BAE SYSTEMS
INSPIRED WORK

GimmeGimmeGimme

Introduction

This document describes an ECOA® client-server example, named "GimmeGimmeGimme".

The client-server model (ref.[2]) is one of the most basic data, task, or workload, distribution mechanisms in computing. Clients and servers may be distributed across a network, or they may reside on the same computing system. Service oriented concepts, which form a basis behind the ECOA, naturally fit with the client-server model, the clients referencing (using) the services provided by the server. Service orientation, and therefore the ECOA, goes on a step extra, in that a component can be a client (service user) to one or more other components, whilst simultaneously being a *server* (service provider) to others.

This document presents the principal user generated artefacts required to create the "*GimmeGimmeGimme*" client-server 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 "*GimmeGimmeGimme*" client-server example is intended to demonstrate a minimum effort example of data distribution from a single data server to multiple data-accessing clients. Hence the example's name – "Gimme data (and lots of it)".

ECOA Features Exhibited

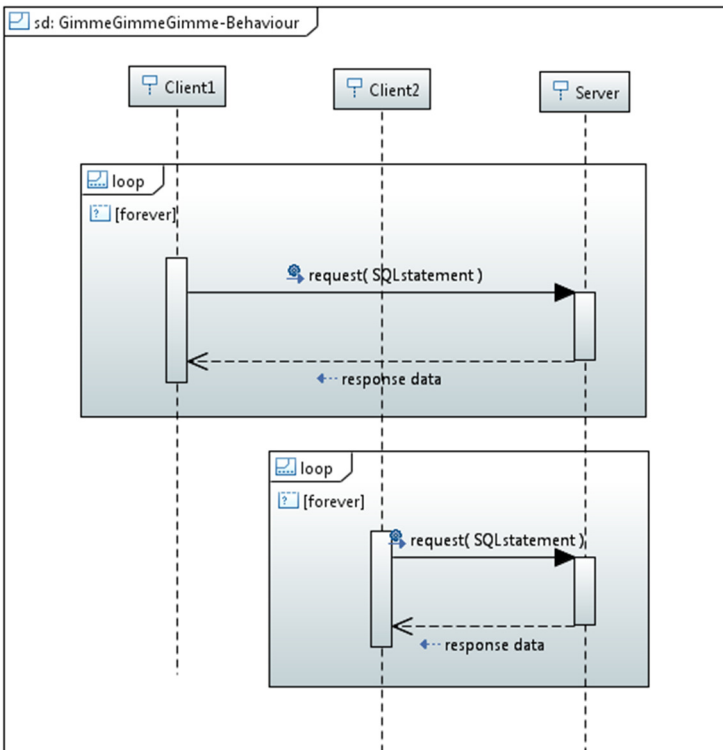
- Composition of an ECOA Assembly of multiple ECOA ASCs (components).
- Contention-free resource sharing within an ECOA Assembly.
- Multiple cooperating ECOA Protection Domains.
- Use of the ECOA Logical Interface (ELI) between ECOA Protection Domains.
- Service Availability.
- Module Lifecycle Management.
- Use of the ECOA runtime logging API.

Design and Definition

Client-Server Functional Design

The "GimmeGimmeGimme" client-server example will simply demonstrate a basic remote data access and retrieval mechanism. Each client will periodically request a data item from the server and will receive a data item in return (Figure 1).

Figure 1 ECOA "*GimmeGimmeGimme*" Client-Server Example Behaviour



For the current purposes, the data content of the request and the response does not matter, but will simulate a database access. The request message will therefore be in the form of an SQL statement, and the response will be in the form of a numerical data item (expressed in text).

The SQL statement send will be of the form:

*select * from iTable where <X> = <seq>*

where *<X>* is either "A" for Client1, or "B" for Client2, and *<seq>* is a sequence number increasing by one each time a request by the particular client is made.

The response returned by the Server will be of the form:

