BAE SYSTEMS
INSPIRED WORK

Simple Lifecycle Example

Introduction

This document describes an ECOA® component lifecycle example named "*Simple Lifecycle Example*".

The example is an extension to the "Simple Example" (ref. [3]) client-server example, with the addition of a simple component lifecycle service.

This document presents the principal user generated artefacts required to create the "*Simple Lifecycle Example*" component lifecycle example using the ECOA. It is assumed that the reader is conversant with the ECOA Architecture Specification (ref. [1]) and the process of defining and declaring ECOA Assemblies, ASCs (components), Modules, and deployments in XML, and then using code generation to produce Module framework (stub) code units and ECOA Container and Platform code.

Aims

This ECOA "*Simple Lifecycle Example*" component lifecycle example is intended to demonstrate the use of a "component lifecycle" service to manage the functional start-up and shutdown of components. It is based on an example lifecycle service defined in the ECOA System Management Guidance document (ref. [2]).

ECOA Features Exhibited

- Composition of an ECOA Assembly of multiple ECOA ASCs (components).
- Contention-free resource sharing within an ECOA Assembly.
- Use of the ECOA runtime logging API.
- Use of a component lifecycle service to manage functional start-up/shutdown of components.

Design and Definition

Client-Server Functional Design

The "Simple Lifecycle Example" component lifecycle example will demonstrate a simple component lifecycle service scheme, whereby a manager component orchestrates the functional start-up and shutdown of two managed components. When the client is in a "running" state, it will periodically perform a request, from the server and will receive a data item in return if the

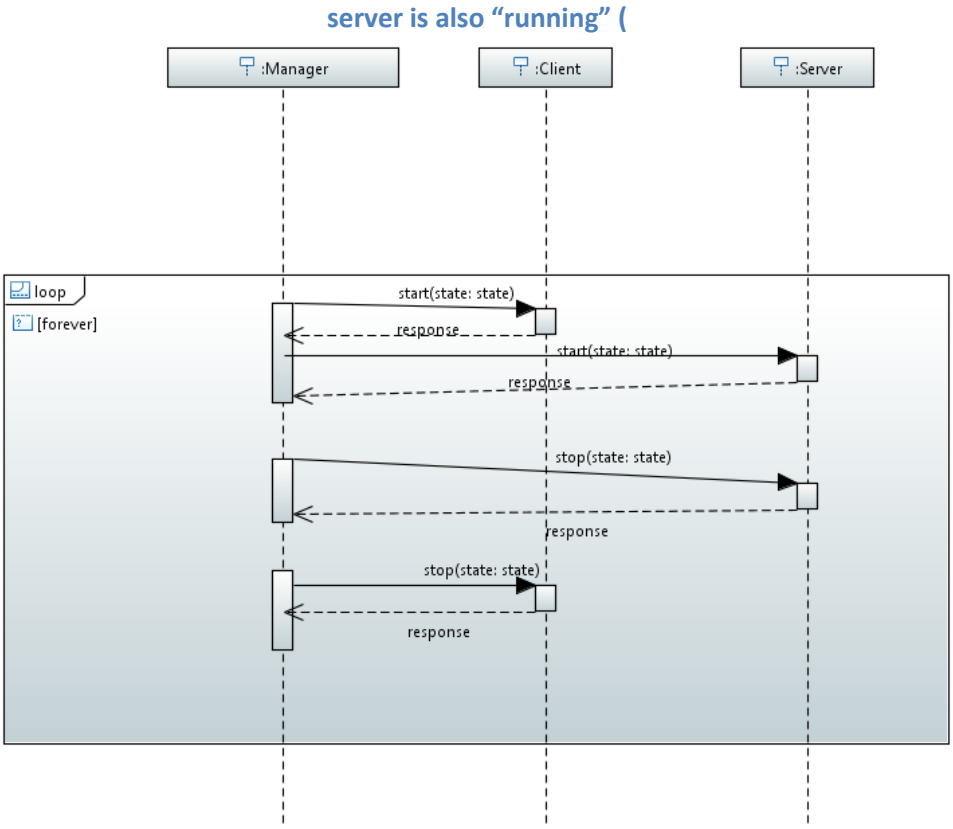

Figure 1).

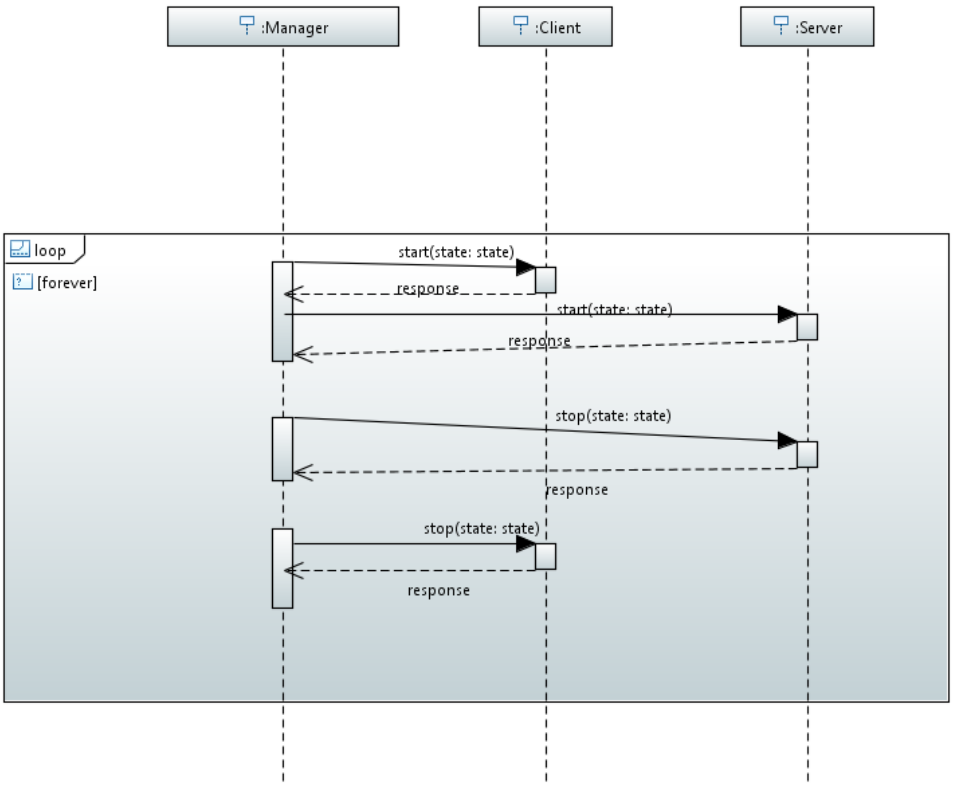

Figure 1 - ECOA "Simple Lifecycle Example" component lifecycle Behaviour

The data content of the request will be the current absolute time and the response will be of a user defined type.

