Using the Component and Aggregate Abstractions in the GENI Architecture
GDD-06-42

GENI: Global Environment for Network Innovations

Version of December 19, 2006

Status: DRAFT Document (Version 0.2)

This document is an unreleased work in progress that continues to evolve rapidly. General release outside of the GENI PG/WG community is not appropriate at this time, although comments from all within this community are solicited and welcome.

Certain aspects of the GENI architecture, with potential effect on this note, are not yet addressed at all, and for those aspects that are addressed, a number of unresolved issues have been identified in the relevant documents. Further, due to the active development and editing process, some portions of this document may be logically inconsistent with others in the series.

This document was prepared by the Facility Architecture Working Group.

Author: John Wroclawski, *USC/ISI*

This work is supported by NSF grant CNS-0631303.

Revision History

Version	Changes log	Date
v0.1	Original version posted	10/28/06
v0.2	Figures, minor updates	12/19/06

Table of Contents

1 Introduction

The GENI Architecture Overview [GDD-06-11] defines and describes two system-wide abstractions of fundamental importance; Components and Aggregates. However, little guidance is provided in that document as to the use of these abstractions. The purpose of this note is to expand on the architecture overview by considering aspects of the component and aggregate abstractions in more detail, and discussing some ways these abstractions might be used to describe different kinds of components in practice.

We start by revisiting the definition of a component, with slight additional detail. Three basic properties of a component are important to the material of this note.

1. A component implements a logically related collection of resources of value to GENI users.
2. Each component can be characterized by two fundamental properties:
    a. An RSpec, which consists of
        i. A list of resources that can be allocated and managed, and
        ii. A set of constraints that express relationships among those resources.
    b. A set of access rights, controlling which principals in the system can acquire and manage the component's resources.
3. Each component is controlled by a *Component Manager*, or CM. The CM exports a standard interface that allows users and system services to acquire, allocate, and manage resources within the component.
4. Each component is named, and this name is registered in a (the..) *Component Registry* with a reference back to the component. Registration of the component allows it to be located by users and system services.[1] The reference allows these users or services to invoke operations at the component's Component Manager, if access controls permit. Naming and Registration of Components is described further in [GDD-06-11].

    Technology Decision: In the current implementation, the component name includes a URI. This URI is used to invoke Web Services based operations at the component's Component Manager, which implements the defined component Interface.[2] Note that these control operations do not flow through the registry, they are carried out directly between the component exporting an interface and the user or service invoking that interface.

    Engineering Note: It may be appropriate to explore whether the Web Services registry mechanism UDDI is suitable for use as a component registry in this context.

[1] These system services might in turn provide higher-level component selection functions. The component registry is just the low-level starting point for this hierarchy of services.

[2] The Component Interface is accessed through a Web Services URL associated with the component's name.

2 Canonical Implementation

Much discussion of GENI components to date has assumed a canonical implementation model, with the following general properties.

1. The component is implemented on a platform, such as a general-purpose computer or blade server, that includes both a general-purpose CPU of some sort and a direct connection to the GENI control plane.[3]
2. The "component manager" controlling this component is implemented within the component, using the general-purpose CPU. The component manager for the component can be reached through the GENI control plane's connectivity.

Figure 1 shows important elements of this canonical implementation. This implementation pattern is well suited for a significant number of GENI components.

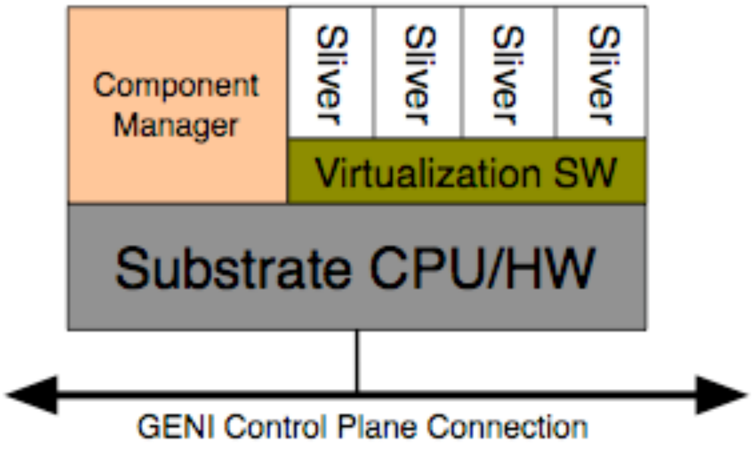

Figure 1: Elements of a Canonical Component Environment

3 Separating the Component Manager and Components

The canonical implementation of Section 2 shows the use of one Component Manager per component, with the Component Manager software embedded within the component's operating system. However, there are cases where it is appropriate to decouple these two functions, and to implement the Component Manager on a separate control computer. In many such cases, it may be appropriate for a single Component Manager to support and manage more than one component.

