Singularity Design Note

24

Self Describing Boot

Making Boot a Data-Driven Process

Singularity presents a revolutionary approach to driver and application activation. By leveraging metadata and the trusted run-time, Singularity can perfectly sandbox device drivers, fully account for hardware resource usage, and enable off-line system administration and analysis.

1. Introduction

There are precisely two methods through which a Singularity process can interact with the outside world: all communication between processes is handled through strongly typed channels, and device drivers interact with physical devices through the traditional mechanism of ports, Irqs, Dma channels, and special memory regions.

This enables a new paradigm for the configuration of the system at boot time. By leveraging the metadata capabilities of the C# and Sing# languages, Singularity can manage the boot process more fully than existing operating systems. In particular, Singularity can:

- Pre-allocate and pre-configure all resources, so that no process can dynamically acquire any resources at run-time (sandboxing)
- Predetermine requirements, so that no application, driver, or service is started unless all of its requirements are satisfied
- Reason about a system off-line, since all resources consumed or provided by a process are statically declared

This design note describes the declaration and use of metadata by the Singularity kernel and third-party code to achieve these ends.

2. Leveraging Metadata and the Trusted Runtime

The heart of a self-describing system is an awareness by the system of its components and their interaction. Additionally, there is a critical trust component. In a traditional operating system, it is possible to create manifests that declare the resource requirements of a device driver. However, the kernel must trust such a manifest to a very high degree; it can prove neither completeness nor

correctness. A driver may claim resources not stated in its manifest, and it may misuse the resources it declares. Singularity solves the first problem by leveraging application metadata. It is not certain that the latter problem of program correctness can be solved, although Singularity makes great strides toward this goal, as documented in numerous other design notes.

2.1. Declaring Resource Needs

The majority of applications begin with an initialization routine that follows a basic loop for acquiring resources and configuration. There is an implicit set (or disjunction of sets) of states that must be met in order for the application to run, and if the set is unfulfilled the application fails.

This applies equally to device drivers; the driver writer knows precisely what device signature (or set of signatures) his software is capable of serving. He also knows exactly what resources he needs to possess in order for his software to run. However, rather than annotate these requirements in a static declarative syntax, these needs are often implicit in the structure and semantics of the code.

Since Singularity is strongly typed, these requirements are limited in form. If a device driver uses an Io port, then the code must contain a declaration of an `IoPort` object. Similarly, if a driver communicates over a channel, the code declares a variable whose type derives from `Endpoint`.

By providing the driver writer with an annotation language, Singularity enables the driver writer to decorate every resource he uses in a manner that can be read off-line using standard metadata tools such as `IlDasm`, as well as custom tools within the Singularity system.

2.2. Acquiring Pre-configured Resources

Allowing the driver writer to declare his resource requirements is insufficient to guarantee safety. After all, the software may fail to decorate resources that it uses. In order to address this requirement, Singularity enforces protection through the trusted runtime.

2.2.1. Limiting Access to Io Resources

The runtime only provides access to Io resources through four objects: `IoPort`, `IoDma`, `IoIrq`, and `IoMemory`. Within the driver space there is no public constructor for any of these objects. As a result, applications can only instantiate these objects through trusted ABI calls. Thus a metadata-aware Singularity system can guarantee that the hardware resources given to a process are only and precisely those that the resource declared. Software cannot consume resources without the kernel's awareness, precisely because the runtime prevents the software from creating any Io resource except through a mechanism wholly managed by the kernel and trusted runtime.

2.2.2. Restricting Channel Access

In order for processes to communicate, there must exist a channel between them. One process must bind a channel endpoint to a public location in the namespace, and then the other process can bind his endpoint to that name. Singularity makes the public name opaque by performing all binding within the kernel. Before an application starts, the kernel pre-binds all of its endpoints according to the application's metadata, and then when the application starts it can (through the runtime) transfer exactly these endpoints into its process space. As in the case of Io resources, the application cannot create bindings, and as a result is only able to communicate along those channels declared in its metadata.

2.2.3. Device Signatures

A device driver does not simply use resources; it uses resources associated with a physical piece of hardware that is identified by a particular signature. As a result, it is necessary to group together a set of annotated resources with the signature of the device for which those resources are valid.