<X>[<seq>]: <num

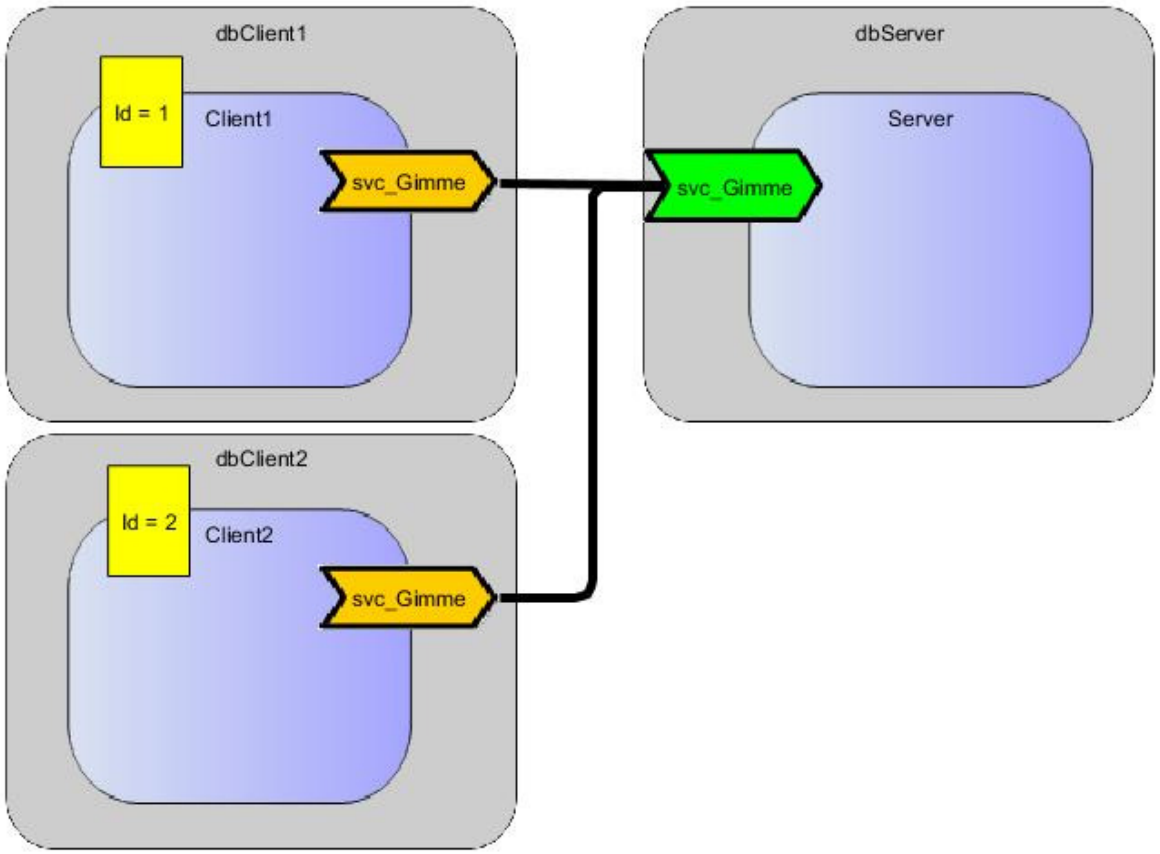
where *<X>* is either "A" or "B" depending on which client made the request, *<seq>* is the sequence number provided with the request, and *<num>* is the total number of requests handled by the server

(from any client) so far. This response form should allow the response stream to be analysed to check that the client requests are being handled by the server equally and correctly.

ECOA Assembly Design and Definition

This ECOA "*GimmeGimmeGimme*" client-server example ECOA Assembly comprises three ECOA ASCs named "*Client1*", "*Client2*" (which are instances of the same ASC type) and "*Server*". The "*Client*" ASC type is instantiated twice within the ECOA Assembly, once as "*Client1*" with the ECOA Property "*Id*" set to "1", and once as "*Client2*" with it set to "2". The "*Server*" ASC provides the "*svc_Gimme*" ECOA Service, which is referenced (used) by the two "*Client*" ASCs. Each ASC is allocated to a separate ECOA Protection Domain so as to invoke the ELI in passing data between the ASCs, and to represent networked clients accessing a remote server.

Figure 2 ECOA "*GimmeGimmeGimme*" 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 "*GimmeGimmeGimme*" Assembly is as follows (file *GimmeGimmeGimme_impl.composite*):

```
<csa:composite xmlns:csa="http://docs.oasis-open.org/ns/opencsa/sca/200912"
      xmlns:ecoa-sca="http://www.ecoa.technology/sca"
```

```
        name="GimmeGimmeGimme_impl"
        targetNamespace="http://www.ecoa.technology/sca">
        <!-- -->
        <csa:component name="Server">
             <ecoa-sca:instance componentType="Server">
                  <ecoa-sca:implementation name="Server"/>
             </ecoa-sca:instance>
             <csa:service name="svc_Gimme"/>
        </csa:component>
        <!-- -->
        <csa:component name="Client1">
             <ecoa-sca:instance componentType="Client">
                  <ecoa-sca:implementation name="Client"/>
             </ecoa-sca:instance>
             <csa:reference name="svc_Gimme"/>
             <csa:property name="Id"><csa:value>1</csa:value></csa:property>
        </csa:component>
        <!-- -->
        <csa:component name="Client2">
             <ecoa-sca:instance componentType="Client">
                  <ecoa-sca:implementation name="Client"/>
             </ecoa-sca:instance>
             <csa:reference name="svc_Gimme"/>
             <csa:property name="Id"><csa:value>2</csa:value></csa:property>
        </csa:component>
        <!-- -->
        <!-- System Wiring...  -->
        <csa:wire source="Client1/svc_Gimme" target="Server/svc_Gimme"
                                                    ecoa-sca:rank="1"/>
        <csa:wire source="Client2/svc_Gimme" target="Server/svc_Gimme"
                                                    ecoa-sca:rank="1"/>
   </csa:composite>
```