[3] The concept of an explicit GENI control plane has not been identified in the architecture to date. We assume into existence this concept, represented in the current implementation and in this discussion by Web Services based control operations carried over the extant Internet. A future version of GENI could easily substitute a different connectivity path while preserving the same service-based operation model, and perhaps the same Web-based protocols, which are not intrinsically dependent on TCP/IP.

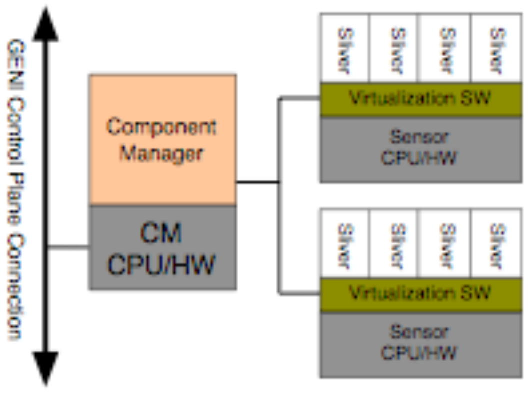

Figure 2: Remote CM Managing Multiple Components

Figure 2, above, shows one such example. In this figure, a single Component Manager, implemented on a general-purpose computer, implements and exports (registers) a separate Component interface for two separate components, perhaps sensor platforms. The result is that each sensor platform is visible to the system as a separate component, and can be configured and allocated independently by users and higher-level system services. However, the hardware and operating system of the sensor platform does not need to support a CM, and does not need to directly concern itself with registering or managing the component.[4] The form of the connection between the CM and the individual components it is managing is private to the CM and its managed components.

Because each component is separately visible, each component will have a separate name, and be registered separately in the component registry. However, the actual responsibility for implementing and interpreting the Component interface operations, and for registering each sensor Component, falls to the shared Component Manager running on the general-purpose computer. In this circumstance, the actual control channel mechanism and protocol between the Component Manager running on the general-purpose computer and the individual sensor platforms is not specified by the architecture, and may be private.

> **Engineering Note:** An alternative perspective that achieves exactly the same external result is that the general-purpose computer in the example runs multiple instances of a Component Manager, with each instance associated with a single sensor platform. In some circumstances, this may lead to simpler software design and facility management. The important concept common to both approaches is that a large number of lightweight Components can be managed and controlled from a single, non co-located Component Manager.

4 Synthetic Resources

The GENI Architecture Overview [GDD-06-11] describes the concepts of Resources and Resource Specifications (RSpecs). In this discussion, resources are generally viewed as

[4] A better picture here might show very small, perhaps non-virtualizable sensor platforms, such as Motes.

representations of physical (or virtualized) quantities, such as CPU cycles or network bandwidth.

However, it will sometimes be useful to describe and control a Component in terms of resources that describe frequently required low-level logical functions of the Component, independent of the actual physical elements used to implement the function.