2.3. Example

The legacy keyboard driver illustrates the most basic scenario:

```
[DriverCategory]
[Signature("/pnp/PNP0303")]
class KeyboardResources : DriverCategoryDeclaration
{
   [IoPortRange(0, Default = 0x0060, Length = 0x01)]
   public IoPortRange port1;

   [IoPortRange(1, Default = 0x0064, Length = 0x01)]
   public IoPortRange port2;

   [IoIrqRange(2, Default = 0x01)]
   public IoIrqRange irq1;

   [ExtensionEndpoint(typeof(ExtensionContract.Exp))]
   public TRef<ExtensionContract.Exp:Start> ec;

   [ServiceEndpoint(typeof(KeyboardDeviceContract.Exp))]
   public TRef<ServiceProviderContract.Exp:Start> kbdsp;
}
```

This states that the KeyboardResource class declares all of the resources needed by our driver in order to operate correctly for devices whose signature starts with "/pnp/PNP0303." The driver expects that the Pnp enumeration of a device with this signature will create three resources, a port at 0x60, a port at 0x64, and an Irq at 0x1. In addition, for the driver to work correctly, it will need a ServiceProvider contract to register it with the namespace, and an ExtensionContract through which it can communicate with the kernel. If any of these resources cannot be pre-allocated, the driver will fail.

During the process of building manifests, the following xml will be created for this device:

```
<driverCategory>
  <device signature="/pnp/PNP0303" />
  <ioPortRange index="0" baseAddress="96" rangeLength="1" />
  <ioPortRange index="1" baseAddress="100" rangeLength="1" />
  <ioIrqRange index = "2" baseAddress="1" rangeLength="1" />
  <extension startStateId="3"
                contractName="Microsoft.Singularity.Extending.ExtensionContract"
                endpointEnd="Exp"
                assembly="Namespace.Contracts"
                version="0.0.0.0"
                culture="neutral"
                publicKeyToken="null">
    <imp>
      <inherit name="Microsoft.Singularity.Channels.Endpoint" />
      <inherit name="Microsoft.Singularity.Extending.ExtensionContract.Imp" />
    </imp>
    <exp>
```



```
      <inherit name="Microsoft.Singularity.Channels.Endpoint" />
      <inherit name="Microsoft.Singularity.Naming.ServiceContract.Exp" />
      <inherit name="Microsoft.Singularity.Extending.ExtensionContract.Exp" />
    </exp>
  </extension>
  <serviceProvider startStateId="3"
                   contractName="Microsoft.Singularity.Io.KeyboardDeviceContract"
                   endpointEnd="Exp"
                   assembly="Io.Contracts"
                   version="0.0.0.0"
                   culture="neutral"
                   publicKeyToken="null">
    <imp>
      <inherit name="Microsoft.Singularity.Channels.Endpoint" />
      <inherit name="Microsoft.Singularity.Naming.ServiceProviderContract.Imp" />
    </imp>
    <exp>
      <inherit name="Microsoft.Singularity.Channels.Endpoint" />
      <inherit name="Microsoft.Singularity.Naming.ServiceProviderContract.Exp" />
    </exp>
  </serviceProvider>
</driverCategory>
```

`<driverCategory>` declares that this app can be a driver for any device whose signature starts with the signature of any of its `<device>` children. Such a driver expects three resources from the Pnp enumeration: two ports and one Irq (in that order). These resources have an expected base and a minimum length, but can reside elsewhere or be longer than the length specified. The metadata also provides all of the information necessary for pre-binding two endpoints.

With this information, Singularity can determine whether this driver is appropriate, and determine whether all resources necessary to run the driver are available. It can do this statically, off line, or during the boot process. Singularity can also guarantee that this driver will never have another service endpoint, will never have another extension, and will only have the exact three Io resources that are pre-declared. In effect, this driver is perfectly sandboxed, and will not interact with the outside world except through these known mechanisms.

2.3.1. Syntactic Sugar

By using Compile-Time Reflection, the driver writer gains the added benefit of not writing configuration code at all. He simply declares the class exactly as above (without a constructor!) and relies on the CTR to safely build a constructor that does all interaction with the runtime to correctly populate the fields. We reduce error by declaring all resources, enabling total accounting, and decreasing the amount of boilerplate code written by developers.

2.4. Other Declarations

There are several more declarations in the metadata language that are necessary to support the full range of devices in Singularity today.

2.4.1. Fixed Resources

A device driver may require an exact port, Irq, Dma channel, or memory range. If this information is known, but the corresponding resource is never created during device enumeration, the device driver

may declare the resource using the syntax above, but with an `IoFixed` prefix replacing the `Io` prefix in the attribute name, and `Base` replacing `Default`:

```
[IoFixedPortRange(Base = 0x03d4, Length = 0x01)]
IoPortRange port1;
```

This example states that the driver will not work without a fixed `IoPort` at 0x3d4 of length 1. Supported fixed resources include `IoFixedPort`, `IoFixedIrq`, `IoFixedMemory`, and `IoFixedDma`.