The Client will set a local variable to zero and output this to the log prior to performing the request. The result will be returned into this variable and logged.

The Client will be periodically activated at a rate of 0.5Hz (once every 2 seconds).

ECOA Assembly Design and Definition

This ECOA "*Simple Lifecycle Example*" component lifecycle example ECOA Assembly comprises three ECOA ASCs named "*Manager*", "*Client*" and "*Server*". The "*Manager*" ASC type is instantiated once within the ECOA Assembly as "*Manager_Inst*". The "*Client*" ASC type is instantiated once within the ECOA Assembly as "*Client_Inst*". The "*Server*" ASC is instantiated once within the ECOA Assembly as "*Server_Inst*" and provides the "*Provide_Value_Service*" ECOA Service, which is referenced (used) by the "*Client_Inst*" ASC (Figure 2).

In addition, each of the managed components ("*Client_Inst*" and "*Server_Inst*"), provide the "*simple_lifecycle*" service. This service allows the "*Manager*" component to manage the functional state of the managed components.

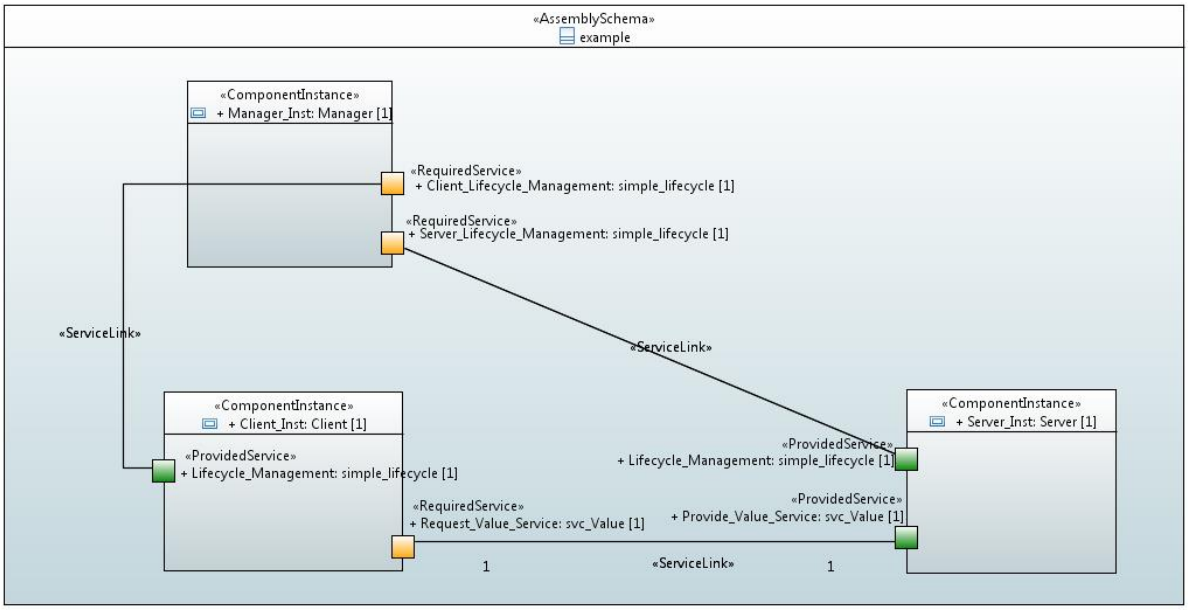

Figure 2 - ECOA "Simple Lifecycle Example" Assembly Diagram

This ECOA Assembly is defined in an Initial Assembly XML file, and declared in a Final Assembly (or Implementation) XML file (which is practically identical). The Final Assembly XML for the ECOA "*Simple Lifecycle Example*" Assembly is as follows (file *example.impl.composite*):

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<csa:composite
   xmlns:csa="http://docs.oasis-open.org/ns/opencsa/sca/200912"
   xmlns:ecoa-sca="http://www.ecoa.technology/sca-extension-2.0"
   name="example"
   targetNamespace="http://www.ecoa.technology">


   <csa:component name="Client_Inst">
      <ecoa-sca:instance componentType="Client">
         <ecoa-sca:implementation name="Client_Im"/>
      </ecoa-sca:instance>

      <csa:reference name="Request_Value_Service">
         <ecoa-sca:interface syntax="svc_Value"/>
      </csa:reference>

      <csa:service name="Lifecycle_Management">
         <ecoa-sca:interface syntax="simple_lifecycle"/>
      </csa:service>

   </csa:component>

   <csa:component name="Server_Inst">
```

```
      <ecoa-sca:instance componentType="Server">
         <ecoa-sca:implementation name="Server_Im"/>
      </ecoa-sca:instance>

      <csa:service name="Provide_Value_Service">
         <ecoa-sca:interface syntax="svc_Value"/>
      </csa:service>

      <csa:service name="Lifecycle_Management">
         <ecoa-sca:interface syntax="simple_lifecycle"/>
      </csa:service>

   </csa:component>

   <csa:component name="Manager_Inst">
      <ecoa-sca:instance componentType="Manager">
         <ecoa-sca:implementation name="Manager_Im"/>
      </ecoa-sca:instance>

      <csa:reference name="Client_Lifecycle_Management">
         <ecoa-sca:interface syntax="simple_lifecycle"/>
      </csa:reference>

      <csa:reference name="Server_Lifecycle_Management">
         <ecoa-sca:interface syntax="simple_lifecycle"/>
      </csa:reference>

   </csa:component>

   <csa:wire source="Client_Inst/Request_Value_Service"
target="Server_Inst/Provide_Value_Service"/>

   <csa:wire source="Manager_Inst/Client_Lifecycle_Management"
target="Client_Inst/Lifecycle_Management"/>

   <csa:wire source="Manager_Inst/Server_Lifecycle_Management"
target="Server_Inst/Lifecycle_Management"/>

</csa:composite>
```

The *Server* ASC type is defined in XML as follows (file *Server.componentType*):

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:ecoa-sca="http://www.ecoa.technology/sca-extension-2.0">

   <service name="Provide_Value_Service">
      <ecoa-sca:interface syntax="svc_Value"/>
   </service>

   <service name="Lifecycle_Management">
      <ecoa-sca:interface syntax="simple_lifecycle"/>
```

```
   </service>

</componentType>
```

The ASC definition (the *<componentType>* XML element) declares the provision (by the ASC) of the *Provide_Value_Service* and the *Lifecycle_Management* ECOA Service.