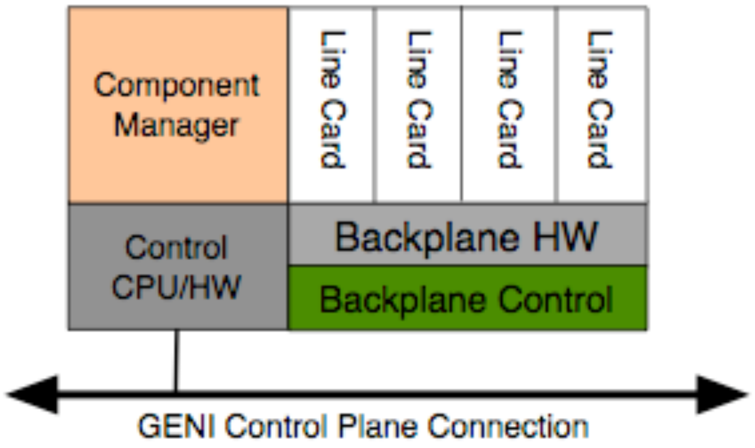

Figure 3: Programmable Router with Line Cards

Figure 3 shows an example of such a *synthetic resource.* Here, the component being described and managed is a high-end router, which is constructed in the usual fashion from a number of line cards, a switch fabric, and a control processor.[5] One function commonly needed from such a component might be to accept data (in the form of packets, frames, lambdas, etc.) on one port and pass the data through to another port without otherwise processing it. This function can be described by a synthetic resource called a *splice.*

The actual implementation of a splice is dependent both on the details of the component and the form of the data, but need be known only to the software implementing the component. For example, in a packet router, the allocation of a *splice* resource might involve configuring an incoming line card to remove a packet header label and allocate a switch fabric label; configuring the switch fabric to pass data arriving with that label to an output line card; and configuring an output line card to add an appropriate packet header label and send the packet to a particular output port.

In the implementation described above, the *splice* resource requires no cycles from a general-purpose data-path CPU, and might be implemented from physical elements (ie, line card label table slots) that are otherwise not visible to the outside world. In a different packet router implementation, the *splice* resource might require the use of a data-path CPU to perform the label swapping. In this case, and unlike the first, the logical *splice* resource might be coupled by a constraint with the virtualized *CPU* resource. In a third, lambda-level implementation, the

[5] A GENI programmable router will contain one or more high-speed programmable data path processors as well. However, in this particular example the existence of these processors is not relevant.

splice resource might require a completely different set of physical resources, but express the same logical concept to the requestor.[6]

It is important to note that these synthetic or logical resources become part of the agreed-on ontology of resources for the classes of component they are implemented on, and should be defined using the same level of specification mechanism and global agreement as are more traditional resources.

5 Aggregates

The GENI Architecture document [GDD-06-11] describes an abstraction called an *aggregate.* Unfortunately some ambiguity is introduced because the same name is used to refer to two fundamentally different constructs; coordination aggregates and portal aggregates. The discussion of this section refers only to coordination aggregates, hereafter called simply aggregates. Discussion of the uses of portal aggregates may be found elsewhere.

The waters are further muddied because, as discussed below, aggregation is only a subset of the functionality provided by the aggregate construct. Other uses may be equally important. However, for the moment we retain the name for consistency.

5.1 Fundamental Concepts

Recall that a component is characterized by two fundamental properties:

1. An RSpec, which consists of
    a. A list of resources that can be allocated and managed, and
    b. A set of constraints that express relationships among those resources.
2. A set of access rights, controlling which principals in the system can acquire and manage the component's resources.

An atomic or *base* component is, at high level, a component for which the resources being described are implemented directly on some supporting platform as physical, sliceable, or logical resources.

An aggregate, on the other hand, is best thought of as a transformer. It is a component that itself registers under a unique name and exposes a component interface, but actually implements its function by controlling one or more other components. In so doing, it generally alters, adds to, or transforms one or more of the basic component properties - resources, constraints, or access rights[7] - so that the new component differs from the old component(s) in some useful way.

5.2 A Two-Component Example

Figure 4 shows an example of an aggregate used to provide different levels of access to a component's internal resources for normal users, versus a limited class of "management" or

[6] Here we assume that the nature of the resources required and function to be performed can be determined by the specific ports that are being *spliced* together. Other approaches are possible.

[7] It is possible but of uncertain usefulness to discuss the *null aggregate,* which maps a component to another component (ie, one with a different name) without altering any of the properties of the mapped component.

"sophisticated" users. This situation might arise when certain configurations of a component's resources are needed for certain experiments or in special situations, but could also cause unwanted side effects or damage to the component or network.

