Singularity
Design Note
27

The Singularity Directory Service (SDS)

This document contains a description of the singularity Namespace and it associated constructs. The contracts described in this document include:

Service Contract

ServiceProvider Contract

DirectoryService Contract

Notification Contract

Copyright © Microsoft Corporation. All Rights Reserved.

Copyright © Microsoft Corporation. All Rights Reserved.

1. Overview

Singularity's extension model consists of Software Isolated Processes (SIPs) communicating over channels. Applications, services, and device drivers all adhere to this extension model. In order for a SIP to communicate externally it must have a way to name and bind to other SIPs. If a SIP wants to be known, it registers itself with the Singularity Directory Services (**SDS**). If a client wants to communicate with a SIP it must use the SDS to find and bind to that service. The binding process involves traversing nodes in the SDS looking for the named entity, and connecting the requestor to the named service. In addition to binding and registration the SDS supplies services for organizing names and notifying clients of the creation, deletion or modification of names contained therein.

The SDS provides a local namespace for a given Singularity installation. Conceptually, the namespace described by the SDS consists of directories, symbolic links, and services. Directories are used to organize the names in the namespace. Symbolic links are used to give multiple names to an entity. A service, in this context, is simply a means to connect an endpoint to the registered service.

The system-local SDS is composed as a federated tree of Directory Service Providers (**DSP**). A DSP is a SIP supporting the DirectoryServiceContract contract. Two examples of DSPs in Singularity are: the kernel supplying the local root, and the Boxwood File System. Both of these DSPs have rich internal structure; both support symbolic links and files. While at the time of this writing there are no remote DSPs the SDS has been architected to support them.

The system local SDS namespace is presented as a hierarchical tree rooted at "/". Paths within this tree are alphanumeric names delimited by "/". Some example pathnames are "/dev/nic0", "/tcp/128.0.0.1/80", and "/fs/files/C/Work/proposal.doc". As stated above, DSPs can be registered within the local tree and handle all directory service requests for names found below the point at which they are registered. This functionality is similar to CNAMES in DNS (and mount points in UNIX file systems). When operations in the SDS would cause the traversal of a DSP, the client is notified of the boundary. With the information returned, it is up to the client to bind to the DSP servicing the sub hierarchy and then continue the original operation. Symbolic links are handled in a similar manner.

2. Services

To be a locatable service a SIP must:

1) Support a contract derived from the ServiceContract over which it will communicate with clients.
2) Support the ServiceProviderContract over which it will communicate with the SDS.

Both of these contracts are explained below.

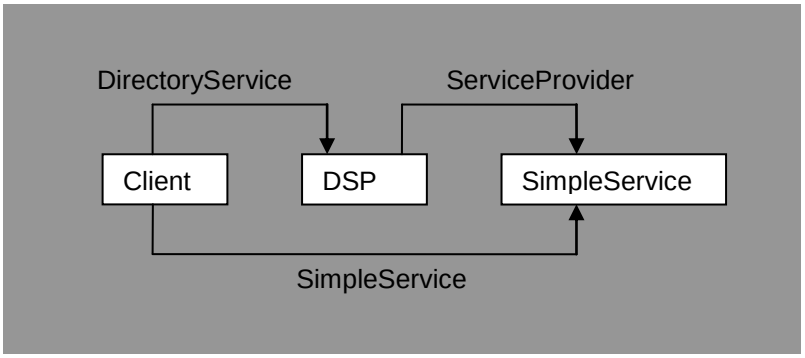


Figure 1

In the Figure 1 above there are three SIPs: the client, a Directory Service Provider and, a service called SimpleService. The client communicates with DSP, over a channel conforming to the DirectoryService

Copyright © Microsoft Corporation. All Rights Reserved.

contract, sending a request to bind to “SimpleService”. The DSP, when requested to bind to “SimpleService” must communicate with the actual service over a channel conforming to the ServiceProvider contract. The DSP sends a connect message to the service and waits for a response. If the request is accepted a channel is established between the client and SimpleService conforming to the SimpleService contract. If not, the client is informed why not.

A cursory inspection of the arguments to the bind, register and connect messages will help motivate the following subsections. They are:

```
Register(char[]! in ExHeap path,ServiceProviderContract.Imp:Start! imp);
Bind(char[]! nn ExHeap path, ServiceContract.Exp:Start! exp);
Connect(ServiceContract.Exp:Start! exp);
```

Before the client was able find SimpleService it was necessary for SimpleService to register with the DSP. This was accomplished by sending a **Register** message to a DSP node supplying the name under which to register the service and an import endpoint with which to send the service connect messages. All services that register with SDS must supply an endpoint conforming to the ServiceProviderContract.

As part of issuing a bind request the client must pass in an export endpoint conforming to the ServiceContract. In the DSP, upon discovering the named service, the DSP in turn, passes along the endpoint, via the connect message, to the service. In both the bind and connect requests the contract type specified is the same. Once the service has the export end of the channel it can then use this to communicate via the SimpleService Protocol.