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

```
   <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">
        <service name="svc_Gimme">
             <ecoa-sca:interface syntax="svc_Gimme" qos="Required-svc_Gimme"/>
        </service>
   </componentType>
```

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

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

```
<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">
     <reference name="svc_Gimme">
          <ecoa-sca:interface syntax="svc_Gimme" qos="Required-svc_Gimme"/>
     </reference>
     <property name="Id" type="xs:string" ecoa-sca:type="int32"/>
</componentType>
```

This ASC definition (the *<componentType>* XML element) declares a reference (by the ASC) to the *svc_Gimme* ECOA Service, and it also defines the *Id* ECOA Property. Note that the Property *value* is given in the Assembly declaration (above), not here in the ASC type definition.

In the above ASC definitions a "*qos*" attribute is declared. "Quality of Service" (QoS) in ECOA is a matter on on-going work and definition. For the present, this attribute is optional and is ignored. It is included in this example only as an illustration.

ECOA Service and Types Definition

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

```
<serviceDefinition xmlns="http://www.ecoa.technology/interface-1.0">
     <use library="sql"/>
     <operations>
          <requestresponse name="Gimme">
               <input name="statement" type="sql:string"/>
               <output name="data" type="sql:string"/>
          </requestresponse>
     </operations>
</serviceDefinition>
```

The Service comprises a single ECOA Request-Response Operation called *Gimme* which has one input parameter (*statement* which is passed from the requesting client to the server), and one output parameter (*data* which is the response from the server to the client). These parameters are both defined as being of data type *sql:string*, where *sql* names a data types library *used* by the service definition. The data types library is, unsurprisingly, also defined in XML (file *sql.types.xml*):

```
<library xmlns="http://www.ecoa.technology/types-1.0">
     <types>
          <array name="string" maxNumber="65536" itemType="char8" />
     </types>
</library>
```

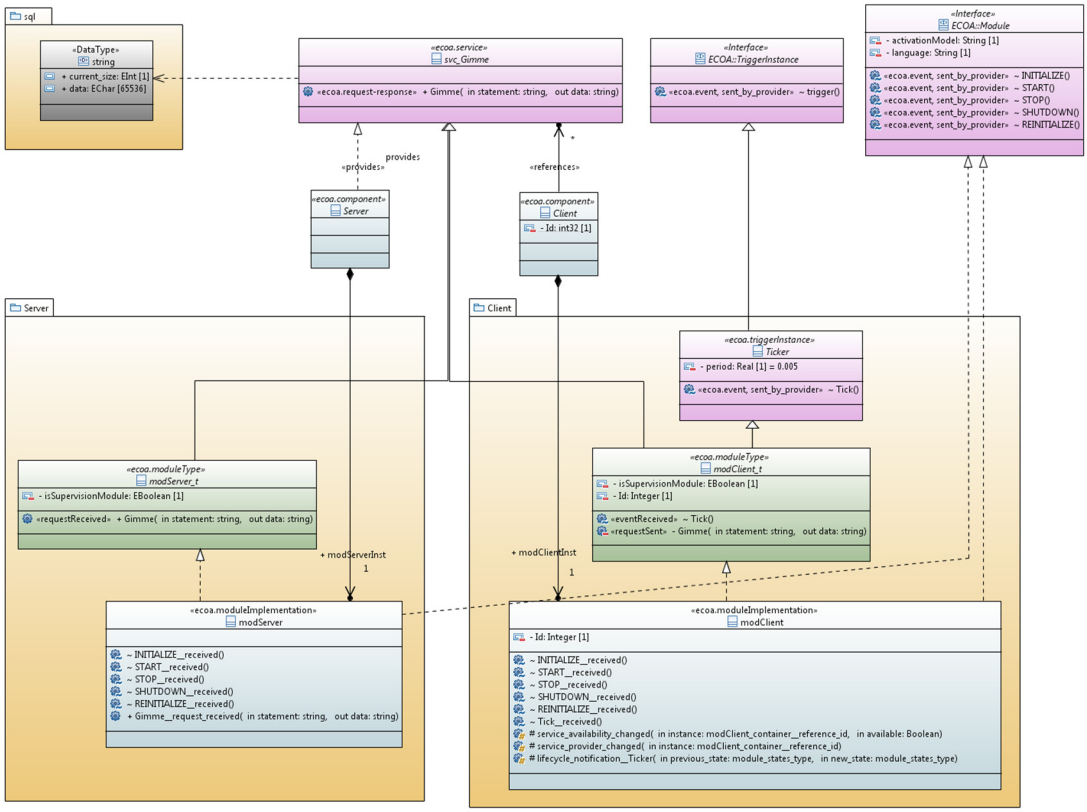
The data type *sql:string* is therefore an array of (up to) 65536 8-bit characters.