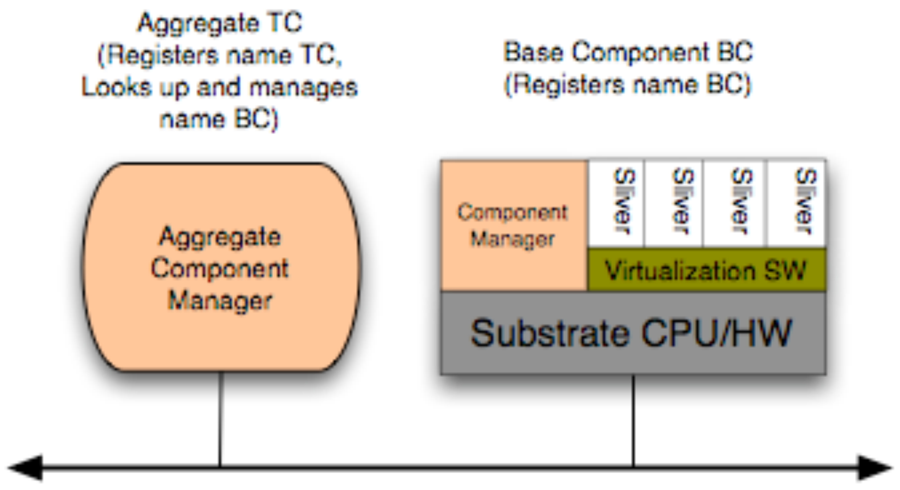

Figure 4: Aggregate Used to Transform a Component's Characteristics

In the example, the base component BC is described by an RSpec that defines no constraints on the use of component resources.[8] At the same time, the base component imposes very tight access controls. The effect is that system elements allowed to control the base component can configure and utilize the component's resources with full freedom. In return, it is expected that these system elements will have full understanding of the possible negative effects they can cause.

In the example, one of the system elements granted this full freedom is the aggregate (transformer) component TC. This component takes three actions. First, it implements a filter process that imposes additional constraints on the allocation and configuration of the base component's resources. Second, it creates a new RSpec that captures and describes those constraints. Third, it registers itself as a new (differently named) component, with relaxed access controls that permit utilization by normal system users.

When this component is utilized, it operates by first receiving requests through its own component interface and applying the filter to ensure that requests meet the desired constraints. Requests that meet the constraints are passed to the base component. Requests that do not meet the constraints generate an error.

It is easy to see that the overall effect of this structure is to create two different "access paths" to the same underlying component, but with different behaviors and constraints, and with different access rights. Similar structures can be used to create different "views" of a component, where the base component's resources are expressed using different metrics or units; in this case the aggregate (transformer) component performs the desired transformation before passing the requests on to the base component.

[8] And, of course, is implemented in a way that matches the RSpec.

A central merit of this structure is that transformer components need not be created by the same entity that created the base component. This allows any principal with access rights to the original, base component to offer a transformed or constrained view of that component to others for further use.

5.3 Synthesis and Constraint Across Multiple Components

Figure 5 shows an aggregate created to manage a set of components. This construct is most useful when some constraint needs to be enforced *between* components. A simple example might be a wireless network, where constraints are desired on the allocation of spectrum shared across all elements of the network.

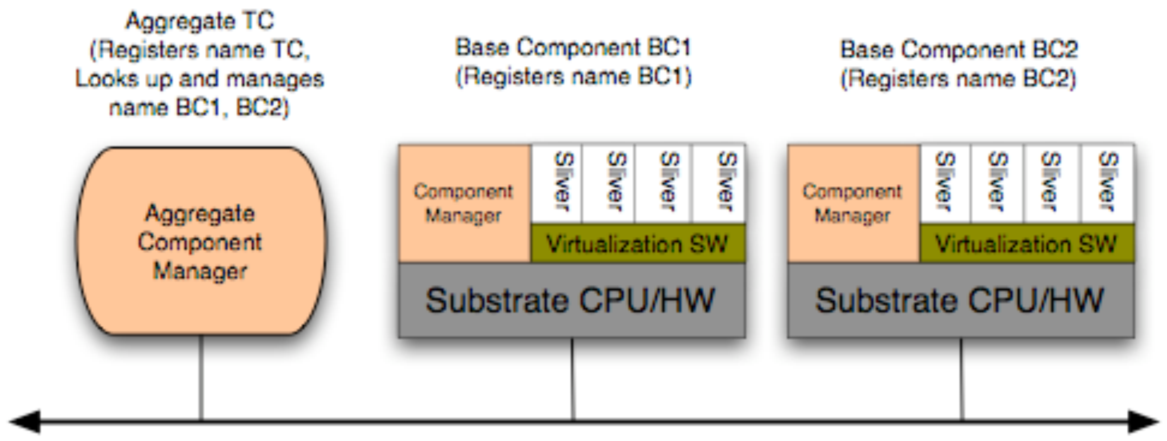
Aggregate TC (Registers name TC, Looks up and manages name BC1, BC2)
Base Component BC1 (Registers name BC1)
Base Component BC2 (Registers name BC2)
Aggregate Component Manager
Component Manager
Sliver
Sliver
Sliver
Sliver
Virtualization SW
Substrate CPU/HW
Component Manager
Sliver
Sliver
Sliver
Sliver
Virtualization SW
Substrate CPU/HW