2.1. Service Contract

The service contract defines the base abstract contract for all actual service contracts. It is by extension of this base contract that all usable contracts are defined. In the definition of the ServiceContract below you will notice the state **Start** is defined. This is the initial state for all services.

```
public contract ServiceContract {
   out message ContractNotSupported();
   state Start: one{
       ContractNotSupported! -> Done;
   }
   state Done: one {}
}
```

In practice, usable service contracts inherit from the base ServiceContract contract and override the Start state with their own state machine. As of this writing there are several restrictions on contract inheritance. In order to inherit from a base contract one must use the override clause to “take over” the initial state of the contract; no messages or states survive the override. If an overridden state exists on the export (server) end of a channel, that end is required to send a message before it can go into a receive loop. In the simple service contract below the Start state is overridden, a Success message is sent first and then the ready state is entered where it can loop receiving Hi messages.

Copyright © Microsoft Corporation. All Rights Reserved.

```
public contract SimpleServiceContract: ServiceContract
{
    in     message Hi(char[]! in ExHeap name);
    out    message AckHi(char[]! in ExHeap name);

    out message Success();
    override state Start: one { Success! -> Ready; }

    state Ready: one {
      Hi? -> AckHi! -> Ready;
    }
```

2.2. Service Provider Contract

```
contract ServiceProviderContract
{
    in  message Connect(ServiceContract.Exp:Start! exp);
    out message AckConnect();
    out message NackConnect(ServiceContract.Exp:Start exp, ErrorCode code);
    // return the unconnected endpoint if possible.

    state Start: Connect? ->  Ack;
    state Ack: one {
        AckConnect!  -> Start;
        NackConnect! -> Start;
    }
}
```

In order for a service to register with a DSP it must support the ServiceProvider contract. When a bind is requested on a service, the DSP sends a **Connect** message over that channel to the service. The argument to a **Connect** message is the export endpoint to any service derived from the ServiceContract. Upon receiving the connect message, the service decides to either accept or reject the connection. If the service decides to accept the connection the server will

- Send an acknowledgement to the DSP,
- Send a success message over the appropriately casted service endpoint to the client and,
- Enter a receive loop.

The code fragment below is from the SimpleService service. Notice the **switch receive** statement embedded in the **for** loop. This statement contains two case statements, one for receiving **Connect** messages from DSPs on a ServiceProvider channel, and the other for receiving Hi messages from clients. The candidate argument of the connect message is, as explained above, an export endpoint for channel conforming to the ServiceContract. Upon receipt of the message the service attempts to cast the endpoint to a SimpleServiceContract endpoint using the “as” language operator. If the cast is successful the service knows that the endpoint passed in will adhere to the message types and ordering defined in the SimpleServiceContract. After choosing to accept a connection the service contacts the client, sending a success message, over the channel passed in and is ready to receive inbound messages over that channel. In the second case statement the service, upon receiving a “Hi” message from the client, responds with acknowledgement.

Copyright © Microsoft Corporation. All Rights Reserved.

```
            // get the endpoint and set up the main switch receive
             ServiceProviderContract.Exp sp =
                SimpleServiceResources.Values.sp.Acquire();

             // create a set of all client endpoints connected to the Simple
             // service
             ESet<SimpleServiceContract.Exp:Ready> clients
                 = new ESet<SimpleServiceContract.Exp:Ready>();

             for (bool run = true; run;) {
                 switch receive {
                     // Listen for new connections
                     case sp.Connect(candidate) in clients:
                         SimpleServiceContract.Exp newClient =
                                 candidate as SimpleServiceContract.Exp;
                         if (newClient != null) {
                             newClient.SendSuccess();
                             clients.Add(newClient);
                              sp.SendAckConnect();
                         }
                         else {
                             sp.SendNackConnect(candidate);
                         }
                         break;

                             // Listen for client requests
                     case ep.RecvHi(data) in clients:
                         ep.SendAckHi(data);
                        break;
                 }
             }
```

3. The DirectoryService Contract

The DirectoryServiceContract is for clients and DSPs to communicate with. The client uses the import end to make requests to the Directory Service; the DSPs hold various export endpoints to service those requests. The DirectoryServiceContract provides operations to bind, find, and notify users of entities in the SDS. Additionally it has operation to register entities, query and manipulate directories (internal structure), symbolic links, ACLs and files.

Figure 2 will be referred to extensively throughout this section. Contained in Figure 2 are four SIPs: the kernel, the FS, service2 and the client. The kernel and the FS are DSPs denoted by the circle found within their SIP. The kernel, the FS and Service2 are all services. FS is registered as "/FS". Service2 is registered as "/FS/c/d/Service2". The nodes "a" and "b" are directories internal to the kernel DSP. Nodes c and d are directories internal to the FS DSP. F2 and File1 are files. Link1 is a relative symbolic link. In the kernel DSP FS is service leaf node and Link1 is a symbolic link leaf node. In the FS DSP Service2 is a service leaf node and File1 is a file leaf node.

Copyright © Microsoft Corporation. All Rights Reserved.