2.4.2. Sharable Resources

Some drivers share resources (for example, the Ide Bus shares its resources with the controllers on it). In order to allow non-exclusive access to a resource, all parties must declare the resources as `Shared`.

```
[IoPortRange(0, Default = 0x03d4, Length = 0x01, Shared = true)]
IoPortRange port1;
```

This declaration applies to the same resources as the `Fixed` property.

2.4.3. Dma Memory Ranges

A driver may require memory in the lower 16MB of space. This can be achieved by placing a `Region` attribute on an `IoMemoryRange` declaration:

```
[MemoryRegion(MemoryRegionAttribute.AddressRange.Bytes24, 0x20000, 0x4000)]
public readonly IoMemoryRange fixedMem;
```

This creates a memory region with 24-bit addressing, aligned on a 128KB boundary, of size 16KB.

2.4.4. Read-only and Write-only Ports and Memory

It may be desirable to mark a port or memory range as having limited accessibility. This is achieved with the `AllowRead` and `AllowWrite` properties:

```
[IoPortRange(0, Default = 0x0, Length = 1, AllowWrite = false)]
IoPortRange port1;

[IoFixedPortRange(Default = 0x0a79, Length = 0x01, AllowRead = false)]
IoPortRange port2;
```

In this example, `port1` is read-only, and `port2` is write-only.

2.4.5. General Endpoint Binding

Some drivers (and almost all applications) require a client channel to communicate with another process. These are declared with the `Endpoint` attribute:

```
[Endpoint(typeof(VolumeManagerContract.Imp))]
TRef<VolumeManagerContract.Imp:Start> vmEp;
```

In this example, the driver metadata will state that the application cannot start unless it is given a pre-bound channel to the Volume Manager.

3. The System Distribution

With the needs of each application fully specified in a declarative syntax, Singularity can become a fully self-describing artifact by simply aggregating this information with system policy to create a

manifest. The overall image, consisting of this manifest and all of the parts of the Singularity system described and configured in the manifest is called a Distribution.

3.1. Rationale

The system distribution enables the following desirable features:

3.1.1. Define a Singularity System

The distribution is a compound artifact. It consists of a Kernel, Applications, Drivers, and sufficient metadata to make these individual artifacts self-describing. It also contains a manifest which declares all policy for a Singularity system.

3.1.2. Aggregate All System Policy

The distribution will contain a manifest which aggregates all of the metadata used by the system to reach the point where a shell can run. This includes the metadata of each driver that is to be started before the shell, as well the metadata for any service that must be started before the system is interactive.

3.1.3. Simulate the Installation Process

Singularity will eventually contain an installer which configures applications according to system policy. The metadata for any executable software artifact must not contain any such configuration, and thus until there is an installer, the distribution manifest will contain the policy necessary to describe how the various applications and drivers in the system are to be configured.

3.1.4. Enable Off-line analysis

The distribution and its manifest should be sufficient to enable off-line analysis. An administrator should be able to use only a description of the hardware devices and the distribution manifest, and answer questions about whether the system will boot, which drivers will initialize, which applications can run, etc.

3.1.5. Optimize the Boot Process

Declarative metadata-driven boot is a limited form of logical resolution. In order to speed the boot process, the manifest may optionally provide policy that establishes a total order or otherwise optimizes the process by which the satisfiable requirements of driver software are converted into activated software processes.

3.2. Static System Policy

The distribution is itself a declarative policy. However, it is also the output of a declarative process which resolves four key inputs:

3.2.1. The File Listing

This listing identifies every application and manifest that ought to be included in a Singularity distribution. It will ultimately be determined based on a declarative policy, but for the time being is simply a listing of every app and manifest in the Distro build tree, as well as the kernel, the system manifest, and any files in Distro's Files subfolder.

3.2.2. Naming Conventions

The naming conventions tree is a static policy that is copied verbatim into the distribution manifest. It specifies a one-to-one mapping between contract types and names in the namespace:


```
<namingConventions>
  <name contract="Microsoft.Singularity.Io.DiskDeviceContract" nsName="/dev/disk"
        allowMultiple="True" limit="3" />
  <name contract="Microsoft.Singularity.Io.Network.NicDeviceContract" nsName="/dev/nic"
        allowMultiple="True" />
  <name contract="Microsoft.Singularity.Io.VolumeManagerContract" nsName="/dev/volmgr" />
  <name contract="Microsoft.Singularity.Io.KeyboardDeviceContract"
        nsName="/dev/keyboard" />
  <name contract="HalConsole" nsName="/dev/conout" />
  <name contract="Microsoft.Singularity.Io.SoundDeviceContract" nsName="/dev/audio" />
  <name contract="Microsoft.Singularity.Io.VideoDeviceContract" nsName="/dev/video" />
  <name contract="Microsoft.Singularity.Io.ConsoleDeviceContract"
        nsName="/dev/video-text" />
 </namingConventions>
```

This policy specifies 3 types of behavior. In the case of the keyboard contract, the map states that the first process to start who declares that he provides a `KeyboardDeviceContract` will be given the public name `/dev/keyboard` and all subsequent apps who try to start a `ServiceProvider` contract to listen for a `KeyboardDeviceContract` will not be allowed to start.