ECOA Module Design and Definition

The *Server* and *Client* ASC (component) types are composed of a single ECOA Module each (Module Implementations *modServer* and *modClient* of Module Types *modServer_t* and *modClient_t* respectively) as illustrated in UML in Figure 3. Here is depicted in UML the *Server* ASC

(component) ***providing*** the *svc_Gimme* ECOA Service, whilst the *Client* ASC ***references*** the Service, and possesses the ECOA Property *Id*. As always in the ECOA, the Module Implementations implement the Module Lifecycle operations defined by the ECOA (as represented in UML by the interface class *ECOA::Module*).

Figure 3 ECOA "*GimmeGimmeGimme*" Module Design (as UML Class Diagram)



The Server ASC

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

```
<componentImplementation xmlns="http://www.ecoa.technology/implementation-
    1.0" componentDefinition="Server">
    <use library="sql"/>
    <!-- -->
    <moduleType name="modServer_t" isSupervisionModule="true">
        <operations>
            <requestReceived name="Gimme">
                <input name="statement" type="sql:string"/>
                <output name="data" type="sql:string"/>
            </requestReceived>
        </operations>
    </moduleType>
    <!-- -->
    <moduleImplementation name="modServer" moduleType="modServer_t"
                          activationModel="reactive" language="C" />
    <!-- -->
```

```
        <moduleInstance name="modServerInst"
                        implementationName="modServer"
                        relativePriority="1"/>
        <!-- -->
        <requestLink>
            <clients>
                <service instanceName="svc_Gimme" operationName="Gimme"/>
            </clients>
            <server>
                <moduleInstance instanceName="modServerInst"
                                operationName="Gimme"/>
            </server>
        </requestLink>
    </componentImplementation>
```

That is, a Module Type (*modServer_t*) is declared which has a *requestReceived* operation "*Gimme*" inherited from the ECOA Service (depicted by the UML *generalization* association). This Module Type is implemented by a concrete Module Implementation *modServer* (depicted in the UML expanded in the form of the code class produced by the code generation process), which in turn is instantiated at runtime as the Module Instance *modServerInst*.

The *<requestLink>* XML logically associates the specific concrete operations of the runtime Module Instance with the abstract Service operations.

A single functional code unit will be produced by the code generation process, implementing in code the concrete *modServer* class, and named "*modServer.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.impl.xml*):

```
    <componentImplementation xmlns="http://www.ecoa.technology/implementation-
        1.0" componentDefinition="Client">
        <use library="sql"/>
        <!-- -->
        <moduleType name="modClient_t" isSupervisionModule="true">
          <properties>
            <property name="Id" type="int32"/>
          </properties>
          <operations>
            <eventReceived name="Tick" />
            <requestSent name="Gimme" isSynchronous="true" timeout="-1.0">
              <input name="statement" type="sql:string"/>
              <output name="data" type="sql:string"/>
            </requestSent>
          </operations>
        </moduleType>
```

```
        <!-- -->
        <moduleImplementation  name="modClient" moduleType="modClient_t"
                                    activationModel="reactive" language="C" />
        <!-- -->
        <moduleInstance name="modClientInst"
                                          implementationName="modClient"
                                          relativePriority="1">
          <propertyValues>
            <propertyValue name="Id">$Id</propertyValue>
          </propertyValues>
        </moduleInstance>
        <!-- -->
        <triggerInstance name="Ticker" />
        <!-- -->
         <requestLink>
          <clients>
            <moduleInstance instanceName="modClientInst"
                                                 operationName="Gimme"/>
          </clients>
          <server>
            <reference instanceName="svc_Gimme" operationName="Gimme"/>
          </server>
        </requestLink>
        <!-- -->
        <eventLink>
          <senders>
            <trigger instanceName="Ticker" period="0.005" />
          </senders>
          <receivers>
            <moduleInstance instanceName="modClientInst"
                                                  operationName="Tick"/>
          </receivers>
        </eventLink>
    </componentImplementation>
```

That is, a Module Type (*modClient_t*) is declared which has two operations:

- A "*Gimme*" *requestSent* operation inherited from the ECOA Service.
- The *eventReceived* operation "*Tick*" inherited from the "*Ticker*" Trigger Instance;

The *Ticker* 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 (0.005 seconds as set in the *<eventLink>* XML[1]) the Trigger will fire and the Module Operation *Tick* will be invoked.