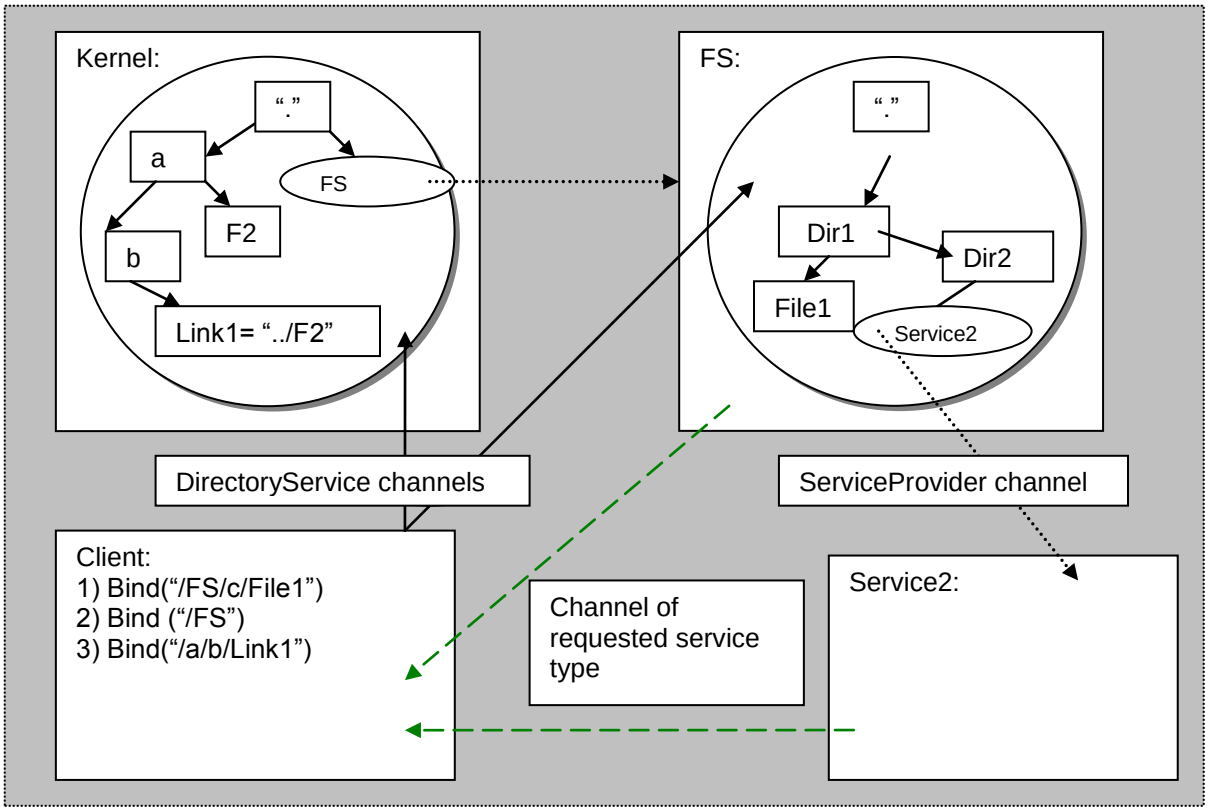


Figure 2.

The remaining sections are structured as follows. Section 3.1 discussion the fundamental operations bind, find, notify and register, all of which involve navigating and traversing the names in the SDS. Section 3.2 discusses symbolic links, Section 3.3 discusses directory manipulation, Section 3.4 discusses file manipulation and Section 3.5 discusses ACL manipulations.

3.1. Traversing Operations

The operations Register, Bind, Enumerate, Notify all involve the path traversal of entities encountered in the directory services. Recall that the local directory service is comprised of a root DSP under which all locally-known names including other DSPs can register. In all of these operations a client passes in as an argument the path of the entity of interest.

Let **P** denote the full path specified by the initial request made by some client **C** intended for some DSP **D**. As every node is traversed more and more of P is consumed. At any point in the process let **prefix(P**) denote that part of P which has already been processed. Let **suffix(P)** denote the remainder of the path left to traverse. Finally let **V** denote the value contained in a symbolic link. As an example let P=”/a/b/c/d/e. If node “c” is being processed then prefix(P) = “/a/b/c” and suffix(P)=”/d/e”.

3.1.1. Reparse points

A DSP only knows how to parse structure within its own container. Any names outside of its container must be processed elsewhere. When a DSP registered within a DSP is encountered, a reparse operation is necessary and it is up to the client, more specifically the client library, to take action to continue the traversal. It the Figure 2 the FS DSP is registered within the kernel DSP at “/fs”. When performing a traversing operation with P=”/fs/c/file1” the kernel DSP encountering the FS service leaf node send a NakReparse message to the client. The client, upon receiving the NakReparse, attempts to bind the FS directly with the DirectoryService contract using the path return in the reparse message. The client then continues the original search overt this new connection. This sequence will take place for every DSP encountered when traversing the original path. Each directory operation has a specific reparse message but they all have the form:

Copyright © Microsoft Corporation. All Rights Reserved.

```
        out    message NakReparse(char[]! in ExHeap prefix,
                                  char[]! in ExHeap suffix,
                                  bool linkFound,
                                  SomeContractType ep );
```

Where **SomeContractType** is specific to the operation and will be discussed below in the individual operations. The prefix and suffix arguments are the same in all reparse messages and represent prefix(P) and suffix(P) respectively. The prefix argument is used to bind to a DSP; the suffix is used to continue the traversal once a channel to the next DSP is established. With the security model in mind a decision was made to return the string representing the next DSP rather than a new DSP endpoint. To simplify authorization it was deemed better to let the client attempt the bind to the subsequent DSP itself rather than having a DSP do it on the client's behalf. Several examples of this will be detailed in section below describing Bind.

In SDS traversal of symbolic links also causes reparse messages. In this case linkFound=true, prefix=V and suffix=suffix(P). This will be described in detail in the section below on symbolic links.