In the case of the `NicDeviceContract`, the policy states that multiple applications can be create `ServiceProvider` channels that listen for `NicDeviceContract` connections, and that their names will be `/dev/nic0`, `/dev/nic1`, etc. (this is achieved by the `allowMultiple` attribute).

The `DiskDeviceContract` extends the previous example by adding the attribute `limit`. It declares that no more than 3 `ServiceProvider` channels can be bound into the namespace who listen for `DiskDeviceContract` connections.

When a driver or application specifies in its metadata that it wishes to create a `ServiceProvider` contract, its metadata decoration must give the type of the contract that the ServiceProvider ultimately provides. The following examples demonstrate the difference. The second example is incorrect, as it does not declare the type of the listened-for contract.

```
[ServiceEndpoint(typeof(SoundDeviceContract.Exp))]
public TRef<ServiceProviderContract.Exp:Start> audioSp;
[ServiceEndpoint(typeof(ServiceProviderContract.Exp))] // INCORRECT!!!
public TRef<ServiceProviderContract.Exp:Start> invalidSp;
```

If the contract type specified in the metadata does not appear in the namingConventions, the application will never be authorized to run. This ensures that the system policy is the sole determinant of namespace names. Thus the second example will result in a driver that cannot start unless `ServiceProvider` is listed as the `contract` for a `<name>` entry in the name map.

3.2.3. The Driver Registry

When Singularity includes a system installer, then each device driver that runs in a separate process will be installed through such a mechanism. This process will validate driver requirements and transform them into a format easy to parse at boot time.

3.2.3.1. Adding and Removing Entries

This procedure is simulated through an imperative script, encoded as Xml and included in the system policy used to build a distribution:

```
<driverRegistryConfig>
```



```
<include>
 <device signature="" />
 <driver identity="" />
</include>

<remove>
 <device signature="/pci/06/01/10de/0050" />
 <driver identity="S3Trio64" />
</remove>

<include>
 <device signature="/pci/06/01/10de/0050" />
</include>

<ordering>
 <provides signature="/pci/06/01/10de/0050" name="NvPciLpc" />
 <follows signature="/pci/02/00" name="NvPciLpc" />
 <follows signature="/pci/01/01/10de" name="NvPciLpc" />
</ordering>

</driverRegistryConfig>
```

The imperative policy is applied in order. The distribution build tool first analyzes the entire file list to find all apps whose manifest states that there is a driver role. It then transforms their `<driverCategory>` tags into a format optimized for system boot. It then applies `<include>` and `<remove>` policies to prune and augment the list of legal drivers.

All comparisons are achieved through prefix match. Thus in the above example, all legal entries are copied in, then any driver who serves a device named `/pci/06/01/10de/0050` is removed, then any driver whose name is `S3Trio64` is removed, and lastly any drivers who can serve devices named `/pci/06/01/10de/0050` are reinserted.

3.2.3.2. Creating Total Order from Partial Order

In the previous example, there is also an <ordering> tag. This allows the system to create a declarative policy for specifying a more restrictive partial order. The order is not firm; if no driver is activated to provide a name, then the drivers who follow that name will still run. However, this lets a system administrator specify the order in which drivers are initialized.

3.2.3.3. Transforming Driver Registry Entries

In order to optimize the boot process, the distribution manifest contains an optimized version of the manifests of every driver added by the driverRegistryConfig script. The optimized metadata achieves three goals:

- `driverCategory` entries are cloned for each validsignature
- Resouces marked as Fixed are moved to a separate tree
- Empty trees are inserted to ensure a fixed structure

Two examples illustrate the result. First, we see that a driver's information is reformatted into a new structure (endpoint entries are abbreviated):


```
<driverCategory>

  <device signature="/pnp/PNPB003" />

  <ioPortRange index=”0” baseAddress="544" rangeLength="16" />
  <ioPortRange index=”1” baseAddress="896" rangeLength="16" />
  <ioIrqRange index=”2” baseAddress="5" rangeLength="1" />
  <ioDmaRange index=”3” baseAddress="1" rangeLength="1" />
  <ioDmaRange index=”4” baseAddress="5" rangeLength="1" />
  <region addressing="24" alignment="131072" size="16384" />

   <extension> … </extension>
   <serviceProvider> … </serviceProvider>

</driverCategory>
```