This Module Type is implemented by a concrete Module Implementation *modClient* (depicted in the UML expanded in the form of the code class produced by the code generation process), which in turn is instantiated at runtime as the Module Instance *modClientInst*.

[1] The UML does not explicitly depict Service Links. The period attribute is therefore depicted as a UML property of the *«ecoa.triggerInstance»* UML interface class.

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

Because the Client ASC references an ECOA Service, the concrete Module Implementation will require additional "housekeeping" operations to manage the referenced Service Availability, and also to manage the Module Lifecycle of the employed Trigger Instance. These too have been added to the UML depiction of the Module Implementation.

ECOA Deployment Definition

The ECOA "*GimmeGimmeGimme*" Assembly is deployed (that is, the declared Module and Trigger Instances are allocated to ECOA Protection Domains, which are themselves allocated to computing nodes) by the following XML (file *deployment.xml*):

```
<deployment xmlns="http://www.ecoa.technology/deployment-1.0"
     finalAssembly="GimmeGimmeGimme_impl"
     logicalSystem="hostbased_logical_system">
     <!-- -->
     <protectionDomain name="dbServer">
          <executeOn computingNode="env1" computingPlatform="host1"/>
          <deployedModuleInstance componentName="Server"
               moduleInstanceName="modServerInst" modulePriority="50"/>
     </protectionDomain>
     <!-- -->
     <protectionDomain name="dbClient1">
          <executeOn computingNode="env2" computingPlatform="host2"/>
          <deployedModuleInstance componentName="Client1"
               moduleInstanceName="modClientInst" modulePriority="50"/>
          <deployedTriggerInstance componentName="Client1"
               triggerInstanceName="Ticker" triggerPriority="50"/>
     </protectionDomain>
     <!-- -->
     <protectionDomain name="dbClient2">
          <executeOn computingNode="env3" computingPlatform="host3"/>
          <deployedModuleInstance componentName="Client2"
               moduleInstanceName="modClientInst" modulePriority="50"/>
          <deployedTriggerInstance componentName="Client2"
               triggerInstanceName="Ticker" triggerPriority="50"/>
     </protectionDomain>
</deployment>
```

Thus in this case, three separate ECOA Protection Domains are declared (*dbClient1*, *dbClient2*, and *dbServer*) each executing on a separate ECOA Computing Node, each of which is in a different ECOA Computing Platform. This deployment therefore potentially represents three independent, networked computing resources with one running the server, and two running a client each (as represented as a UML Deployment Diagram in Figure 4).

More often, the three Protection Domains (in such a simple example) would be hosted on a common Computing Node (such as a desktop PC) as in Figure 5. This is accomplished by modifying only the Transport Binding XML, and then re-generating and rebuilding the ECOA Software Platform code.