3.1.2. Bind

```
        // attempt to bind to an service exp endpoint
        in     message Bind(char[]! in ExHeap path,
                            ServiceContract.Exp:Start! exp);
        out    message AckBind();
        // attempt to return unused endpoint if possible
        out    message NakBind(ServiceContract.Exp:Start exp,
                               ErrorCode code);
        out    message NakBindReparse(char[]! in ExHeap path,
                               char[] in ExHeap rest,
                               bool linkFound,
                               ServiceContract.Exp:Start! exp);
```

Clients interact with SDS by sending bind requests to it. A fresh channel endpoint is included in the lookup request. If the bind succeeds, then the client and server can communicate on the channel.

Services interact with the SDS by responding to bind requests. When a service registers with an SDS, it supplies a fresh channel for bind requests. Whenever a bind request comes in, the service would most likely create a thread to handle client requests on the channel passed in with the lookup message.

The following chronology illustrates how the SDS is typically used to handle the operations needed to set up establish communications with a service, such as SimpleService found in Figure 1. Assume that the processes **C**, **S**, and **DSP** represent a client, a service, and a Directory Service Provider. We'll also assume that *ds:imp* and *ds:exp* represent the client import and server export endpoints of a Directory Service channel. In steps 1-3 the service registers itself with the SDS. Steps 4-8 the steps needed to complete a client-initiated bind to that service.

1. S creates new *sp* channel
2. (**S** to **NS** on *ds:imp*) Register with endpoint *sp:imp*
3. (**DSP** to **S** on *ds:exp*) Registration acknowledgement
4. C creates new *service* channel
5. (**C** to **DSP** on *ds:imp*) Bind with endpoint *service:exp*
6. (**DSP** to **S** on *sp:imp*) Connect with *service:exp*
7. (**S** to **DSP** on *sp:exp*) Connect reply

Copyright © Microsoft Corporation. All Rights Reserved.

8. (**DSP** to **C** on ds*:exp*) Bind reply
9. **C** and **S** communicate using channel *service*

A more complicated yet common scenario is shown in Figure 2 when the client attempts to perform Bind("/FS/c/File1"). In this case there are 2 DSPs involved. In Steps 1-3 the FS DSP registers itself as a service with the root DSP. Steps 4-6 are the client's initial attempt to resolve the path. In Step 6 the DSP responds that a reparse point has be encountered, i.e. that a service was encountered before the end of the path. The reparse message tells the client where to bind to next to continue the traversal and the rest of the original path left to be parsed. In Steps 7-9 the client creates a new channel and attempts to bind to the service named in the NakReparse message with the DirectoryServiceContract; this establishes communication with the FS DSP. The client then continues the original bind operation by passing the suffix returned in the reparse message to the new DSP.

1. S creates new *sp* channel
2. (**S** to **DSP** on *ds:imp*) Register ("/fs") with endpoint *sp:imp*
3. (**DSP** to **S** on *ds:exp*) Registration acknowledgement
4. C creates new *file* channel
5. (**C** to **DSP** on *ds:imp*) Bind("/fs/dir1/file1") with endpoint *file:exp*
6. *(**DSP** to **C** on ds:exp) NakBindReparse("/fs", "dir1/file1")*
7. C creates new ds2 channel
8. (**C** to **DSP** on *ds:imp*) Bind("/fs") with endpoint *ds-2:imp as argument*
9. *DSP Acks*
10. (**C** to **S** on *ds-2:imp*) Bind("/dir1/file1") with endpoint *file:exp*
11. (**S** to **C** on *ds:exp*) Bind reply
12. **C** and **S** communicate using channel *file*

3.1.3. Enumeration

The Enumeration messages are used to retrieve the contents of a directory. In order to enumerate a directory one must first **Bind** to it. Once bound to a directory enumeration is a multi-step process. The typical message sequence is:

```
SendBeginEnumeration();
RecvEumerationEntries();
SendEndEnumeration();
```

The state machine for Enumeration messages can be found below. Once in the Enumerate state the client will receive either EnumerationEntries() or EnumerateTerminated() messages from the service. If the client receives an EnumerateEntries() message and the **moreEntries** argument is **true** then the client once again enters the Enumerate State. This process continues until there are no more entries, signified from the server by the EnumerationTerminated() message. The client then sends a EndEnumeration() message to the service terminating the enumeration process.

Copyright © Microsoft Corporation. All Rights Reserved.

```
        state Ready: one
        {
            BeginEnumeration? ->     Enumerate;
            …
        }

        state Enumerate : one {
            EnumerationTerminated!  -> Ready;
            EnumerationEntries! -> EnumerateAck;
        }

        state EnumerateAck : one {
            ReadEnumeration? -> Enumerate;
            EndEnumeration? -> Ready;
        }
```

EnumerateEntries returns an array of entries for the entire directory. The entries include the name as found within the path (i.e. not fully qualified) and the type of node it represents. The following **enum** defines the types returned.

```
    public enum NodeType
    {
        Directory,
        File,
        IoMemory,
        ServiceProvider,
        SymLink,
        BadNode,
    }
```

Below is the complete list of messages used for enumeration.

```
    // Enumerate the contents of the current directory
    in     message BeginEnumeration();
    in     message ReadEnumeration();
    in     message EndEnumeration();

    out    message EnumerationEntries (EnumerationRecords[]! in ExHeap results,
                                       bool moreEntries);
    out    message EnumerationTerminated(ErrorCode error);
```

3.1.4. Notify

A client can request notification of changes made within the SDS namespace. The arguments to Notify are the same as in Find above with the addition of import end of a notify contract. If the notify request is accepted the user will be sent a message for any modifications matching the <path,pattern> pair.