Becomes

```
<driver identity="Sb16" signature="/pnp/PNPB003" image="Sb16.x86">
  <endpoints>
    <extension> …</extension>
    <serviceProvider> … </serviceProvider>
  </endpoints>

  <fixedResources count="1">
    <region addressing="24" alignment="131072" size="16384" />
  </fixedResources>

  <dynamicResources count="5">
    <ioPortRange index="0" baseAddress="544" rangeLength="16" />
    <ioPortRange index="1" baseAddress="896" rangeLength="16" />
    <ioIrqRange index="2" baseAddress="5" rangeLength="1" />
    <ioDmaRange index="3" baseAddress="1" rangeLength="1" />
    <ioDmaRange index="4" baseAddress="5" rangeLength="1" />
  </dynamicResources>
</driver>
```

Secondly, note that the IdeBus entry is cloned for each signature, and empty endpoint and fixedResource trees are created:

```
<driverCategory>
  <device signature="/pci/01/01/8086/7111" />
  <device signature="/pci/01/01/10de/0053" />
  <ioPortRange index="0" baseAddress="65440" rangeLength="16" Shared="True" />
</driverCategory>
```

Becomes

```
  <driver identity="internal" signature="/pci/01/01/8086/7111" image="">
    <endpoints />
    <fixedResources count="0" />
    <dynamicResources count="1">
      <ioPortRange index="0" baseAddress="65440" rangeLength="16" Shared="True" />
    </dynamicResources>
  </driver>

  <driver identity="internal" signature="/pci/01/01/10de/0053" image="">
```



```
  <endpoints />
  <fixedResources count="0" />
  <dynamicResources count="1">
   <ioPortRange index="0" baseAddress="65440" rangeLength="16" Shared="True" />
  </dynamicResources>
  <follows name="NvPciLpc" />
 </driver>
```

3.2.4. The /init Namespace Configuration

The /init namespace configuration consists of two pieces. The first is a static configuration of system security policy, and is not described in this document. The namespace also includes as a child tree the list of all the files included in a distribution. This information is used to name each application in the distribution, and to associate manifests with applications. An abbreviated version of the configuration follows:

```
<files>

  <file distroName="/Singularity/Kernel/kernel.dmp" name="kernel.dmp" index="0" />
  <file distroName="/Singularity/metadata.xml" name="metadata.xml" index="1" />
  <file distroName="/Singularity/Applications/Attr.x86" name="Attr.x86" index="2"
        executable="True" manifestIndex="3" />
  <file distroName="/Singularity/Applications/Attr.manifest" name="Attr.manifest" index="3"
        manifest="True" />
   …
 </files>
```

This configuration, created from the file listing, enables the complete configuration of the namespace so that all applications are associated with their manifests and configured correctly. The tags are relatively self-evident, and somewhat redundant.

3.3. SingBoot.ini

In order for the system to boot, the Singularity Boot Loader must request files individually from the boot server (which may be a hard disk, CdRom, Usb memory device, or Pxe network server). Rather than build a full Xml parser into the boot loader, the distribution build process creates a simplified .ini file from the `<files>` listing. This file is then requested by the boot loader and used to guide the file transfer process.

4. Resolving Dependencies at Boot Time

When Singularity boots, it possesses a full set of all the devices that it may legally activate, along with their metadata. The entire driver initialization process is thus simply a process of matching the hardware signatures of physical devices to the most appropriate registered driver. This is accomplished through a basic initialization step with four classifications of drivers:

4.1. Registering Drivers

The registered driver listing is simply a list of all drivers whose metadata appears in the distribution manifest. For each such driver, Singularity uses metadata to maintain the following information:

- Driver Signature Prefix – names all devices that this driver can serve
- Driver Metadata – an exact copy of the `driverRegistry` entry for this driver/signature pair

- Public Name – every registered driver is named uniquely in the `/hardware/drivers` namespace tree. Thus the nvmac driver for Nvidia network cards would be given the name `/hardware/drivers/nvmac`.

- Executable Image Name or Factory – most drivers are separate processes, in which case this holds the name of the executable to run in order to activate this driver. In the case of a few drivers that do not run as processes (for example the pci bus), the factory provides a method for creating the driver object.

4.2. Creating Device Instances

Whenever Singularity enumerates a bus, it enters any devices it finds into the driver instance listing. This list consists of four fields:

- Location – the location is the exact, unique name of the physical slot where the device exists, for example `/pnp0/107/0000/0007/0001`.

- Device Signature – This is the full name of the device that is in the particular location. For example, if there were three identical Nvidia network cards in the machine, each would have a different location, but all of the instances would have the same signature, `/pci/02/00/10de/0056/a3/1462/7125`.