The *Client* ASC type is defined in XML as follows (file *Client.componentType*):

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:ecoa-sca="http://www.ecoa.technology/sca-extension-2.0">

   <reference name="Request_Value_Service">
      <ecoa-sca:interface syntax="svc_Value"/>
   </reference>

   <service name="Lifecycle_Management">
      <ecoa-sca:interface syntax="simple_lifecycle"/>
   </service>

</componentType>
```

This ASC definition (the *<componentType>* XML element) declares a reference (by the ASC) to the *Request_Value_Service* ECOA Service and the provision of the *Lifecycle_Management* ECOA Service.

The *Manager* ASC type is defined in XML as follows (file *Manager.componentType*):

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:ecoa-sca="http://www.ecoa.technology/sca-extension-2.0">

   <reference name="Client_Lifecycle_Management">
      <ecoa-sca:interface syntax="simple_lifecycle"/>
   </reference>

   <reference name="Server_Lifecycle_Management">
      <ecoa-sca:interface syntax="simple_lifecycle"/>
   </reference>

</componentType>
```

This ASC definition (the *<componentType>* XML element) declares a reference (by the ASC) to the Client_Lifecycle_Management and Server_Lifecycle_Management ECOA Services.

ECOA Service and Types Definition

svc_Value Service

The *svc_Value* Service, which is provided by the *Server* ASC and referenced by the *Client* ASC, is defined in a XML file (*svc_Value.interface.xml*):

```
<serviceDefinition xmlns="http://www.ecoa.technology/interface-2.0">

	<use library="example" />

	<operations>
		<requestresponse name="Request_Value">
			<input name="Time" type="global_time" />
			<output name="Value" type="example:value_type" />
		</requestresponse>
	</operations>

</serviceDefinition>
```

The Service comprises a single ECOA Request-Response Operation called *Request_Value* which has one input parameter (*Time* which is passed from the requesting client to the server), and one output parameter (*Value* which is the response from the server to the client). The first parameter is defined as being of type *global_time*, which is a pre-defined ECOA type. The second parameter is defined as being of type *example:value_type*, where *example* names a data types library *used* by the service definition. The data types library is, unsurprisingly, also defined in XML (file *example.types.xml*):

```
<?xml version="1.0" encoding="UTF-8"?>
<library xmlns="http://www.ecoa.technology/types-2.0">

	<types>
		<simple name="value_type" type="uint32" />
	</types>

</library>
```

The data type *example:value_type* is therefore an unsigned 32 bit integer type.

simple_lifecycle Service

The *simple_lifecycle* Service, which is provided by both the *Server* and *Client* ASCs and referenced by the *Manager* ASC, is defined in a XML file (*simple_lifecycle.interface.xml*):

```
<?xml version="1.0" encoding="UTF-8"?>
<serviceDefinition xmlns="http://www.ecoa.technology/interface-2.0">

   <use library="simple_lifecycle"/>

   <operations>
      <requestresponse name="start">
```

```
         <output name="state" type="simple_lifecycle:state"/>
      </requestresponse>

      <requestresponse name="stop">
         <output name="state" type="simple_lifecycle:state"/>
      </requestresponse>

      <requestresponse name="get_current_state">
         <output name="state" type="simple_lifecycle:state"/>
      </requestresponse>

   </operations>
</serviceDefinition>
```

The Service comprises three ECOA Request-Response Operations named *start*, *stop* and *get_current_state;* each of which has a single output parameter (*state* which is the response from the server to the client). The *state* parameter is defined as being of type simple_lifecycle:state, where *simple_lifecycle* names a data types library *used* by the service definition. The data types library is, unsurprisingly, also defined in XML (file *simple_lifecycle.types.xml*):

```
<?xml version="1.0" encoding="UTF-8"?>
<library xmlns="http://www.ecoa.technology/types-2.0">


   <types>
      <enum name="state" type="ECOA:uint32">
         <value name="IDLE" valnum="0"/>
         <value name="RUNNING" valnum="1"/>
      </enum>

   </types>
</library>
```

The data type *simple_lifecycle:state* is therefore an enumeration type (base type 32 bit unsigned integer), with the values of *IDLE* and *RUNNING*.

ECOA Module Design and Definition

The implementations of the *Server* and *Client* ASC are composed of a single ECOA Module each (Module Implementations *Server_Module_Im* and *Client_Module_Im* of Module Types *Server_Module_Type* and *Client_Module_Type* respectively) as illustrated in UML in Figure 3 and Figure 4 respectively.

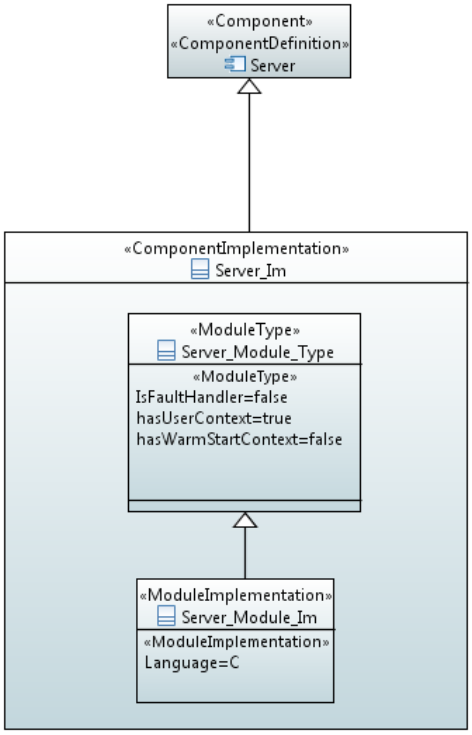

Figure 3 "Server" Module Design (as UML Composite Structure Diagram)

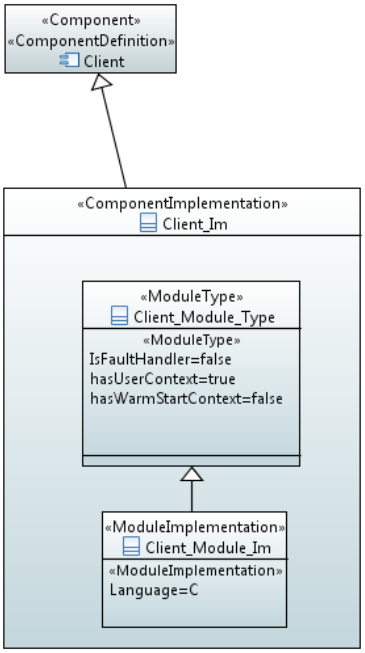

Figure 4 – "Client" Module Design (as UML Composite Structure Diagram)

The implementation of the *Manager* ASC is also composed of a single ECOA Module (Module Implementation *Manager_Module_Im* of Module Type *Manager_Module_Type* as illustrated in UML in Figure 5.