Copyright © Microsoft Corporation. All Rights Reserved.

```
public enum NotifyType {
    Creation,
    Deletion,
    Modifcation
}
in     message Notify(char[]! in ExHeap path,char[]! in ExHeap pattern,
                      NotifyContract.Imp:Start! imp);
out    message AckNotify();
out    message NakNotify(NotifyContract.Imp:Start! imp,
                         ErrorCode error);
```

Once the Notify message has been acknowledged the client will receive ChangeNotifcation messages on the export end of the Notify channel it owns. The **ChangeNotifcation** message returns the (short) name of the entity that has changed and an enum entry describing the type of change that occurred. Presently, creation and deletion changes are recognized.

```
public contract NotifyContract
{
    in  message Begin();

    in message ChangeNotification(char []! in ExHeap path, NotifyType type);
    out  message AckChangeNotification();

    state Start : Begin? -> Notify;

    state Notify : ChangeNotification? -> NotifyAck;
    state NotifyAck: AckChangeNotification! -> Notify;
}
```

3.2. Symbolic links

Symbolic links in Singularity are represented as string substitutions. At link creation time the client supplies the name where the link is to be inserted into the SDS and the value to be substituted when the name is traversed. When any of the traversing operations encounter a symbolic link, a NakReparse<operation> message is sent to the client. The prefix returned is string value stored at registration. The suffix is suffix(P) as of time of the traversal.

The client distinguishes symbolic links from DSP reparse messages by inspecting the linkFound argument. If linkFound is true then the client forms a new path by concatenating the returned suffix and prefix and re-issuing a bind request. This is different from reparse messages in that there is, at least at this point, no additional DSP to bind to.

Copyright © Microsoft Corporation. All Rights Reserved.

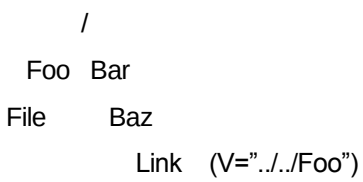
```
        in      message CreateLink(char []! in ExHeap linkPath,
                          char []! in ExHeap linkValue );
        out     message AckCreateLink();
        out     message NakCreateLink(ErrorCode code);
        out     message NakCreateLinkReparse(char[]! in ExHeap path,
                                char[]! in ExHeap rest,
                                bool linkFound);

        in      message DeleteLink(char []! in ExHeap linkPath);
        out     message AckDeleteLink();
        out     message NakDeleteLink(ErrorCode code);
        out     message NakDeleteLinkReparse(char[]! in ExHeap path,
                                char[]! in ExHeap rest,
                                bool linkFound);
```

3.2.1. Relative Symbolic Links

By design, symbolic links are just text substitutions within the SDS. "." (dot) and ".." (dotdot) have no significance within the SDS or in any File Systems supported. Support for dot and dotdot are expected to come from the client library.

The current proposal supports a simple scheme for handling dotdot within the client. When a reparse messages' linkFound argument is true the prefix argument will contain V, the value of the link. To handle relative links the actual prefix(P) is needed as well. Assume that within the library whenever a symbolic link is encountered prefix(P) is constructed by subtracting suffix(P) from P and the string CP is initialized to that value. For every ".." encountered in V we remove the tail if the path CP and the ".." front of V. Assuming no errors we now form the new path NP=CP+V+suffix(P) and reissue the operation. If CP is null and there are still ".."s left in V the result is an error.

For example assume the following tree;



Tying to bind to "/bar/baz/link/file" will cause a link reparse. At the time of the reparse prefix(P)=""/bar/baz", suffix(P)="/file" and V="../../foo" When it is time to assign NP the value of CP="" and the value of V="/foo". The resulting path is "/foo/file".

A more complex and richer solution has proposed that supports the user having channels to disparate parts of the SDS namespace and being able to "island hop" from one to the another using relative links. This presupposes the SIP is somehow given the Least Common Ancestor of all the parts of the SDS namespace it has access to. We have convinced ourselves that no changes are needed to the SDS contracts to enable this scenario.

Copyright © Microsoft Corporation. All Rights Reserved.

3.2.2. Registration

```
in     message Register(char[]! in ExHeap path,
                        ServiceProviderContract.Imp:Start! imp);
out    message AckRegister();
out    message NakRegister(ServiceProviderContract.Imp:Start imp,
                           ErrorCode error);
out    message NakRegisterReparse(char[]! in ExHeap path,
                          char[]! in ExHeap rest,
                          ServiceProviderContract.Imp:Start imp);
```

The Singularity Namespace contract provides a message for registering a service. As described in Section 2, a service must support a channel conforming to the ServiceProviderContract. When registering the service provides the name under which the service wants to be registered and the import endpoint to a channel conforming to the ServiceProvider contract discussed in Section 2.2. The nack

3.3. Directory operations

Directories provide a mechanism for structuring entities within a DSP. In fact, DSPs themselves are directories. The DirectoryServiceContract provides messages for creating and deleting subdirectories within a DSP. Internal directories within a DSP can be bound to with a DirectoryServiceContract just as a DSP can.

```
////////////////////////////////////////////////////
// Directory-related messages
////////////////////////////////////////////////////
in      message CreateDirectory(char []! in ExHeap dirName);
out     message AckCreateDirectory();
out     message NakCreateDirectory(ErrorCode code);

in      message DeleteDirectory(char []! in ExHeap dirName);
out     message AckDeleteDirectory();
out     message NakDeleteDirectory(ErrorCode code);
```