- Public Name – every instance of every device is uniquely identified by a tree in the `/hardware/devices` namespace. The public name consists of the device signature and instance. For example, if there were three identical Nvidia network cards in the system, the first such device would have the public name `/hardware/devices/pci/02/00/10de/0056/a3/1462/7125/instance0`.

- Dynamic Hardware Resource Config – enumeration of a plug-and-play bus lists the location, device signature, and also the set of resources dynamically assigned to the device. This list of resources completes the definition of a driver instance.

4.3. Associating Drivers with Device Instances

Association is the process of choosing the best driver registration for a particular device instance. This is accomplished by prefix-match between the device signature and a driver signature.

When such a registration is chosen, the device instance node is given copies of all fields in the registration. In addition, at this stage the metadata is parsed and all fixed resource requirements are determined. The factory or device image is used to create an executable `IDevice` object. At this point, Singularity knows the exact resources assigned to a device, as well as the full information needed to start driver code for that device.

In a traditional operating system, this is as much information as can be known, and the device would be activated. However, Singularity also possesses the full metadata of the device driver, and can perform pre-activation analysis to ensure that the driver's resource needs are satisfiable. As a result, Singularity separates association from activation.

4.4. Activating Driver Code

In order to activate a driver, Singularity analyzes the driver metadata to ensure that the device resources fit the driver expectations. Singularity also knows every resource assigned to every driver, and can ensure no resource conflicts exist. Furthermore, Singularity knows the full set of endpoints that a driver needs to communicate with other processes, and can ensure that no driver starts unless its communication endpoints are successfully pre-bound.

Singularity always starts activatable bus drivers before other activatable drivers. The rules for determining if a driver can be activated are detailed below:

4.4.1. Obeying Ordering Policy

The driver metadata may include policy to induce a particular order for driver activation. Singularity will defer the activation of associated drivers as necessary to achieve this ordering. However, if a driver is deferred and its ordering constraint cannot be satisfied (for example, because it follows a driver for a device that does not exist), then the driver will be activated after all drivers whose ordering constraints are met.

4.4.2. Checking Endpoint Availability

Singularity checks the `endpoint` tree of a device driver's metadata to guarantee that all inter-process communication needs are satisfiable.

Every `ServiceProvider` channel required by a device must pass two verification steps. First, the system's `namingConventions` policy must explicitly provide a public name that can be assigned to the requested channel. Secondly, the set of existing activated drivers must not have exhausted the limits of the `namingConventions` policy.

Additionally, every channel deriving from `ServiceContract` is checked to ensure that a corresponding `ServiceProvider` has been started. Since Singularity pre-binds all endpoints, this is a necessary requirement: if the kernel cannot attach the driver to a process it requires, the driver should not start.

4.4.3. Preventing Resource Sharing

Unlike existing systems, Singularity has a full knowledge of all hardware resources that a driver will use before it initializes the driver. As a result, the system can guarantee that no drivers share a physical hardware resource unless all parties agree to share the resource (via the Shared tag in the metadata). Thus Singularity will deny a driver the opportunity to run if any of the resources it requires (fixed or dynamic) are claimed by another driver. Upon activation, every resource used by a driver is recorded to ensure conflict-free execution.

4.4.4. Matching Pnp Resources to Driver Metadata

In addition, drivers specify the shape and appearance of the dynamic resource set they expect device enumeration to give for a particular device signature. Since this information is encoded in the metadata, the validation step will reject drivers who have been associated with devices whose dynamic resource configuration does not match expectations.

This test compares ordered subsets. Consider the following metadata declaration:

```
<dynamicResources count="2">
  <ioPortRange index="0" baseAddress="64" rangeLength="4" />
  <ioIrqRange index="1" baseAddress="0" rangeLength="1" />
```



```
</dynamicResources>
```

Each of the following dynamic configuration packets is acceptable:

```
<dynamicResources count="2" rem=”this is an exact match”>
  <ioPortRange index="0" baseAddress="64" rangeLength="4" />
  <ioIrqRange index="1" baseAddress="0" rangeLength="1" />
</dynamicResources>

<dynamicResources count="2">
  <ioPortRange index="0" baseAddress="64" rangeLength="8"
                  rem=”the range can be longer than the metadata specifies” />
  <ioIrqRange index="1" baseAddress="0" rangeLength="1" />
</dynamicResources>

<dynamicResources count="2">
  <ioPortRange index="0" baseAddress="64" rangeLength="4" />
  <ioIrqRange index="1" baseAddress="8" rangeLength="1"
                 rem=”base values aren’t checked for dynamic resources” />
</dynamicResources>

<dynamicResources count="3">
  <ioPortRange index="0" baseAddress="64" rangeLength="4" />
  <ioIrqRange index="1" baseAddress="0" rangeLength="1" />
  <ioPortRange index="2" baseAddress="0x3c0" rangeLength="4"
                    rem=”extra resources are ok if they appear at the end of the list” />
</dynamicResources>
```