Figure 4 ECOA "*GimmeGimmeGimme*" Deployment Across a Network

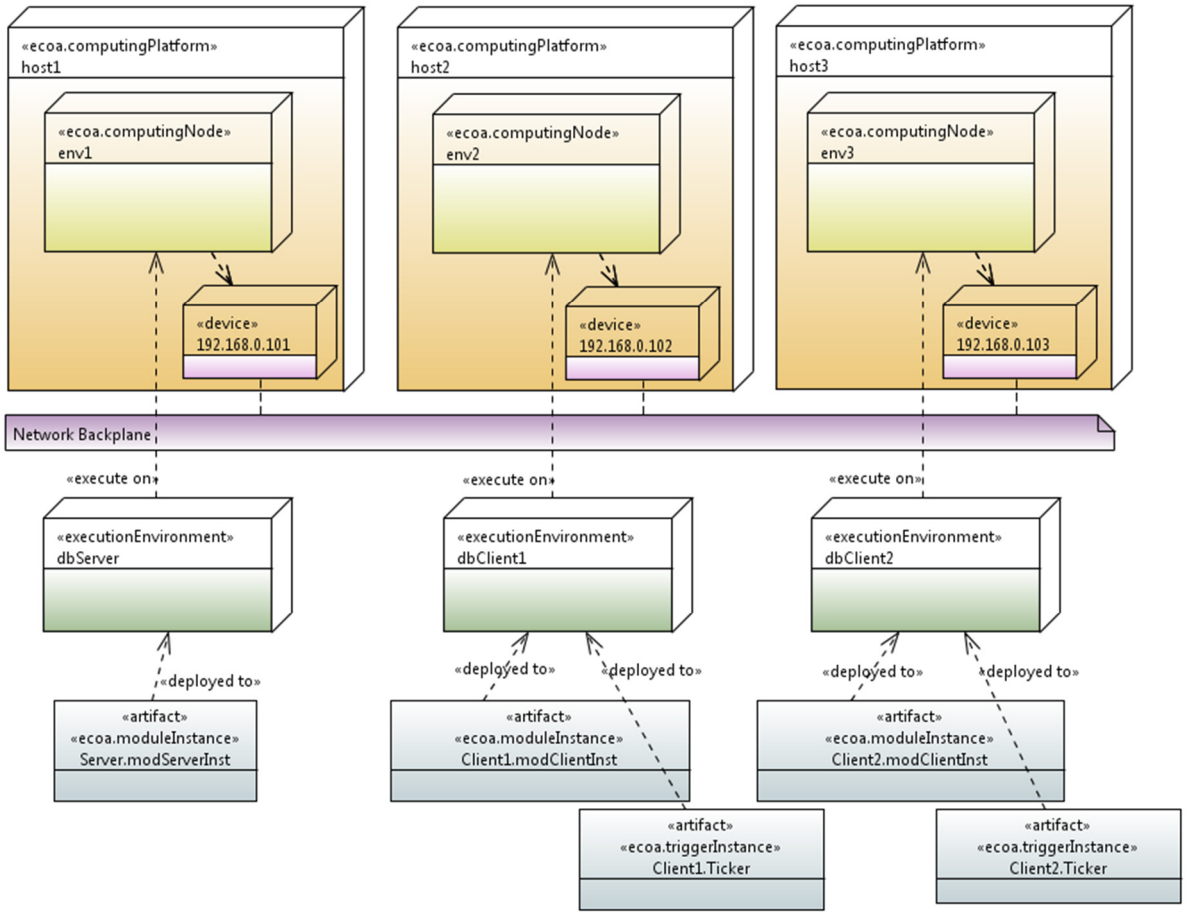
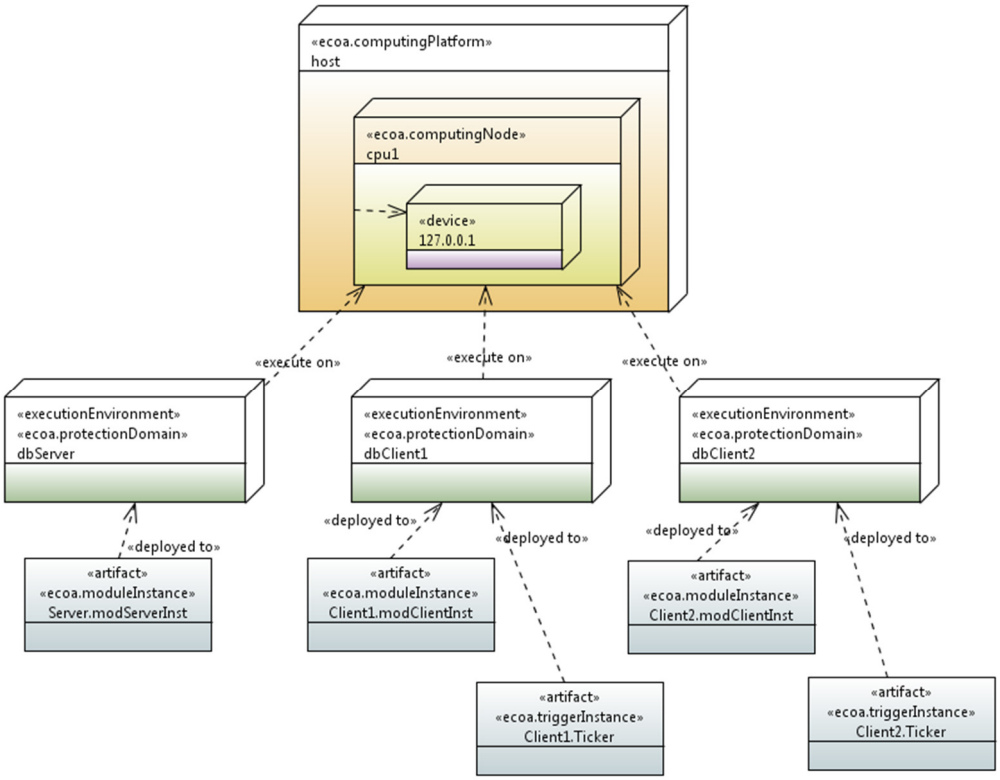


Figure 5 ECOA "*GimmeGimmeGimme*" Deployment on a Single Node



The UDP Transport Binding (file *udpbinding.xml*) for the multi-Node case (as in Figure 4) might be:

```
<ecoa:UDPBinding xmlns:ecoa="http://www.ecoa.technology/udpbinding-1.0">
     <ecoa:platform name="host1"
                    receivingMulticastAddress="192.168.0.101"
                    receivingPort="60424"
                    platformId="1"/>
     <ecoa:platform name="host2"
                    receivingMulticastAddress="192.168.0.102"
                    receivingPort="60425"
                    platformId="2"/>
     <ecoa:platform name="host3"
                    receivingMulticastAddress="192.168.0.103"
                    receivingPort="60426"
                    platformId="3"/>
</ecoa:UDPBinding>
```

where the *receivingMulticastAddress* is set to each platform's own unique IP Address.

