Singularity Design Note

36

Service Management Service

A User-Level Service Management

This document describes Service Management Service (SMS) that is in charge of user-level service activities. Given the dedicated channel between SMS and a service, SMS is able to start, stop and monitor its activity. A privileged client is also allowed to those operations through SMS.

1. System Overview

Service Management Service (SMS) itself is launched by the kernel at the system boot time and runs as an SIP. Most of the other services are launched by SMS. It defines two channel contracts for user-level services and clients. User-level services are registered to SMS by using the system manifest file or through the channel on the client side. The clients establish the interactions with SMS via the Name Service.

1.1. Features

Service Launcher

Most services are launched by SMS. The services launched by SMS are registered to the registry inside SMS.

Configurable Boot Process

SMS determines the services launched at the system boot time according to the system-wide manifest file.

Service Channel

SMS provides a channel for a client to control the services under the SMS control.

1.2. Processes and Interactions

Figure 1 shows all the interactions of the system. The rectangles represent SIPs and the arrows represent channels. SMS has four types of channels total: two for the Name Service, one for a client and one for a service. The services including the Name Service provide channels for their own services in addition to the channels for SMS[1].

A client connects to SMS through the Name Service. SMS is registered to the Name Service in prior to accepting the connections from the clients. Also, the other services have to be registered to SMS in advance. Then, the client can control the services and obtain their status information through SMS.

[1] Current implementation of Name Service has no channels to SMS.

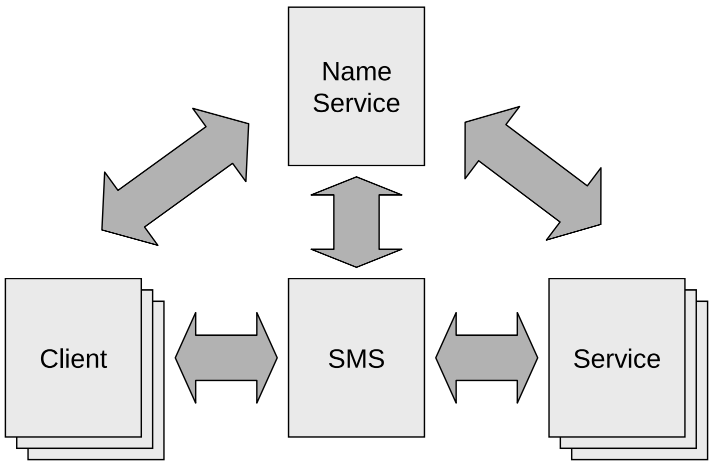

Figure 1. Process and Channels around SMS

Figure 2 illustrates the interactions among the kernel, SMS, and services. SMS (depicted SrvMng) consists of the user-level component and the in-kernel component. The in-kernel SMS plays a role of parsing the configuration information and creating a process for the user-level SMS. The in-kernel SMS is a component and has no own threads.