These examples demonstrate pairings that would result in a driver not activating:

```
    <dynamicResources count="3">
      <ioPortRange index="0" baseAddress="64" rangeLength="4" />
      <ioPortRange index="1" baseAddress="68" rangeLength="4" rem="ioIrqRange expected"
/>
      <ioIrqRange index="2" baseAddress="0" rangeLength="1" />
    </dynamicResources>

    <dynamicResources count="2">
      <ioPortRange index="0" baseAddress="64" rangeLength="2" rem="length is not long
enough"/>
      <ioIrqRange index="1" baseAddress="0" rangeLength="1" />
    </dynamicResources>

    <dynamicResources count="1" rem="device has fewer resources than metadata">
      <ioPortRange index="0" baseAddress="64" rangeLength="4" />
    </dynamicResources>
```

5. Impact

The implications of Singularity's innovative use of metadata to sandbox drivers and increase stability are tremendous. This section highlights three key results:

5.1. Stability

Singularity's driver model is more stable than other operating systems. Since a driver cannot access kernel state, an errant driver cannot affect kernel stability. Furthermore, the ability of a device driver to affect another process non-deterministically is reduced significantly: there is only one possible channel for a device driver to affect another process, as described in section 5.3.

5.2. Off-line Analysis

Since the entire boot process is based on a simple data-driven algorithm, it is now possible to create tools to reason about a system off-line, given only the metadata for a system and a description of the hardware. In particular, the following questions can now be statically answered:

- Will this system boot?
- Will driver X start?
- Does application Y have everything it needs to run?
- Can I add device Z to my system without installing new software?
- What happens if I remove device Q or driver Z from the system?

5.3. A New Trust Model

There is exactly one remaining hole in the Singularity driver system. With the metadata-driven design, no driver can ever claim resources secretly. However, it is still impossible to verify that a driver will use the resources it requests correctly. Driver code may use resources inappropriately, and fail as a result. It also can declare a resource as shared, but not share it correctly, in which case it is possible to affect any other drivers who also share that resource.

Nonetheless, Singularity presents a new trust model for drivers. In traditional operating systems, the trust boundary is coarse – the kernel must trust the driver completely, and must load the driver into the kernel's memory space. In essence, the kernel must trust everything about the driver.

In contrast, Singularity must only trust that a program does not contain bugs. The system does not trust that drivers do not affect kernel memory – the runtime guarantees it! Likewise, the system does not trust that drivers only use the resources they declare up front, because this, too, is guaranteed by the runtime and metadata.

6. The /hardware Namespace

To aid in system administration, the device driver initialization process includes the creation of a `/hardware` namespace. This has been alluded to in previous sections. Here it is explicitly described.

6.1. Goals

The `/hardware` namespace achieves two goals:

- Establish a single site for viewing all device driver configuration at run time
- Transform the public names of `ServiceProvider` contracts into opaque links

6.2. Structure

6.2.1. /hardware/locations

The locations tree lists all buses, and every location on each bus. Each location is represented as a directory tree, in which there resides a symbolic link to the instance of the device that resides in this location.

6.2.2. /hardware/registrations

The registrations tree lists every driver that is registered in the system. The registration exists as a directory tree built from the signature prefix served by the driver. Within this tree, there is one symbolic link pointing to the driver that is registered for the corresponding signature prefix.

6.2.3. /hardware/devices

The devices tree contains an entry for each instance of each physical device in the system. The signature of a device (as determined by device enumeration) is used to create a directory tree. Within the tree, each instance of the device is a separate subtree with symbolic links pointing to corresponding entries in the locations and drivers trees to show how the device instance was found, as well as how it was associated and activated.

6.2.4. /hardware/drivers

The drivers tree lists every registered driver, with a subtree for each instantiation of that driver. The names here based on the namespace name of the driver class itself. For a particular driver, the tree consists of a symlink pointing to the executable image that the driver uses. It also contains a subtree for each instance of the driver. This subtree holds links to the corresponding device instance, and back up to the executable image. Also contained in this space are the true bindings for all `ServiceProvider` contracts created for each instance of the driver.

6.2.5. /dev

The `/dev` namespace is a public folder containing symbolic links to the `ServiceProvider` endpoints residing in the `/drivers` subtree. In this manner, an application can be bound to a public name without having any information about the true name of the driver providing a contract implementation.