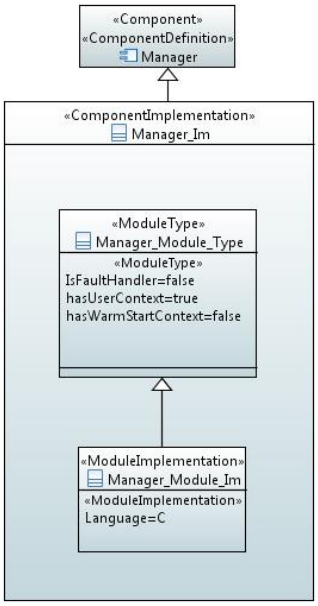

Figure 5 - "Manager" Module Design (as UML Composite Structure Diagram)

Figure 6 and Figure 7 depict in UML the internal design of the *Server* ASC (component) ***providing*** the *svc_Value* and *simple_lifecycle* ECOA Services, whilst the *Client* ASC also ***provides*** the simple_lifecycle Service but ***references*** the *svc_Value* Service. As always in the ECOA, the Module Implementations implement the Module Lifecycle operations defined by the ECOA.

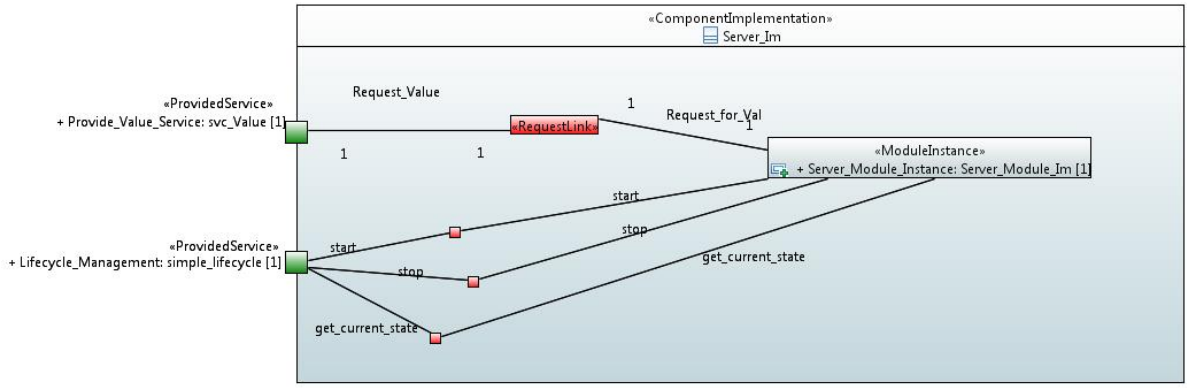

Figure 6 - "Server" Component Design (as UML Composite Structure Diagram)

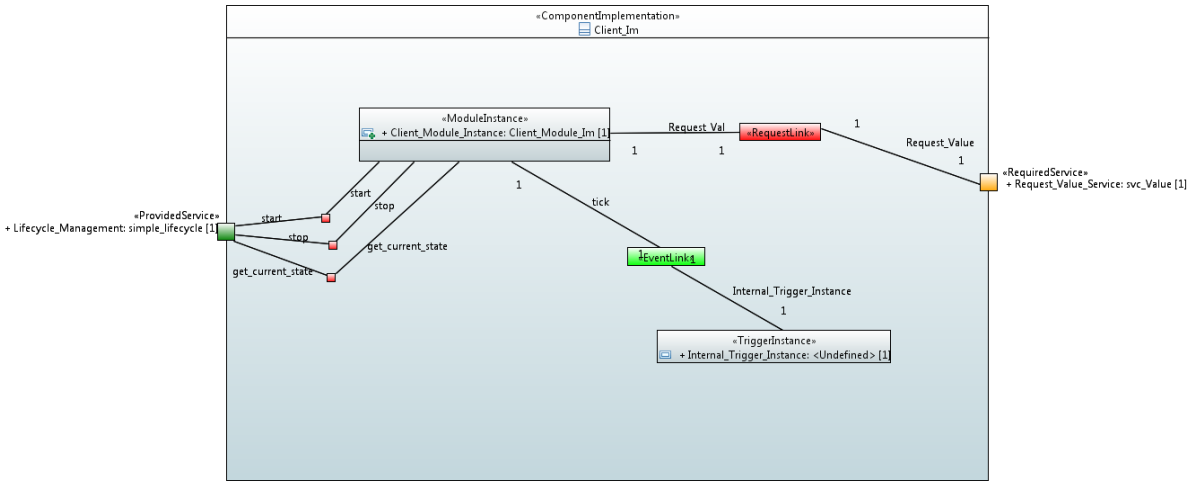

Figure 7 - "Client" Component Design (as UML Composite Structure Diagram)

Figure 8 depicts in UML the internal design of the *Manager* ASC (component) which ***references*** the two instances of the *simple_lifecycle* ECOA Service; one for managing the *Client* ASC and one for managing the *Server* ASC.

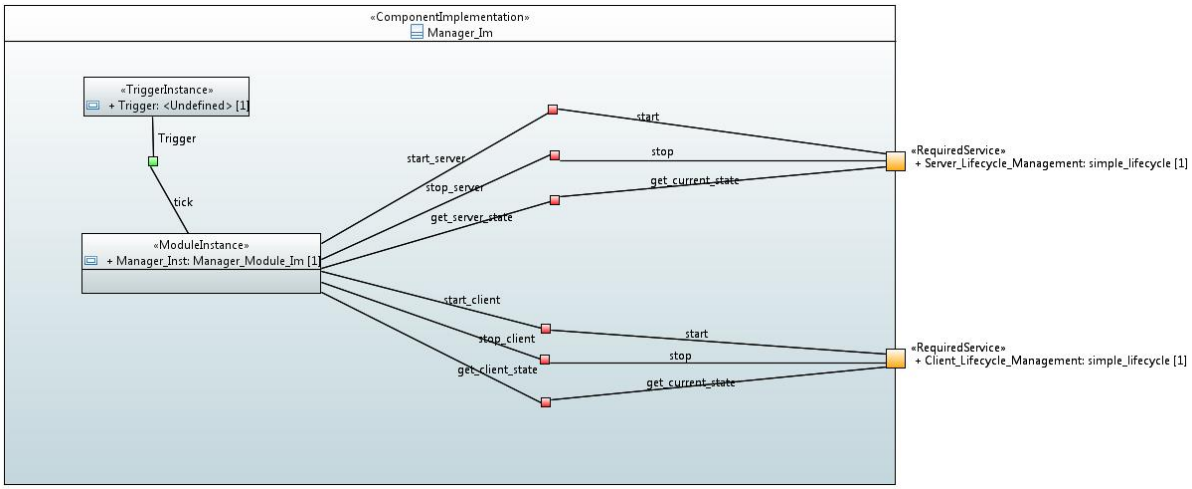

Figure 8 - "Manager" Component Design (as UML Composite Structure Diagram)