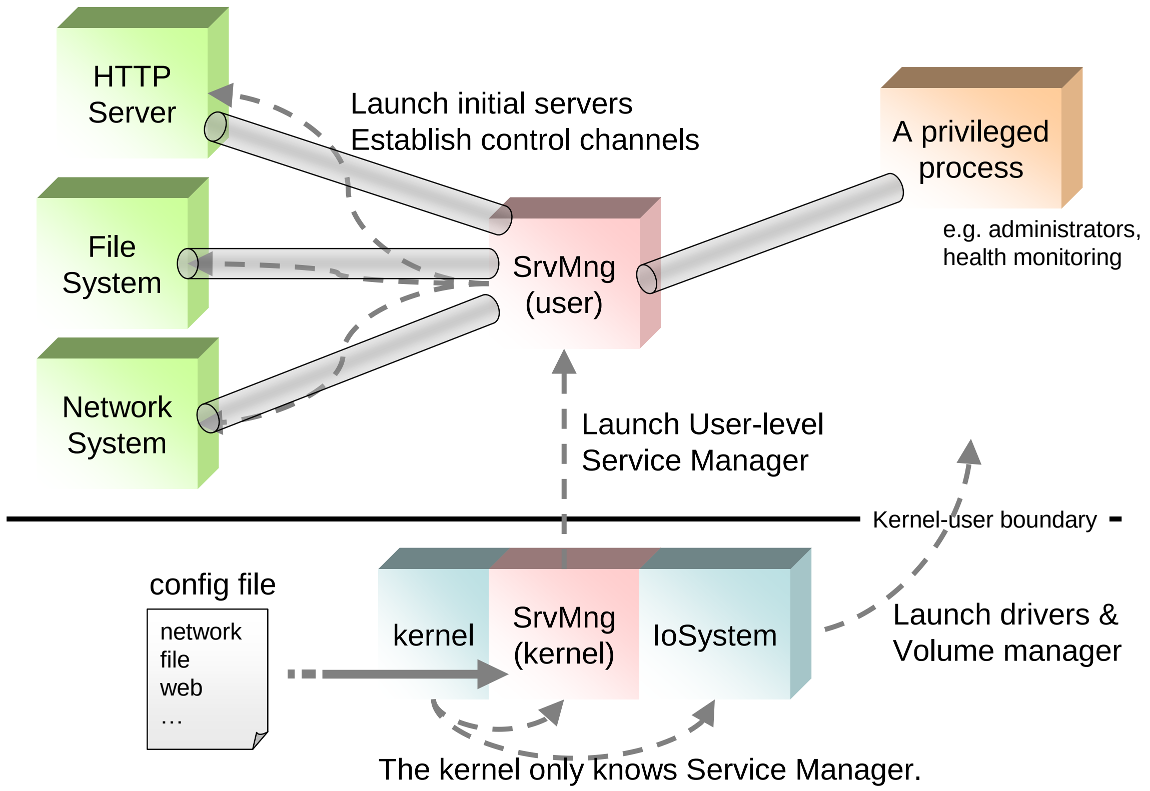
HTTP Server
File System
Network System
Launch initial servers
Establish control channels
A privileged process
e.g. administrators, health monitoring
SrvMng (user)
Launch User-level Service Manager
Kernel-user boundary
config file
network
file
web
…
kernel
SrvMng (kernel)
IoSystem
Launch drivers & Volume manager
The kernel only knows Service Manager.

Figure 2. System Components Interactions Overview

2. Contracts

SMS provides two types of channel contracts. `ManagedServiceContract` is a contract for services and `ServiceManagementContract` is a contract for its clients.

2.1. Managed Service Contract

`ManagedServiceContract` defines the channel between SMS and services (Figure 3). In addition to the contracts for its own services, a service is required to implement this contract so that SMS controls its activity. In `ManagedServiceContract`, SMS runs as the client of services.

```
contract ManagedServiceContract {
  out message Success();
  out message Busy();
  in message Knock();
  out message Alive();
  in message RestartService();
  out message ServiceRestarted();
  in message StopService();
  out message ServiceStopped();

  state Start : one {
    Success! -> Ready;
  }
  state Ready : one {
    Knock? -> Alive! -> Ready;
    RetartService? -> (ServiceRestarted! or Busy!) -> Ready;
    StopService? -> (ServiceStopped! or Busy!);
  }
}
```

Figure 3. Managed Service Contract

A service can send a *busy* reply upon a restart or stop request. It is used for the case where the server is running on another channel and cannot disconnect it immediately.

2.2. Service Management Contract

`ServiceManagementContract` defines the channel between SMS and the clients. It inherits `ServiceContract` so that SMS is referenced by clients via Directory Service. The default service location in the Directory Service is `/services`.

The contract defines five operations. `Start`, `Stop`, and `Restart` are creating and starting a service process, terminating a service process, and restarting a service process respectively. `Knock` is used for examining if a server is alive or not. If the server is running, SMS receives `Alive` message, otherwise it receives `NotAlive` message. A sequence of the enumeration messages shows current running services.