6.3. Example

The following example demonstrates the configuration of the `/hardware` namespace for a system with two nvmac NICs and one tulip NIC. Note that the names in /dev are opaque, and that there is no need for a ".." link in order for a driver to acquire all pertinent information if given only a link to its driver instance.

```
/
├──hardware
│   ├──locations
│   │   └──pnp0
│   │       └──109
│   │           └──0000
│   │              ├──0007
│   │              │   └──0000
│   │              │       └──device =>
/hardware/devices/pci/02/00/1011/0009/20/00a0/2114/instance0
│   │              ├──0008
│   │              │   └──0000
│   │              │       └──device =>
/hardware/devices/pci/02/00/10de/0056/a3/1462/7125/instance0
│   │              └──000a
│   │                  └──0000
│   │                      └──device =>
/hardware/devices/pci/02/00/10de/0056/a3/1462/7125/instance1
│   ├──registrations
│   │   └──pci
│   │       └──02
│   │           └──00
│   │              ├──1011
│   │              │   └──0009
│   │              │       └──Driver => /hardware/drivers/microsoft/tulip
│   │              └──10de
│   │                  └──0009
│   │                      └──Driver => /hardware/drivers/nvidia/nvmac
│   ├──devices
│   │   └──pci
│   │       └──02
│   │           └──00
│   │              ├──1011
│   │              │   └──0009
│   │              │       └──20
│   │              │          └──0a00
│   │              │              └──2114
│   │              │                 └──instance0
│   │              │                    ├──location =>
/hardware/locations/pnp0/109/0000/0007/0000
│   │              │                    └──driver =>
/hardware/drivers/microsoft/tulip/instance0
│   │              └──10de
│   │                  └──0056
│   │                      └──a3
│   │                         └──1462
│   │                            └──7125
│   │                               ├──instance0
│   │                               │  ├──location =>
/hardware/locations/pnp0/109/0000/0008/0000
│   │                               │  └──driver => /hardware/drivers/nvidia/nvmac/instance0
│   │                               └──instance1
```



```
|  |                              ├──location =
/hardware/locations/pnp0/109/0000/000a/0000
|  |                              └──driver => /hardware/drivers/nvidia/nvmac/instance1
|  └──drivers
|     ├──nvidia
|     |  └──nvmac
|     |     ├──image
|     |     ├──instance0
|     |     |  ├──image => /hardware/drivers/nvidia/nvmac/image
|     |     |  ├──endpoint0
|     |     |  └──device =
/hardware/devices/pci/02/00/10de/0056/a3/1462/7125/instance0
|     |     └──instance1
|     |        ├──image => /hardware/drivers/nvidia/nvmac/image
|     |        ├──endpoint0
|     |        └──device =>/hardware/devices/pci/02/00/10de/0056/a3/1462/7125/instance1
|     └──microsoft
|        └──tulip
|           ├──image
|           └──instance0
|              ├──image => /hardware/drivers/microsoft/tulip/image
|              ├──endpoint0
|              └──device =
/hardware/devices/pci/02/00/1011/0009/20/00a0/2114/instance0
└──dev
   ├──nic0 => /hardware/drivers/microsoft/tulip/instance0/endpoint
   ├──nic1 => /hardware/drivers/nvidia/nvmac/instance0/endpoint
   └──nic2 => /hardware/drivers/nvidia/nvmac/instance1/endpoint
```

7. Endpoint Configuration Details

The manner in which endpoints are bound is unique: the kernel does not have reflection, and cannot create the exact type of an arbitrary endpoint. Nonetheless, by using metadata the kernel can compute the exact hash and size of the type, register it, and pass it to a client process who then casts it to the appropriate object within its namespace.

7.1. Computing the SystemType from Metadata

In order to create an endpoint, one must first know the exact system type for the endpoint. For an endpoint of type `Microsoft.Singularity.Io.VolumeManagerContract.Exp`, the type is computed from three values: the full name of the endpoint, the type of the endpoint's immediate ancestor, and the hash of the full name of the endpoint. Thus by including in a driver's metadata the full ancestry of a type (extending back to the known `SystemType` for `Microsoft.Singularity.Channels.Endpoint`), as well as the full name of the type, Singularity can create a full `SystemType` registration that is valid for both the kernel and the driver.

7.2. Using Metadata to Set the Endpoint State

Since the metadata declaration is always attached to a strongly typed field, the exact start state of the contract can be determined as an integer value statically during the manifest build process.

7.3. Inferring Endpoint Sizes

At this time, the metadata does not declare the exact size of endpoints. Instead, Singularity conservatively estimates the size of the endpoints and allocates a structure that is large enough to express all existing contracts.