The Server ASC

The *Server* ASC is declared in XML as follows (file *Server_Im.impl.xml*):

```
<?xml version="1.0" encoding="UTF-8"?>
<componentImplementation xmlns="http://www.ecoa.technology/implementation-2.0"
   componentDefinition="Server">

   <use library="example"/>
   <use library="simple_lifecycle"/>

   <moduleType name="Server_Module_Type" hasUserContext="true"
hasWarmStartContext="false">


      <operations>

         <requestReceived name="Request_for_Val" maxConcurrentRequests="10">
            <input name="time" type="ECOA:global_time"/>
            <output name="val" type="example:value_type"/>
         </requestReceived>

         <requestReceived name="start" maxConcurrentRequests="10">
            <output name="state" type="simple_lifecycle:state"/>
         </requestReceived>

         <requestReceived name="stop" maxConcurrentRequests="10">
            <output name="state" type="simple_lifecycle:state"/>
         </requestReceived>

         <requestReceived name="get_current_state" maxConcurrentRequests="10">
            <output name="state" type="simple_lifecycle:state"/>
         </requestReceived>

      </operations>
```

```
   </moduleType>


   <moduleImplementation name="Server_Module_Im" language="C"
moduleType="Server_Module_Type"/>

   <moduleInstance name="Server_Module_Instance"
implementationName="Server_Module_Im" relativePriority="1">

   </moduleInstance>



   <requestLink>

      <clients>
         <service instanceName="Provide_Value_Service"
operationName="Request_Value"/>
      </clients>
      <server>
         <moduleInstance instanceName="Server_Module_Instance"
operationName="Request_for_Val"/>
      </server>
   </requestLink>

   <requestLink>

      <clients>
         <service instanceName="Lifecycle_Management" operationName="start"/>
      </clients>
      <server>
         <moduleInstance instanceName="Server_Module_Instance"
operationName="start"/>
      </server>
   </requestLink>

   <requestLink>

      <clients>
         <service instanceName="Lifecycle_Management" operationName="stop"/>
      </clients>
      <server>
         <moduleInstance instanceName="Server_Module_Instance"
operationName="stop"/>
      </server>
   </requestLink>

   <requestLink>

      <clients>
         <service instanceName="Lifecycle_Management"
operationName="get_current_state"/>
      </clients>
```

```
      <server>
         <moduleInstance instanceName="Server_Module_Instance"
operationName="get_current_state"/>
      </server>
   </requestLink>


</componentImplementation>
```

That is, a Module Type (*Server_Module_Type*) is declared which has four *requestReceived* operations "*Request_for_Val*", "*start*", "*stop*" and "*get_current_state*". This Module Type is implemented by a concrete Module Implementation *Server_Module_Im* which in turn is instantiated once as the Module Instance *Server_Module_Instance*.

The *<requestLink>* XML logically associates the specific concrete operations of the Module Instance with the abstract Service operations. In this example, the "*Request_for_Val*" module operation is connected to the "*Request_Value*" service operation of the "*Provide_Value_Service*" service instance and each of the "*start*", "*stop*" and "*get_current_state*" module operations are connected to the equivalent service operations of the "*Lifecycle_Management*" service instance.

A single functional code unit will be produced by the code generation process, implementing in code the concrete *Server_Module_Im* class, and named "*Server_Module_Im.c*" (assuming the Module Implementation declaration has set the *language* property to "C").

The Client ASC

The *Client* ASC is declared in XML as follows (file *Client_Im.impl.xml*):

```
<?xml version="1.0" encoding="UTF-8"?>
<componentImplementation xmlns="http://www.ecoa.technology/implementation-2.0"
   componentDefinition="Client">

   <use library="example"/>
   <use library="simple_lifecycle"/>

   <moduleType name="Client_Module_Type" hasUserContext="true"
hasWarmStartContext="false">


      <operations>

         <eventReceived name="tick">
         </eventReceived>

         <requestSent name="Request_Val" isSynchronous="true" timeout="2"
maxConcurrentRequests="10">
            <input name="time" type="ECOA:global_time"/>
            <output name="val" type="example:value_type"/>
         </requestSent>
```

```
         <requestReceived name="start" maxConcurrentRequests="10">
            <output name="state" type="simple_lifecycle:state"/>
         </requestReceived>

         <requestReceived name="stop" maxConcurrentRequests="10">
            <output name="state" type="simple_lifecycle:state"/>
         </requestReceived>

         <requestReceived name="get_current_state" maxConcurrentRequests="10">
            <output name="state" type="simple_lifecycle:state"/>
         </requestReceived>

      </operations>

   </moduleType>


   <moduleImplementation name="Client_Module_Im" language="C"
moduleType="Client_Module_Type"/>

   <moduleInstance name="Client_Module_Instance"
implementationName="Client_Module_Im" relativePriority="1">

   </moduleInstance>

   <triggerInstance name="Internal_Trigger_Instance" relativePriority="0"/>


   <eventLink>
      <senders>
         <trigger instanceName="Internal_Trigger_Instance" period="2"/>
      </senders>
      <receivers>
         <moduleInstance instanceName="Client_Module_Instance"
operationName="tick"/>
      </receivers>
   </eventLink>

   <requestLink>

      <clients>
         <moduleInstance instanceName="Client_Module_Instance"
operationName="Request_Val"/>
      </clients>
      <server>
         <reference instanceName="Request_Value_Service"
operationName="Request_Value"/>
      </server>
   </requestLink>

   <requestLink>

      <clients>
         <service instanceName="Lifecycle_Management" operationName="start"/>
```

```
      </clients>
      <server>
         <moduleInstance instanceName="Client_Module_Instance"
operationName="start"/>
      </server>
   </requestLink>

   <requestLink>

      <clients>
         <service instanceName="Lifecycle_Management" operationName="stop"/>
      </clients>
      <server>
         <moduleInstance instanceName="Client_Module_Instance"
operationName="stop"/>
      </server>
   </requestLink>

   <requestLink>

      <clients>
         <service instanceName="Lifecycle_Management"
operationName="get_current_state"/>
      </clients>
      <server>
         <moduleInstance instanceName="Client_Module_Instance"
operationName="get_current_state"/>
      </server>
   </requestLink>


</componentImplementation>
```

That is, a Module Type (*Client_Module_Type*) is declared which has five operations:

- A "*Request_Val*" *requestSent* operation;
- Three *requestReceived* operations "*stop*", "*start*" and "*get_current_state*";
- The *eventReceived* operation "*tick*".