The contract also defines four types or error messages upon its operations. `NotFound` message represents that an operation failure due to service inexistence. This message is used when a service is not found either in a file system or the registry in SMS. `PermissionDenied` message is replied when a client access to the specified server is prohibited. `OutOfResource` message indicates that process creation failed because of lack of system resources. Busy message is replied if SMS receives a `Busy` message from the server. Note that, all the operations don't always support all error messages because some error messages are inappropriate for those operations.

```
public contract ServiceManagementContract : ServiceContract

    {

        public const string ModuleName = "/services";


        out message Success();


        out message NotFound();

        out message PermissionDenied();

        out message OutOfResource();

        out message Busy();


        in message StartService(char[]! in ExHeap name);

        out message AckStartService();


        in message StopService(int pid);

        out message AckStopService();


        in message RestartService(int pid);

        out message AckRestartService();

        in message Knock(char[]! in ExHeap name);

        out message Alive();
        out message NotAlive();


        in message BeginEnumeration();

        in message MoveNext();

        in message EndEnumeration();

        out message EnumerationTerminated();

        out message Current(int pid, char []! in ExHeap name);

        ...
```

3. Components

SMS consists of two contracts and four objects.

3.1. Class ServiceManager

ServiceManager object is the entry point of SMS. It creates Acceptor object and launches the initial servers, which are described in the system manifest file.

3.2. Class Acceptor and Class ServiceController

Acceptor has a thread that waits for an SMS client connection request on the ServiceProviderContract channel, which is a channel to the Name Service. When Acceptor accepts the request, it spawns a ServiceController object. ServiceController is a counterpart of SMS clients; it receives control requests and forwards them to the running services through Service objects. A ServiceController is stopped and removed when the connection with the client is lost.

3.3. Class Service

A Service object represents a running service. Basically, a service process starts with creation of a Service object and stops with finalization of it. This class is constructed in a Factory-like pattern[2] to compensate the immaturity of the design; it is prohibited to create a Service object with the *new* operator. Service Manager doesn't allow duplicating a service process. To enforce this design, Service class provides a static method NewService and checks the existence of the specified service in the method. To explicitly stop a service, ReleaseService method is provided.

[2] I don't know what exactly this pattern is called.

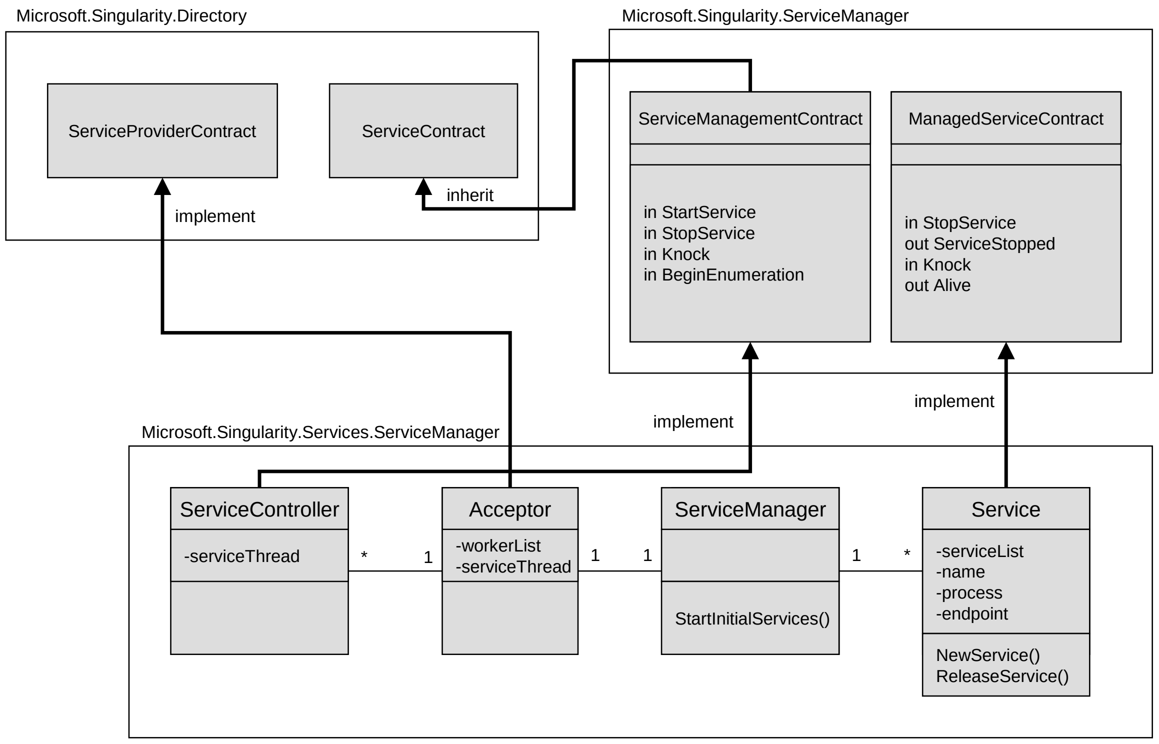

Figure 4. SMS Object Diagram. Only essential members appear here.