3.4. ACLs

```
////////////////////////////////////////////////////
// Security-related messages
////////////////////////////////////////////////////
in      message QueryACL(char []! in ExHeap fileName,
            byte[]! in ExHeap permission);
out     message AckQueryACL(byte[]! in ExHeap acl);
out     message NakQueryACL(ErrorCode code);

in      message StoreACL(char []! in ExHeap fileName,
            byte[]! in ExHeap permission,
            byte[]! in ExHeap acl);
out     message AckStoreACL();
out     message NakStoreACL(ErrorCode code);
```

Copyright © Microsoft Corporation. All Rights Reserved.

4. Security Considerations

The Directory Service in Singularity will implement access control mechanisms as described in the Updated Security Model document (SDN9a). In general, Directory service implementations will be responsible for protecting the portion of the namespace that they export. The Singularity Security Service and Library provide a general-purpose API for implementing access controls. This API provides a method for interpreting access control data, but not storing it. Hence any directory service must itself store such access control data alongside the objects it controls. Client input to this data is achieved through the Get/StoreACL operations above.

In the SDN9a access control model, each reference monitor (e.g. an application that guards a resource) defines a set of permissions required for each exported operation. For each such permission and for each directory object, an access control expression dictates the set of security principals for which the permission will be granted. The exact access control permissions that will be required for the operations in the Directory Service contract is not yet decided. Here is our current thinking:

Permission	Operation	Object
Read	Find	directory
	FindAndNotify	directory
	Attributes	child
	ReadLink	child
	QueryACL	child
	Bind (for read file)	child
Traverse	All	intermediate path nodes
Write	CreateDirectory	parent directory
	DeleteDirectory	parent directory
	CreateFile	parent directory
	CreateAndBindFile	parent directory, child
	DeleteFile	parent directory
	CreateLink	parent directory
	DeleteLink	parent directory
	Register	parent directory
	Bind (for write file)	child
Own	StoreACL	child
Register	Register	child

So, for example, in order to open a file for reading, the caller must be granted Read permission on the file and Traverse permission on the intermediate nodes in the file path. In order to create and bind to a file for writing, the caller must have Write permissions for the parent directory (or Write permissions on the file itself if it exists) plus Traverse permission on the intermediate nodes in the path.

The Bind operation is somewhat of a special case. Permission to traverse the namespace is controlled by the Directory service. However, the right to Bind to a particular point in the namespace is ultimately controlled by the service that has registered the target endpoint. In the table above, for example, the Directory service itself registers files that can be bound to for reading and writing. In the case that a name is registered by a non-Directory service, that service must control its own permissions and access control infrastructure.

The client-driven parsing of paths that this directory service model employs has the convenient property that all Directory operations take place on a direct channel between the client and the service. Thus, the Directory service can determine the identity of the caller by looking at the properties of the communication channel. For the Bind operation, however, the identity associated with the channel being proffered for binding is checked by the target service (either the Directory service or a different service that has registered the target name). The target service may also perform an access check against the identity associated with the caller's channel.

Register is a particularly sensitive operation because most communications in Singularity is rooted in channels to named entities. A malicious service registered at a trusted part of the namespace will control that namespace

Copyright © Microsoft Corporation. All Rights Reserved.

entirely. This is especially problematical for names that are near the top of the naming hierarchy. Although channel clients can, in principle, authenticate service providers (including Directory service providers) and make security-relevant decisions based on this information, we do not expect this to be the default behavior. Instead, we expect most Singularity applications to trust that system policy prevents rogue services from registering themselves in well-known parts of the namespace. This system policy will be realized through access controls enforced by the Directory services on register operations. The exact mechanism for expressing and implementing this policy is yet to be determined, but the likely starting point will be manually controlled access control expressions (or policy definitions) for the root namespace specified as part of the boot image.

5. Appendix 1

Copyright © Microsoft Corporation. All Rights Reserved.

```
//////////////////////////////////////////////////////////////////////////////
//
//  Microsoft Research Singularity
//
//  Copyright (c) Microsoft Corporation.  All rights reserved.
//
//  File:   DirectoryServiceContract.sg
//

using Microsoft.SingSharp;
using Microsoft.Singularity.Channels;

namespace Microsoft.Singularity.Directory
{
    // enumeration of possible errors reported by any operation
    public enum ErrorCode : uint
    {
        NoError                 = 0,
        AccessDenied            = 1,
        AlreadyExists           = 2,
        BadArguments            = 3,
        ContractNotSupported    = 4,
        DirectoryNotEmpty       = 5,
        NotFound                = 6,
        ChannelClosed           = 7,
        Unsatisfiable           = 8,
        NotImplemented          = 9,
        NotSupported            = 10,
        CapacityReached         = 11,
        InsufficientResources   = 12,
        DirectoryFull           = 13,
        NotDirectory            = 14,
        NotLink                 = 15,
        NotProvider             = 16,
        NotFile                 = 17,
        Unknown                 = 99,
    }

    // enumeration of the existing node types found in the namespace
    // impliemtation and the filesystem
    public enum NodeType
    {
        Directory,
        File,
        IoMemory,
        ServiceProvider,
        SymLink,
        BadNode,
    }

    // a REP struct used to return responses to find or notify.
    public rep struct EnumerationRecords : ITracked
    {
```