Figure 5: Aggregate Used to Control Multiple Components

In this case the aggregate component AC is defined by a synthesized RSpec that allows the user to express resource requirements at the aggregate level. The aggregate component implementation then uses this aggregate RSpec to determine what resources are required at the individual component level, after implementing any further constraints that are required between the separate base components. These requests are then passed on to the base components, by means of the aggregate component invoking operations on the Component Interfaces exported by each base component.

Aggregate RSpecs are presently not well defined. Two broad forms are possible. First, an aggregate RSpec may simply consist of a conceptually concatenated list of RSpecs from each base component, together with any additional resource constraints and specifications needed to manage resources shared between components. A second form would express the resources of the aggregate component at a higher level suitable for configuring the aggregate, and depend on the aggregate component to transform the higher-level requests into lower-level requests appropriate for each individual base component.

Note that the concepts of this and the previous sections can be combined. A set of base components can first be captured and managed by an aggregate that imposes additional, shared constraints, but still exposes the resources of each base component at the individual level. A second, transformer aggregate could then be used to implement a separate, higher level resource description and control mechanism appropriate for the aggregated component structure.

While this text describes an aggregate component as aggregating across a set of base components, it may be useful at times for an aggregate component to act over components that themselves are aggregates.

> **Engineering Note:** In principle, it is possible to construct ever larger aggregate components, each concatenating the RSpecs from the constituent components. In practice, such aggregates represent a scalability and comprehensibility concern. It is an open issue whether such very large aggregates are feasible or useful, but it is likely in any case that large aggregates are most easily controlled in normal use by higher-level configuration services.

5.3.1 Aggregation for Convenience vs Aggregation by Necessity

In the picture of the previous section, we see that the aggregate component is acting over a set of components that are also accessible individually. That is, both the aggregate and its constituent base components have registered themselves with the component registry, and are available for use by any system entity with appropriate access rights.

In this situation, we say that the aggregate is providing a service or constraint for convenience. The aggregate is assisting with the provision of a higher level resource management constraint, and use of the aggregate may be the preferred access path to the individual components for the vast majority of users, but it is *possible* for some other system entity – a user or a higher level service – that has the appropriate access rights to access the individual components and manage the resources itself.

In virtually all cases this flexibility is highly desirable.

An alternate situation is that where the additional level of management implemented by the aggregate is *absolutely required* for correct or safe operation of the system. In this case, there is no reason for the individual components to be accessible at all.

There are three ways to accomplish this.

1. The individual components can grant access rights *only* to the aggregate component that manages them. The individual components are still registered in the component registry, and the aggregate component can still rendezvous with its constituent components through the system's normal registration and lookup procedures. Although only the aggregate component would be able to allocate and control resources on the individual components, other system services, such as those that enumerate or probe the status of components, would still be able to do so through registered, and thus locatable, interfaces.
2. The system architecture might support unregistered components. Rather than the subcomponents of the "by necessity" aggregate registering in the usual way, these components might "register" themselves only with the aggregate component. Access control would still be used to ensure that only the aggregate had access to the constituent components. However, because the constituent components are no longer registered in the usual way, other system services would not be able to locate or detect these constituent components individually.
3. The implementer may determine that the component is not really an aggregate. In the circumstance where no access to the individual sub-components of an aggregate is appropriate or required, it is possible that the component is not really an aggregate. In this circumstance, it may be more appropriate to implement the entire structure as a

base component, perhaps managed by a remote component manager, as described in Section 3

References

[GDD-06-11] Larry Peterson and John Wroclawski (Eds.), "Overview of the GENI Architecture", GENI Facility Architecture Working Group, September 2006.