4. Boot Process

SMS is created and started as an SIP by the kernel (Figure 2). At this stage, the service configuration is given to SMS. Then SMS launches the servers according to the service configuration. The services are launched in order from the top of entries to the bottom. Before it moves in to server mode, it registers itself to Name Service that it has launched in the previous stage.

The service configuration is described in `Distro/LegacyPCDistro.xml`. A service is described by means of service directive in serviceConfig directive (Figure 5).

```
<serviceConfig>

  <service name=”...” mode=”managed | unmanaged” />

</serviceConfig>
```

Figure 5. Service configuration syntax

Currently, the server configuration supports **name** and **mode** attributes. The name parameter specifies a service name. And the mode parameter specifies how the service is started. The current implementation supports **managed** and **unmanaged** mode. The managed mode is default. The unmanaged mode is an obsolete mode; it is provided for a legacy service that has no `ManagedServiceContract` channels.

An example of the service configuration is shown below.

```
<serviceConfig>
```

```
  <service name=”boxwood” mode=”unmanaged” />

  <service name=”iso9660” mode=”managed” />

  <service name=”netstack” />

</serviceConfig>
```

Figure 6. An example of service configuration

5. Service Start

SMS allows a client to start service processes. This section explains the internal of the service start process.

5.1. Operation

Figure 7 shows the service start operation with no errors.

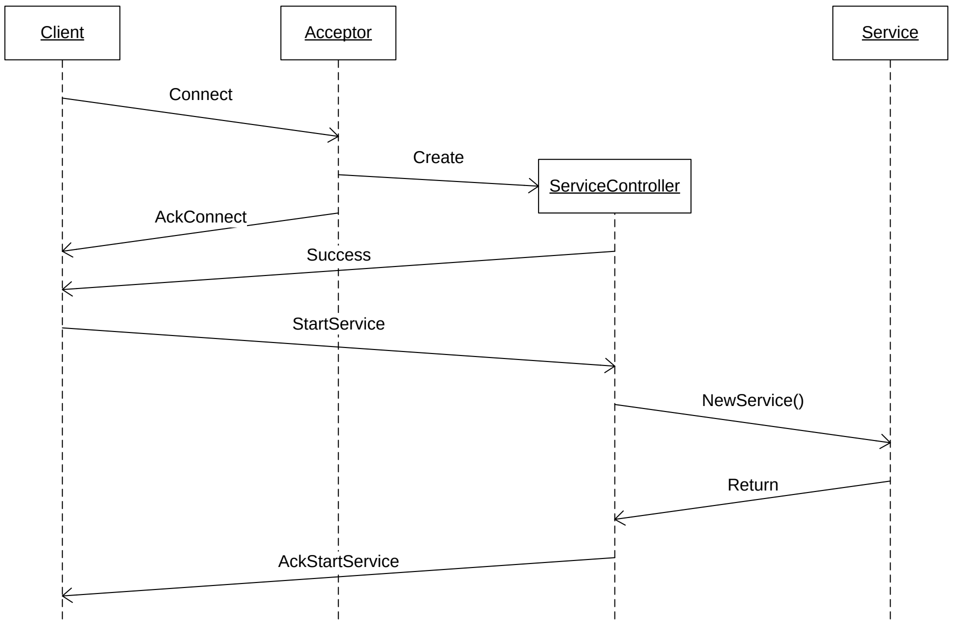

Figure 7. Communication between objects at a Start Service operation

5.2. Limitation of Placement Policy

Every service process managed by SMS is loaded by means of Binder object in the kernel. The location of a service depends on Binder’s policy. Therefore, SMS cannot effectively deal with a service that is frequently loaded and unloaded.