Copyright © Microsoft Corporation. All Rights Reserved.

```
        public char[]! in ExHeap Path; // short name within a directory
        public NodeType Type; // its type
    }

    public contract DirectoryServiceContract : ServiceContract
    {

        // REPARSE MESSAGES
        // used for 2 different scenarios: DirectoryProvider and SymLink traversal
        //  DirectoryProvider:
        //      path=new direcoryProvider to bind to to continue traversal
        //      rest=suffix(P): the remainder of the initial path not yet parsed.
        //  SymLink
        //      path=contents of the symbolic link
        //      rest=suffix(P): the remainder of the initial path not yet parsed.


        // attempt to bind to an service exp endpoint
        in     message Bind(char[]! in ExHeap path,
                            ServiceContract.Exp:Start! exp);
        out    message AckBind();
        // attempt to return unused endpoint if possible
        out    message NakBind(ServiceContract.Exp:Start exp,
                               ErrorCode code);
        out    message NakBindReparse(char[]! in ExHeap path,
                               char[] in ExHeap rest,
                               bool linkFound,
                               ServiceContract.Exp:Start! exp);

        // Find path and notify caller of any changes matching pattern over
        // the imp endpoint supplied.
        in     message Notify(char[]! in ExHeap path,
                              char[]! in ExHeap pattern,
                              bool sendExisting,
                              NotifyContract.Imp:Start! imp);
        out    message AckNotify();
        out    message NakNotify(NotifyContract.Imp:Start! imp,
                                 ErrorCode error);
        out    message NakNotifyReparse(char[]! in ExHeap path,
                               char[]! in ExHeap rest,
                               bool linkFound,
                               NotifyContract.Imp:Start imp);

        // Enumerate the contents of the current directory
        in     message BeginEnumeration();
        in     message ReadEnumeration();
        in     message EndEnumeration();

        out    message EnumerationEntries (EnumerationRecords[]! in ExHeap results,
bool moreEntries);
        out    message EnumerationTerminated(ErrorCode error);

        // given a path return its nodetype and if meaningful its length
```

Copyright © Microsoft Corporation. All Rights Reserved.

```
        in     message GetAttributes(char []! in ExHeap path);
        out    message AckGetAttributes(NodeType type, long size);
        out    message NakGetAttributes(ErrorCode code);
        out    message NakGetAttributesReparse(char[]! in ExHeap path,
                           char[]! in ExHeap rest,
                           bool linkFound);


        // given a path return the associated ACL if present
        in     message QueryACL(char []! in ExHeap path,
                    byte[]! in ExHeap permission);
        out    message AckQueryACL(byte[]! in ExHeap acl);
        out    message NakQueryACL(ErrorCode code);
        out    message NakQueryACLReparse(char[]! in ExHeap path,
                           char[]! in ExHeap rest,
                           bool linkFound);


        // Greater permissions needed for ops below

        // register a service's name serviceProvider endpoint with the SDS
        in     message Register(char[]! in ExHeap path,
                            ServiceProviderContract.Imp:Start! imp);
        out    message AckRegister();
        out    message NakRegister(ServiceProviderContract.Imp:Start imp,
                                   ErrorCode error);
        out    message NakRegisterReparse(char[]! in ExHeap path,
                           char[]! in ExHeap rest,
                           bool linkFound,
                           ServiceProviderContract.Imp:Start! imp);

        // Deregister a service's name from the SDS
        in     message Deregister(char[]! in ExHeap path);
        out    message AckDeregister(ServiceProviderContract.Imp:Start! imp);   //
return deregistered endpoint
        out    message NakDeregister(ErrorCode error);
        out    message NakDeregisterReparse(char[]! in ExHeap path,
                           char[]! in ExHeap rest,
                           bool linkFound);


        // create dirName within the current directory
        // the current directory is the one bound on this channel
        in     message CreateDirectory(char []! in ExHeap dirName);
        out    message AckCreateDirectory();
        out    message NakCreateDirectory(ErrorCode code);
        out    message NakCreateDirectoryReparse(char[]! in ExHeap path,
                           char[]! in ExHeap rest,
                           bool linkFound);
        // delete dirName within the current directory
        // the current directory is the one bound on this channel
        in     message DeleteDirectory(char []! in ExHeap dirName);
        out    message AckDeleteDirectory();
```