For the single Node deployment, the local loop-back address "127.0.0.1" would be used for all *receivingMulticastAddress* entries, which means that data passes directly between the Protection Domains.

ECOA Lifecycle and Service Availability Considerations

Since we have introduced a Trigger Instance into the *Client* ASC, it is necessary that the Supervision Module (see ref.[1]) of that ASC (i.e. the Module Implementation *modClient*) manage its lifecycle.

The management is generic and comprises:

- invoking an **INITIALIZE** ECOA Event Operation to the Trigger Instance to bring its Module Lifecycle state from **IDLE** to **READY**;
- invoking a **START** ECOA Event Operation to the Trigger Instance once its Module Lifecycle state is **READY.**

No other actions (such as invoking **STOP** or **SHUTDOWN**) are required in the present simple example.

Since the *Server* ASC *provides* an ECOA Service (*svc_Gimme*) it is necessary that the Service be declared (at runtime) as "available". For the present simple example, this will be done when the *Server* ASC's Supervision Module (namely the Module Implementation *modServer*) receives a **START** Event Operation. No error conditions are defined, so once set, the *svc_Gimme* Service will always be "available".

Implementation

The Server ASC

On invocation of the *START* operation, the *Server* ASC will announce its presence, and set the Service Availability for the *svc_Gimme* ECOA Service. That *START* operation is implemented by the (C) code function *modServer__START__received* in the (C) code unit *modServer.c*:

```
void modServer__START__received(modServer__context* context)
{
     ECOA__return_status erc;
     printf( "Server is running...\n" );
     erc = modServer_container__set_service_availability( context,
          modServer_container__service_id__svc_Gimme, ECOA__TRUE );
}
```

The "Gimme" Service request handler is implemented by the code function *modServer__Gimme__request_received*:

```
void modServer__Gimme__request_received(modServer__context* context,
          const ECOA__uint32 ID,
          const sql__string* statement)
{
     ECOA__char8 src;
     ECOA__int32 value;
     sql__string data;
     //
     sscanf( statement->data, "select * from iTable where %c = %d",
                                   &src, &value );
     data.current_size = sprintf( data.data, "%c[%d]: %d",
                                   src, value, context->user.count++ );
     modServer_container__Gimme__response_send( context, ID, &data );
}
```

This function extracts the source client and sequence number information from the request (into the local variables *src* and *value* respectively), builds the response message in the variable *data*, which is then returned to the calling client using the ECOA Container API function *modServer_container__Gimme__response_send*.

In order to maintain the total number of requests made across invocations of the *modServer__Gimme__request_received* function, a variable (*count*) is included in the "User Context" (see ref.[1]), and accessed as *context->user.count*. The User Context is declared in the (C) code header file *modServer_user_context.h*:

```
typedef struct
{
     ECOA__int32 count;
} modServer_user_context;
```

This count is initialized (to zero) when the Module is initialized:

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

The Client ASC

On invocation of the *START* operation, the *Client* ASC will initialize its *Ticker* Trigger Instance, as required by the ECOA (ref.[1]). That *START* operation is implemented by the (C) code function *modClient__START__received* in the (C) code unit *modClient.c*:

```
void modClient__START__received(modClient__context* context)
{
     modClient_container__INITIALIZE__Ticker( context );
}
```

When a Module or Trigger Instance has changed Module Lifecycle state, the ECOA Software Platform issues a Lifecycle Notification to its parent Supervision Module. In the present case, the notification

for the Client ASC's *Ticker* Trigger Instance is implemented by the *modClient__lifecycle_notification__Ticker* code function:

```
void modClient__lifecycle_notification__Ticker(modClient__context* context,
     ECOA__module_states_type previous_state,
     ECOA__module_states_type new_state)
{
     if( previous_state == ECOA__module_states_type_IDLE &&
         new_state == ECOA__module_states_type_READY ){
          modClient_container__START__Ticker( context );
     }
}
```

That is, if the *Ticker* Trigger Instance has transitioned from the **IDLE** to the **READY** state (i.e. it has been initialized), then it is started. Other Module Lifecycle state transitions, such as transitioning from the **READY** to the **RUNNING** state (as a result of a **START** operation) are ignored in this example.

Since we have made the decision not to change the Service Availability once it has been set by the *Server* ASC, the *modClient__service_availability_changed* and *modClient__service-_provider_changed* function stubs can be left empty.