A timeout is defined for the "*Request_Val*" operation. This is to ensure that if the response is never received, the Module will not be blocked indefinitely. This scenario may occur if the request or response is lost, or if the Server Module fails to respond

The *Internal_Trigger_Instance* Trigger Instance is introduced because the Client needs to "*periodically request a data item*" and so an ECOA periodic trigger is required. Once every period (2 seconds as set in the *<eventLink>* XML) the Trigger will fire and the Module Operation *tick* will be invoked.

This Module Type is implemented by a concrete Module Implementation *Client_Module_Im*, which in turn is instantiated once as the Module Instance *Client_Module_Instance*.

The *<requestLink>* XML logically associates the specific concrete operations of the Module Instance with the abstract Service operations. In this example, the "*Request_Val*" module operation is connected to the "*Request_Value*" service operation of the "*Request_Value_Service*" service instance and each of the "*start*", "*stop*" and "*get_current_state*" module operations are connected to the equivalent service operations of the "*Lifecycle_Management*" service instance.

A single functional code unit will be produced by the code generation process, implementing in code the concrete *Client_Module_Im* class, and named "*Client_Module_Im.c*" (assuming the Module Implementation declaration has set the *language* property to "C").

The Manager ASC

The *Manager* ASC is declared in XML as follows (file *Manager_Im.impl.xml*):

```
<?xml version="1.0" encoding="UTF-8"?>
<componentImplementation xmlns="http://www.ecoa.technology/implementation-2.0"
   componentDefinition="Manager">

   <use library="simple_lifecycle"/>

   <moduleType name="Manager_Module_Type" hasUserContext="true"
hasWarmStartContext="false">


      <operations>

         <requestSent name="start_server" isSynchronous="true" timeout="-1"
maxConcurrentRequests="10">
            <output name="state" type="simple_lifecycle:state"/>
         </requestSent>

         <requestSent name="stop_server" isSynchronous="true" timeout="-1"
maxConcurrentRequests="10">
            <output name="state" type="simple_lifecycle:state"/>
         </requestSent>

         <requestSent name="get_server_state" isSynchronous="true" timeout="-1"
maxConcurrentRequests="10">
            <output name="state" type="simple_lifecycle:state"/>
         </requestSent>

         <requestSent name="start_client" isSynchronous="true" timeout="-1"
maxConcurrentRequests="10">
            <output name="state" type="simple_lifecycle:state"/>
         </requestSent>

         <requestSent name="stop_client" isSynchronous="true" timeout="-1"
maxConcurrentRequests="10">
            <output name="state" type="simple_lifecycle:state"/>
         </requestSent>
```

```
         <requestSent name="get_client_state" isSynchronous="true" timeout="-1"
maxConcurrentRequests="10">
            <output name="state" type="simple_lifecycle:state"/>
         </requestSent>

         <eventReceived name="tick">
         </eventReceived>

      </operations>

   </moduleType>


   <moduleImplementation name="Manager_Module_Im" language="C"
moduleType="Manager_Module_Type"/>

   <moduleInstance name="Manager_Inst" implementationName="Manager_Module_Im"
relativePriority="9">

   </moduleInstance>

   <triggerInstance name="Trigger" relativePriority="10"/>


   <eventLink>
      <senders>
         <trigger instanceName="Trigger" period="5"/>
      </senders>
      <receivers>
         <moduleInstance instanceName="Manager_Inst" operationName="tick"/>
      </receivers>
   </eventLink>

   <requestLink>

      <clients>
         <moduleInstance instanceName="Manager_Inst"
operationName="start_server"/>
      </clients>
      <server>
         <reference instanceName="Server_Lifecycle_Management"
operationName="start"/>
      </server>
   </requestLink>

   <requestLink>

      <clients>
         <moduleInstance instanceName="Manager_Inst" operationName="stop_server"/>
      </clients>
      <server>
         <reference instanceName="Server_Lifecycle_Management"
operationName="stop"/>
```

```
      </server>
   </requestLink>

   <requestLink>

      <clients>
         <moduleInstance instanceName="Manager_Inst"
operationName="get_server_state"/>
      </clients>
      <server>
         <reference instanceName="Server_Lifecycle_Management"
operationName="get_current_state"/>
      </server>
   </requestLink>

   <requestLink>

      <clients>
         <moduleInstance instanceName="Manager_Inst"
operationName="start_client"/>
      </clients>
      <server>
         <reference instanceName="Client_Lifecycle_Management"
operationName="start"/>
      </server>
   </requestLink>

   <requestLink>

      <clients>
         <moduleInstance instanceName="Manager_Inst" operationName="stop_client"/>
      </clients>
      <server>
         <reference instanceName="Client_Lifecycle_Management"
operationName="stop"/>
      </server>
   </requestLink>

   <requestLink>

      <clients>
         <moduleInstance instanceName="Manager_Inst"
operationName="get_client_state"/>
      </clients>
      <server>
         <reference instanceName="Client_Lifecycle_Management"
operationName="get_current_state"/>
      </server>
   </requestLink>


</componentImplementation>
```

That is, a Module Type (*Manager_Module_Type*) is declared which has seven operations:

- Three *requestReceived* operations related to the management of the *Server* ASC "*start_server*", "*stop_server*" and "*get_server_state*";
- Three *requestReceived* operations related to the management of the *Client* ASC "*start_client*", "*stop_client*" and "*get_client_state*";
- The *eventReceived* operation "*tick*".

The *Trigger* Trigger Instance is introduced because the Manager will request the managed components to change functional states at various points. Once every period (5 seconds as set in the *<eventLink>* XML) the Trigger will fire and the Module Operation *tick* will be invoked.

This Module Type is implemented by a concrete Module Implementation *Manager_Module_Im*, which in turn is instantiated once as the Module Instance *Manager_Inst*.