Copyright © Microsoft Corporation. All Rights Reserved.

```
        out    message NakDeleteDirectory(ErrorCode code);
        out    message NakDeleteDirectoryReparse(char[]! in ExHeap path,
                                char[]! in ExHeap rest,
                                bool linkFound);
        // create file within the current directory
        // the current directory is the one bound on this channel
        // file creation only applies to the FS
        in     message CreateFile(char []! in ExHeap fileName);
        out    message AckCreateFile();
        out    message NakCreateFile(ErrorCode code);
        out    message NakCreateFileReparse(char[]! in ExHeap path,
                                char[]! in ExHeap rest,
                                bool linkFound);

        // create and "open" or bind to fileName within the current directory
        // the current directory is the one bound on this channel
        // this is common enough pattern to supply an optimization
        in     message CreateAndBindFile(char []! in ExHeap fileName,
                          FileContract.Imp:Ready! imp);
        out    message AckCreateAndBindFile();
        out    message NakCreateAndBindFile(FileContract.Imp:Ready imp,
                                            ErrorCode code
                                            );

        // delete fileName within the current directory
        // the current directory is the one bound on this channel
        in     message DeleteFile(char []! in ExHeap fileName);
        out    message AckDeleteFile();
        out    message NakDeleteFile(ErrorCode code);
        out    message NakDeleteFileReparse(char[]! in ExHeap path,
                                char[]! in ExHeap rest,
                                bool linkFound);

        // for the given path and permission, return the associated ACL if present

        in     message StoreACL(char []! in ExHeap fileName,
                     byte[]! in ExHeap permission,
                     byte[]! in ExHeap acl);
        out    message AckStoreACL();
        out    message NakStoreACL(ErrorCode code);
        out    message NakStoreACLReparse(char[]! in ExHeap path,
                                char[]! in ExHeap rest,
                                bool linkFound,
                                byte[]! in ExHeap permission,
                                byte[]! in ExHeap acl
                                 );
        // create a symlink node at linkPath with value linkValue
        // Upon traveral linkPath will be returned to client
        // it is up to the client to interpret and re-submit bind
        // see reparse messages below.

        in     message CreateLink(char []! in ExHeap linkPath,
                          char []! in ExHeap linkValue );
```

Copyright © Microsoft Corporation. All Rights Reserved.

```
        out    message AckCreateLink();
        out    message NakCreateLink(ErrorCode code);
        out    message NakCreateLinkReparse(char[]! in ExHeap path,
                            char[]! in ExHeap rest,
                            bool linkFound);

        // delete symlink node at linkPath
        in     message DeleteLink(char []! in ExHeap linkPath);
        out    message AckDeleteLink();
        out    message NakDeleteLink(ErrorCode code);
        out    message NakDeleteLinkReparse(char[]! in ExHeap path,
                            char[]! in ExHeap rest,
                            bool linkFound);

        // Get symlink value
        in     message GetLinkValue(char []! in ExHeap linkPath);
        out    message AckGetLinkValue(char []! in ExHeap linkPath);
        out    message NakGetLinkValue(ErrorCode code);
        out    message NakGetLinkValueReparse(char[]! in ExHeap path,
                            char[]! in ExHeap rest,
                            bool linkFound);


        out    message Success();

        ////////////////////////////////////////////////////
        // State  Machine
        ////////////////////////////////////////////////////

        override state Start: one {
            Success! -> Ready;
        }

        state Enumerate : one {
            EnumerationTerminated!  -> Ready;
            EnumerationEntries! -> EnumerateAck;
        }

        state EnumerateAck : one {
            ReadEnumeration? -> Enumerate;
             EndEnumeration? -> Ready;
        }


        state Ready: one
        {
            Bind? ->                (AckBind!
                                     or NakBind!
                                     or NakBindReparse!
                                    ) -> Ready;

            BeginEnumeration? ->     Enumerate;
```

Copyright © Microsoft Corporation. All Rights Reserved.

```
            CreateAndBindFile? ->   (AckCreateAndBindFile!
                                     or NakCreateAndBindFile!
                                    ) -> Ready;

            CreateDirectory? ->     (AckCreateDirectory!
                                     or NakCreateDirectory!
                                     or NakCreateDirectoryReparse!
                                    ) -> Ready;

            CreateFile? ->          (AckCreateFile!
                                     or NakCreateFile!
                                     or NakCreateFileReparse!
                                    ) -> Ready;

            CreateLink? ->          (AckCreateLink!
                                     or NakCreateLink!
                                     or NakCreateLinkReparse!
                                    ) -> Ready;


            DeleteDirectory? ->     (AckDeleteDirectory!
                                     or NakDeleteDirectory!
                                     or NakDeleteDirectoryReparse!
                                    ) -> Ready;

            DeleteFile? ->          (AckDeleteFile!
                                     or NakDeleteFile!
                                     or NakDeleteFileReparse!
                                    ) -> Ready;

            DeleteLink? ->          (AckDeleteLink!
                                     or NakDeleteLink!
                                     or NakDeleteLinkReparse!
                                    ) -> Ready;

            Deregister? ->          (AckDeregister!
                                     or NakDeregister!
                                     or NakDeregisterReparse!
                                    ) -> Ready;


            GetAttributes? ->       (AckGetAttributes!
                                      or NakGetAttributes!
                                      or NakGetAttributesReparse!
                                    ) -> Ready;

            GetLinkValue? ->          (AckGetLinkValue!
                                     or NakGetLinkValue!
                                     or NakGetLinkValueReparse!
                                    ) -> Ready;
            Notify? ->              (AckNotify!
                                     or NakNotify!
                                     or NakNotifyReparse!
```

Copyright © Microsoft Corporation. All Rights Reserved.

```
                                    ) -> Ready;

            QueryACL? ->            (AckQueryACL!
                                     or NakQueryACL!
                                     or NakQueryACLReparse!
                                     ) -> Ready;

            Register? ->            (AckRegister!
                                     or NakRegister!
                                     or NakRegisterReparse!
                                    ) -> Ready;

            StoreACL? ->            (AckStoreACL!
                                     or NakStoreACL!
                                     or NakStoreACLReparse!
                                     ) -> Ready;
        }
    }
}
```

Copyright © Microsoft Corporation. All Rights Reserved.