All we need to do now then is to program what to do when the *Ticker* Trigger Instance fires, i.e. to populate the *modClient__Tick__received* function stub.

```
void modClient__Tick__received(modClient__context* context)
{
     ECOA__int32 IAm;
     ECOA__return_status errc;
     sql__string requestStmt, responseData;
     //
     modClient_container__get_Id_value( context, &IAm );
     //
     requestStmt.current_size = sprintf( requestStmt.data,
                    "select * from iTable where %c = %d;",
                         (IAm==1?'A':'B'), context->user.Count++ );
     printf( "Request: %s\n", requestStmt.data );
     errc = modClient_container__Gimme__request_sync( context,
                                        &requestStmt, &responseData );
     if( errc != ECOA__return_status_OK ){
          printf( "Gimme request failed with errc = %ld\n", (long)errc );
     }else{
          // zero terminate responseData for printf...
          responseData.data[responseData.current_size] = '\000';
          printf( "Response: %s\n", responseData.data );
     }
}
```

That is, the required SQL statement is composed, using the ECOA *get_<property>_value* API (in this case for the "*Id*" property) to determine whether the "A" or "B" source is selected, and (as with

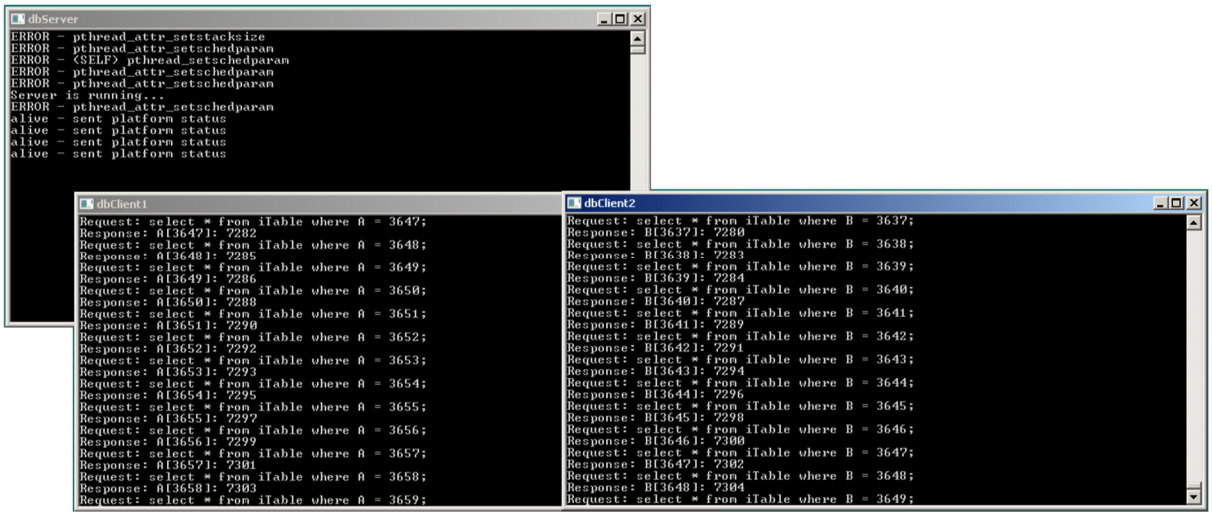
the Server ASC) using a non-volatile *Count* variable held in the User Context to provide a sequence number maintained across calls to *modClient__Tick__received*.

Once composed, the request is sent to the server using the *modClient_container__Gimme__request_sync* API, and because a synchronous Request-Response call is made, the response (in variable *responseData*) is immediately available to print as output.

Program Output

When the ECOA "*GimmeGimmeGimme*" Assembly is built and run (in a single Node deployment), an output similar to Figure 6 should be achieved. This shows three Windows® Command Prompt panes each running one of the three Protection Domains (*dbServer*, *dbClient1*, *dbClient2*). With the exception of the *Server* ASC start-up message, the only significant output[2] from the *dbServer* Protection Domain is the (10 second) periodic ECOA Software Platform heartbeat ("alive") message. Each of the two *Client* ASCs (running in the *dbClient1/2* Protection Domains) outputs, at each iteration, both the sent SQL request message, and the received data response.

Figure 6 ECOA "*GimmeGimmeGimme*" in Execution



[2] On Windows some ignorable error messages are output from the ECOA Platform code on start-up, due to the ECOA Platform code attempting to set thread naming attributes that Windows doesn't support. The functioning of the ECOA Platform is unaffected by these errors.

References

1	European Component Oriented Architecture (ECOA) Collaboration Programme: Architecture Specification (Parts 1 to 11) "ECOA" is a registered trade mark.
2	Client-server model *https://en.wikipedia.org/wiki/Client%E2%80%93server_model*
3	Gimme! Gimme! Gimme! (A Man After Midnight) ABBA, ©1979