The *<requestLink>* XML logically associates the specific concrete operations of the Module Instance with the abstract Service operations. In this example, the "*start_server*", "*stop_server*" and "*get_server_state*" module operations are connected to the relevant service operation of the "Server_Lifecycle_Management" service instance and each of the "*start_client*", "*stop_client*" and "*get_client_state*" module operations are connected to the relevant service operations of the "*Client_Lifecycle_Management*" service instance.

A single functional code unit will be produced by the code generation process, implementing in code the concrete *Manager_Module_Im* class, and named "*Manager_Module_Im.c*" (assuming the Module Implementation declaration has set the *language* property to "C").

ECOA Deployment Definition

The ECOA "*Simple Lifecycle Example*" Assembly is deployed (that is, the declared Module and Trigger Instances are allocated to a single ECOA Protection Domain, which is then allocated to a computing node) by the following XML (file *example.deployment.xml*):

```
<deployment xmlns="http://www.ecoa.technology/deployment-2.0"
finalAssembly="example" logicalSystem="example">

   <protectionDomain name="Ex1">
      <executeOn computingPlatform="Example_Platform" computingNode="card1_bae"/>

      <deployedModuleInstance componentName="Client_Inst"
moduleInstanceName="Client_Module_Instance" modulePriority="11"/>
      <deployedTriggerInstance componentName="Client_Inst"
triggerInstanceName="Internal_Trigger_Instance" triggerPriority="12"/>
      <deployedModuleInstance componentName="Server_Inst"
moduleInstanceName="Server_Module_Instance" modulePriority="3"/>
      <deployedModuleInstance componentName="Manager_Inst"
moduleInstanceName="Manager_Inst" modulePriority="10"/>
```

```
      <deployedTriggerInstance componentName="Manager_Inst"
triggerInstanceName="Trigger" triggerPriority="9"/>
   </protectionDomain>

   <platformConfiguration faultHandlerNotificationMaxNumber="8"
computingPlatform="Example_Platform"></platformConfiguration>

</deployment>
```

Thus in this case, a single ECOA Protection Domain is declared (*Ex1*) executing on an ECOA Computing Node, on a single ECOA Computing Platform.

Implementation

The Server ASC

The "Request_Value_Service" Service request handler is implemented by the code function *Server_SM_Im__Request_for_Val__request_received*:

```
void
Server_Module_Im__Request_for_Val__request_received(Server_Module_Im__context*
context, const ECOA__uint32 ID, const ECOA__global_time* time)
{
   if (context->user.state == simple_lifecycle__state_RUNNING)
   {
      ECOA__return_status return_status;
      return_status =
Server_Module_Im_container__Request_for_Val__response_send(context, ID, 10);
   }
   else
   {
      ECOA__log log;
      log.current_size = sprintf((char *) &log.data, "server received
Request_for_Val when IDLE - ignoring request!");
      Server_Module_Im_container__log_info(context, log);
   }
}
```

This function replies to the request (only if the component is in the functional "*RUNNING*" state) with a data value of *10* by invoking the ECOA Container API function *Server_Module_Im_container__Request_for_Val__response_send*.

The functional state of the Component is handled in the following functions:

```
void Server_Module_Im__start__request_received
   (Server_Module_Im__context* context,
    const ECOA__uint32 ID)
{
   ECOA__return_status status;
```

```
   context->user.state = simple_lifecycle__state_RUNNING;

   status = Server_Module_Im_container__start__response_send(context, ID, context-
>user.state);
}

void Server_Module_Im__stop__request_received
   (Server_Module_Im__context* context,
    const ECOA__uint32 ID)
{
   ECOA__return_status status;

   context->user.state = simple_lifecycle__state_IDLE;

   status = Server_Module_Im_container__stop__response_send(context, ID, context-
>user.state);
}

void Server_Module_Im__get_current_state__request_received
   (Server_Module_Im__context* context,
    const ECOA__uint32 ID)
{
   ECOA__return_status status;

   status = Server_Module_Im_container__get_current_state__response_send(context,
ID, context->user.state);
}
```

When a request is received, the component updates its functional state as appropriate and responds to the request with the updated state. The current state is stored in the user context (state data for the module), which is initialised as follows:

```
void Server_Module_Im__INITIALIZE__received(Server_Module_Im__context *context)
{
   context->user.state = simple_lifecycle__state_IDLE;
}
```

The Client ASC

The main functionality occurs when the *Client* receives the "*tick*" event from the *Internal_Trigger_Instance*, i.e. to populate the *Client_SM_Im__tick__received* function stub.

```
void Client_Module_Im__tick__received(Client_Module_Im__context *context)
{
   ECOA__log log;

   if (context->user.state == simple_lifecycle__state_RUNNING)
   {
      ECOA__global_time time;
      ECOA__return_status return_status;
      example__value_type val;
```

```
      return_status =
Client_Module_Im_container__get_absolute_system_time(context, &time);

      val = 0;

      log.current_size = sprintf((char *) &log.data, "val before request = %d",
val);
      Client_Module_Im_container__log_info(context, log);

      return_status =
Client_Module_Im_container__Request_Val__request_sync(context, &time, &val);

      log.current_size = sprintf((char *) &log.data, "val from response = %d",
val);
      Client_Module_Im_container__log_info(context, log);
   }
   else
   {
      log.current_size = sprintf((char *) &log.data, "client received tick - not
doing any functional work as IDLE");
      Client_Module_Im_container__log_info(context, log);
   }
}
```

That is, the *val* variable is zeroed and logged prior to invoking the *Client_Module_Im_container__Request_Val__request_sync* API (only if the component is in the functional "*RUNNING*" state), and because a synchronous Request-Response call is made, the response (in variable *val*) is immediately available to log.