6. Service Termination

SMS can terminate the servers that SMS has created by sending a `Stop` message or by stopping the process. Receiving a `Stop` message through the server channel, it is a server’s responsibility whether it terminates or not.

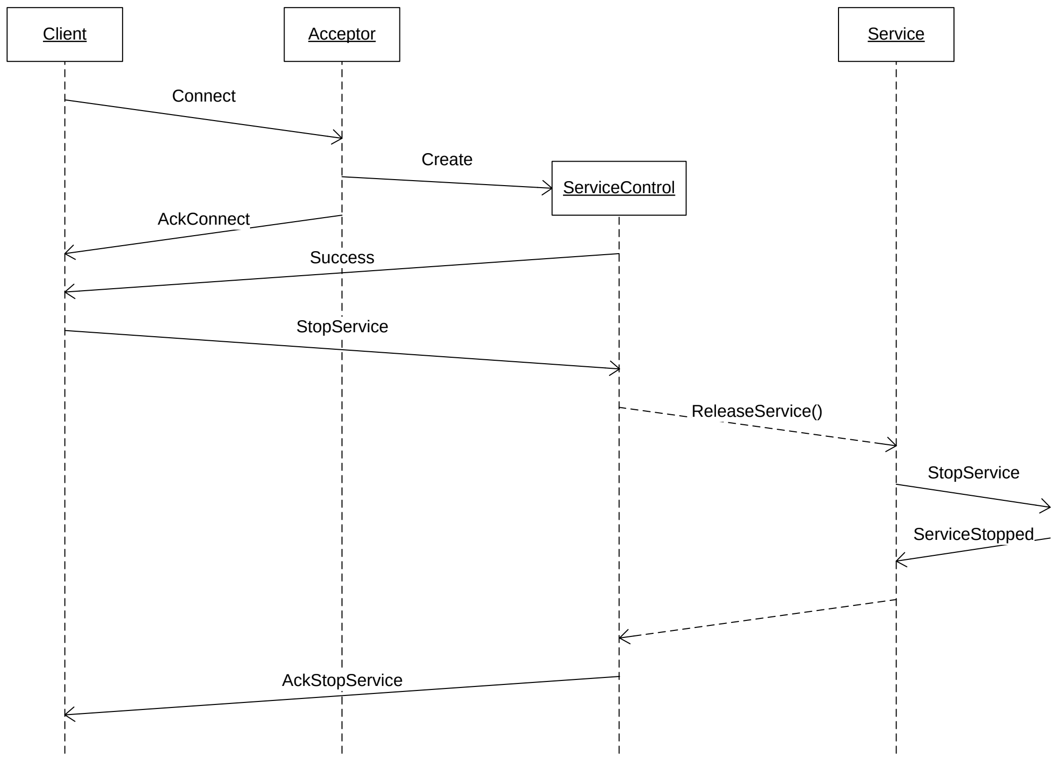

Figure 8. Communication between objects at a Stop Service operation

7. Sample Program

7.1. Service Configurator

svconf is a shell command that directly connects to SMS (Figure 9). The command supports all the operations that SeriviceManagementContract provides[3].

```
> svconf command argument

  @start <name>       - Start the service process

  @stop <pid>        - Stop the service process

  @restart <pid>     - Restart the service process

  @list               - Show all the services currently available

  @stat <name>        - Examine if the service is alive
```

Figure 9. SMSClient command help

8. Test

Service Manager is tested in terms of wrong operations and scalability.

[3] Restart operation has not been implemented yet.

8.1. Operational Test

- Test for invalid operations
- Test for multiple single operations
- Test for various timing of operations
- Unanticipated channel closure

9. Future Work

9.1. Access Control

The current implementation allows any kind of clients to control any service under the SMS's control. This is a serious security hole. A future release will involve mechanisms to control the accesses from the clients to the services.

9.2. Service Restart

Although the proposed two contracts define RestartService, current implementations don't support it at all.

9.3. Service Monitoring

9.3.1. Polling

9.3.2. Exception Handling

9.4. Collaboration with Binder