on the left panel, and list view on the right panel. The content of the tree view is driven by the data returned by the Browser component and would show a virtual switch containing the MGs with their physical equipment hierarchy and the MGC with its logical components, such as route lists. The content of the list view is driven by the Iterator component. The list view frame is replaced by a form to display SO parameters when users create a new SO or modify one of the existing SOs from the list in the list view.

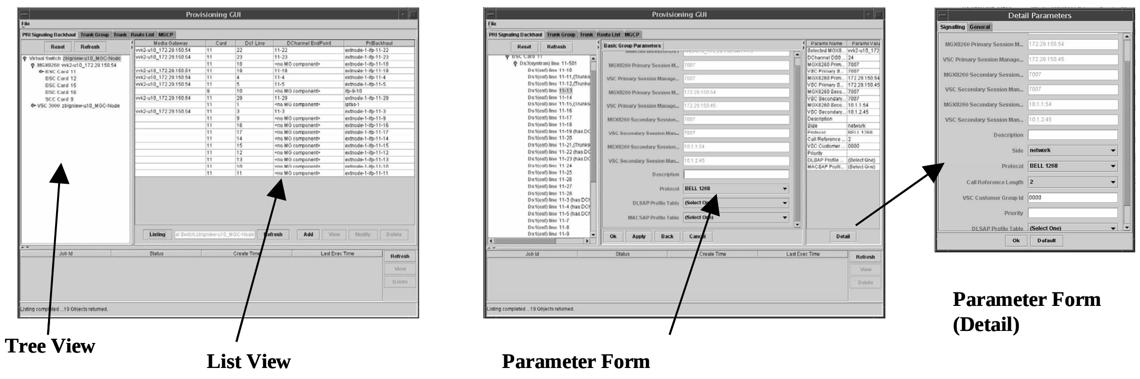


Figure 7 GUI Display

The GUI also shows a list of provisioning job that has been submitted and their status. When user creates, modifies or deletes one or more SOs using a provisioning job, a new provisioning job is created in the Provisioning Object Repository. The behavior associated with the provisioning job in the repository would then execute the newly created provisioning job and update its status after it has been processed. We will discuss the interaction between the GUI and the Handler component in a PRI-Backhaul creation and modification.

6.2. GUI and Service Component Interactions

There are two options for provisioning an SO: immediate provisioning that applies the configuration immediately to the NEs, or provisioning through provisioning job that applies the configuration in the background at a prescheduled time. Here we will illustrate the provisioning of a PRI backhaul that would require the use of a provisioning job.

The first step in the configuration process is for the GUI to create a new provisioning job request. The second step is to fill it with provisioning job items related to creation, modification or deletion requests that are returned by the SO Handler component. Finally when the provisioning job request is completed, the GUI submits the provisioning job request to the provisioning job Handler component. When a user adds a new PRI backhaul, the GUI first retrieves the initial parameter set using the Handler's `getInitParameterSet()` and displays the parameters to the user. For PRI backhaul, the initial parameter set consists of just one parameter: a string where user must specify the MG component name, the card number and the DS1 line number. The tree view constructed by the Browser component can be used to simply drag a DS1 line into the parameter field. Next, the GUI invokes the Handler's `getParamSetForCreate()` method to retrieve the parameter set for the new PRI backhaul. The parameter set and the parameter properties are determined based on the DS1 line that is selected, which in turn determines the MGC, MG and MG card types and versions. These parameters are used by the GUI to render the parameter form and initialize their particular default values. To make the form generic, we even use one of the parameter properties to specify what Java GUI class to use to receive the parameter value from the user. The PRI backhaul parameter set consists of attributes from three objects under the MGC, one object under the MG and one object under virtualSwitch. After the parameters are specified, the GUI invokes the Handler's `getProvItemForCreate()` method to obtain the provisioning item that it can insert into the previously created provisioning job. The provisioning process is completed when all the provisioning items are added and the provisioning job is submitted to the Provisioning Job Handler component.

The interaction to modify a SO is similar to the creation process. In this case the parameter set is returned by the Handler's `getParamSetForUpdate()` method and the parameters initialized with their current values.

7. Summary

In this paper, we presented the concept of Service Objects which are driven by XML-defined metadata to facilitate the provisioning of VoP networks. This approach allows provisioning applications to keep up with the high degree of heterogeneity (with respect to NE types, versions, and network architectures) of those networks. It bridges the gap between coarse-grained concepts needed by users and applications to interact with the provisioning system, and fine-grained concepts that are well suited to keep underlying object models and applications generic. While our target application was the provisioning of VoP networks, we would expect our concepts to apply and provide benefits also to other domains with similar problem characteristics. We have successfully applied these concepts in building a VoP management system for MGCP-based networks. Possibilities for future work include the providing of tools that facilitate the automatic generation of SO definitions.

8. References

[1] Clemm, A., P. Bettadapur: *Building management solutions for Open Packet Telephony networks*. IEEE/IFIP Integrated Management (IM 2001), Seattle, WA, 5/2001.

[2] Ensel, C., A. Keller: *Managing Application Service Dependencies with XML and the Resource Description Framework*. IEEE/IFIP Integrated Management (IM 2001), Seattle, WA, 5/2001.

[3] Ku, H., J. Forslow, J. Park: *Web-Based Configuration Management Architecture for Backbone Router Networks*. IEEE/IFIP Network and Operations Management Symposium (NOMS 2000), Honolulu, HI, 4/2000.

[4] Matena, V., M. Hapner: *Enterprise Java Beans Specification v.1.1.* Sun Microsystems, 11/1999.

[5] Sengul, S., J. Gish, J. Tremlett: *Building a Service Provisioning System Using the Enterprise Java Bean Framework*. IEEE/IFIP Network Operations and Management Symposium (NOMS 2000), Honolulu, HI, 4/2000.

[6] Cisco Systems: *An Introduction to Cisco Open Packet Telephony for Service Providers*. White paper, 3/2000 (http://www.cisco.com/warp/public/cc/so/neso/vvda/pctl/opt_wp.pdf).

[7] ITU-T M.3100: *Maintenance – Telecommunication Management Network – Generic Network Information Model*. 7/1995.

[8] MSF-ARCH-001-FINAL IA, D. McDysan (editor): *System Architecture Implementation Agreement*. Multiservice Switching Forum, 5/2000.

[9] W3C Recommendation: *Extensible Markup Language (XML) 1.0*. 10/2000.