The functional state of the Component is handled (in the same manner as the *Server* ASC) in the following functions:

```
void Client_Module_Im__start__request_received
   (Client_Module_Im__context* context,
    const ECOA__uint32 ID)
{
   ECOA__return_status status;

   context->user.state = simple_lifecycle__state_RUNNING;

   status = Client_Module_Im_container__start__response_send(context, ID, context-
>user.state);
}

void Client_Module_Im__stop__request_received
   (Client_Module_Im__context* context,
    const ECOA__uint32 ID)
{
   ECOA__return_status status;

   context->user.state = simple_lifecycle__state_IDLE;
```

```
   status = Client_Module_Im_container__stop__response_send(context, ID, context-
>user.state);
}

void Client_Module_Im__get_current_state__request_received
   (Client_Module_Im__context* context,
    const ECOA__uint32 ID)
{
   ECOA__return_status status;

   status = Client_Module_Im_container__get_current_state__response_send(context,
ID, context->user.state);
}
```

The state is also initialised in the same manner as the *Server* ASC:

```
void Client_Module_Im__INITIALIZE__received(Client_Module_Im__context *context)
{
   context->user.state = simple_lifecycle__state_IDLE;
}
```

The Manager ASC

The main functionality occurs when the *Manager* receives the "*tick*" event from the *Trigger*, i.e. to populate the *Manager_Module_Im__tick__received* function stub.

```
void Manager_Module_Im__tick__received(Manager_Module_Im__context *context)
{
   ECOA__return_status status;
   simple_lifecycle__state serverState;
   simple_lifecycle__state clientState;
   ECOA__log log;

   if (context->user.count == 1)
   {
      // Start both server and client.
      log.current_size = sprintf((char *) &log.data, "manager requesting client
and server components to start");
      Manager_Module_Im_container__log_info(context, log);

      status = Manager_Module_Im_container__start_server__request_sync(context,
&serverState);
      status = Manager_Module_Im_container__start_client__request_sync(context,
&serverState);
   }
   else if (context->user.count == 2)
   {
      // Stop the server.
      log.current_size = sprintf((char *) &log.data, "manager requesting server
component to stop");
      Manager_Module_Im_container__log_info(context, log);
```

```
      status = Manager_Module_Im_container__stop_server__request_sync(context,
&serverState);
   }
   else if (context->user.count == 3)
   {
      // Stop the client.
      log.current_size = sprintf((char *) &log.data, "manager requesting client
component to stop");
      Manager_Module_Im_container__log_info(context, log);

      status = Manager_Module_Im_container__stop_client__request_sync(context,
&serverState);
   }
   else
   {
      context->user.count = 0;
   }

   context->user.count++;
}
```

This function performs various lifecycle state change requests on the Server and Client ASCs depending on the iteration count (a variable held in the user context).

1. Iteration count 1:
    a. The *Manager* requests both the *Client* and *Server* ASCs to start.
2. Iteration count 2:
    a. The *Manager* requests both the *Server* ASC to stop.
3. Iteration count 3:
    a. The *Manager* requests both the *Client* ASC to stop.

This sequence is continually repeated, as count variable is reset to 0 once it reaches 4. The count variable is initialised as follows:

```
void Manager_Module_Im__INITIALIZE__received(Manager_Module_Im__context *context)
{
   context->user.count = 0;
}
```

Program Output

When the ECOA "*Simple Lifecycle Example*" Assembly is built and run (in a single Node deployment), an output similar to Figure 9 should be achieved.

When the system first starts, the *Client* ASC is in *IDLE* state therefore it does not request values from the *Server*. After the first period (5 seconds), the *Manger* requests both the *Server* and *Client* to start. Once started the *Client* ASC enters begins performing its functional requirements

(requesting value from the *Server*). The *Client* ASC outputs, at each iteration (2 seconds), both the value before sending the request message, and the value after receiving the response (value is 10).

After the next period (5 seconds), the *Manager* requests the *Server* to stop. At this point, the *Client* is functionally *RUNNING*, whereas the *Server* is functionally *IDLE*. The *Client* continues to send requests to the *Server*, but the *Server* does not handle the requests. The *Clients* request will timeout after 2 seconds; at which point it will print the return value (0 as the default value for the type).

After the next period (5 seconds), the *Manager* requests the Client to stop. The *Client* will then stop sending the requests to the *Server*.

This behaviour is continually repeated.

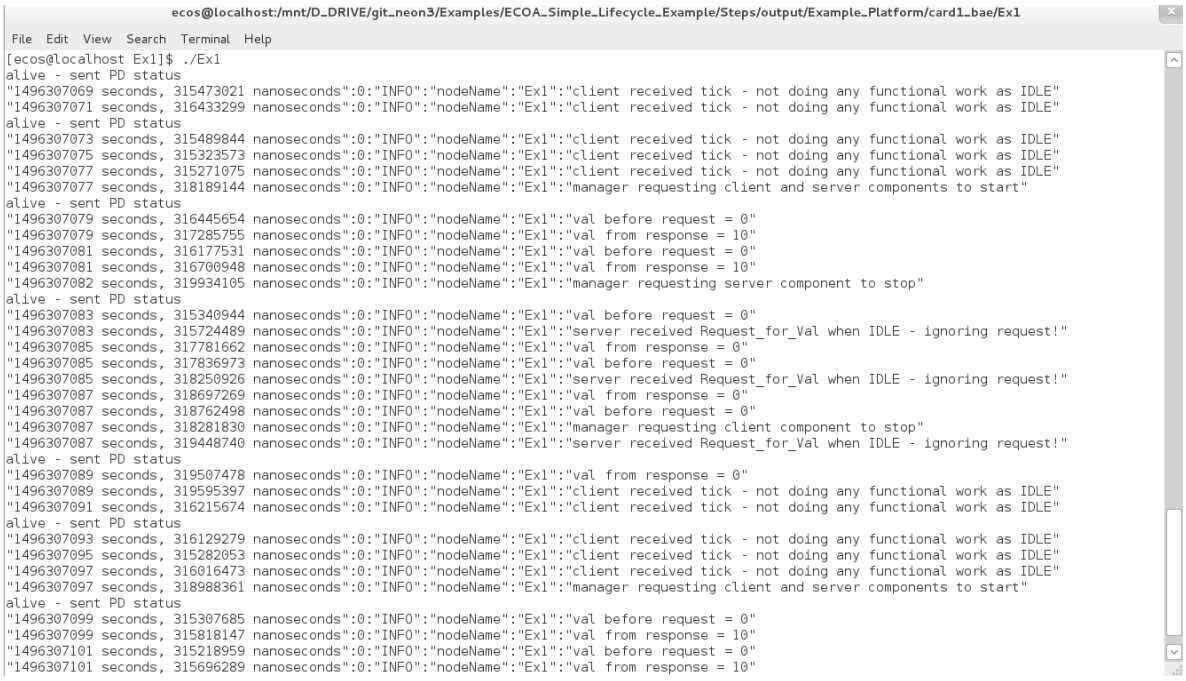

Figure 9 - ECOA "*Simple Lifecycle Example*" in Execution

References

1	European Component Oriented Architecture (ECOA) Collaboration Programme: Architecture Specification (Parts 1 to 11) "ECOA" is a registered trade mark.
2	European Component Oriented Architecture (ECOA®) Collaboration Programme: Guidance Document: System Management
3	Simple Example. *http://www.ecoa.technology/tutorials.html*