

---

# Subscription Server Design Review

Shankar Rachakonda

[vrachako@eos.hitc.com](mailto:vrachako@eos.hitc.com)

---

17 April 1996

# CSCI Overview



- **Driving Requirements**
- **CSCI Software Design**
- **OO Design Models**
- **Design Drill Down**

# Driving Requirements



## Architectural Drivers

- Support a Producer-Consumer paradigm
- Generic Event-Action Model

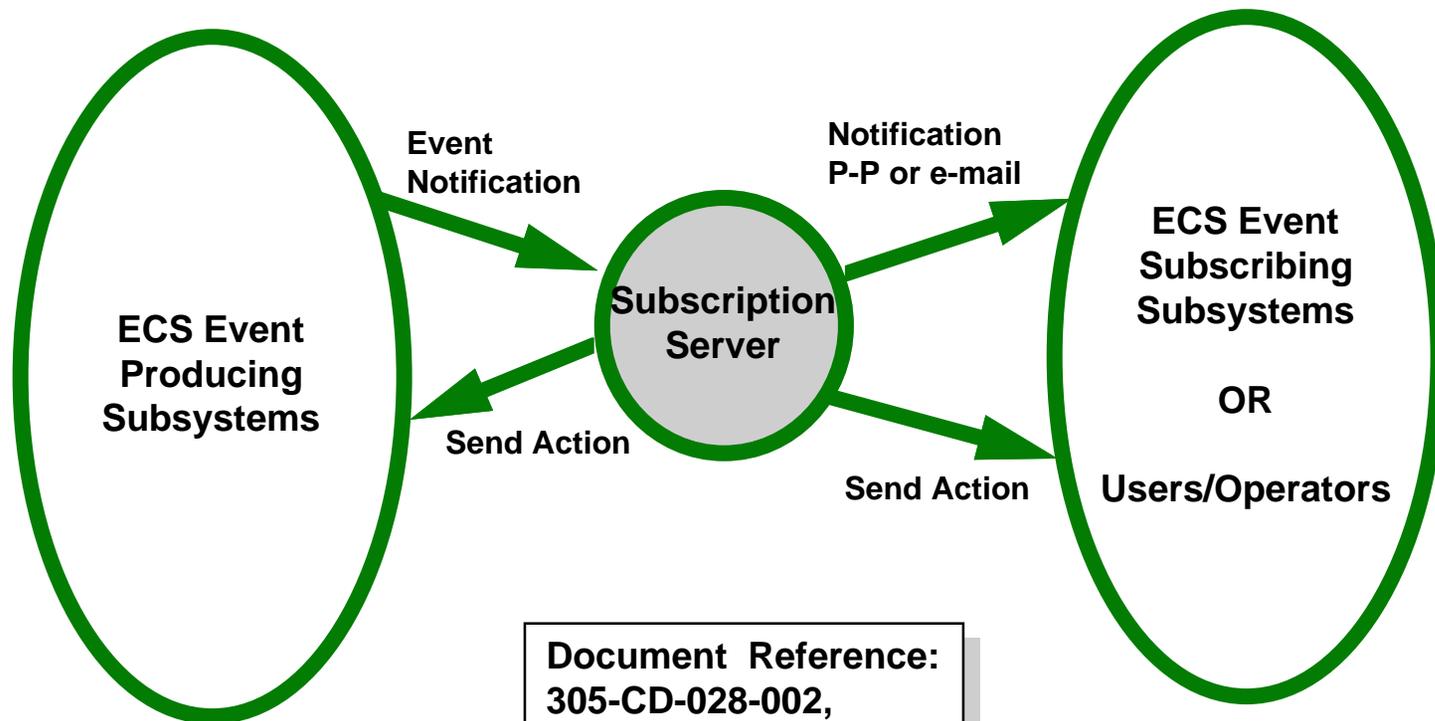
## New Release B Features

- Migration of Data Server Subscription Model
  - Broader range of actions
  - Complete separation of client, event source and service provider
- A common mechanism across subsystems
- Timed events
- Modifying subscriptions



# Subscription Server Context

**Subscription Server allows ECS to support clients' desire to have actions taken based on the occurrence of future events**



**Document Reference:  
305-CD-028-002,  
Fig. 4.5.3.1-1**

# Subscription Server Context



## ECS Subscription Event Producers

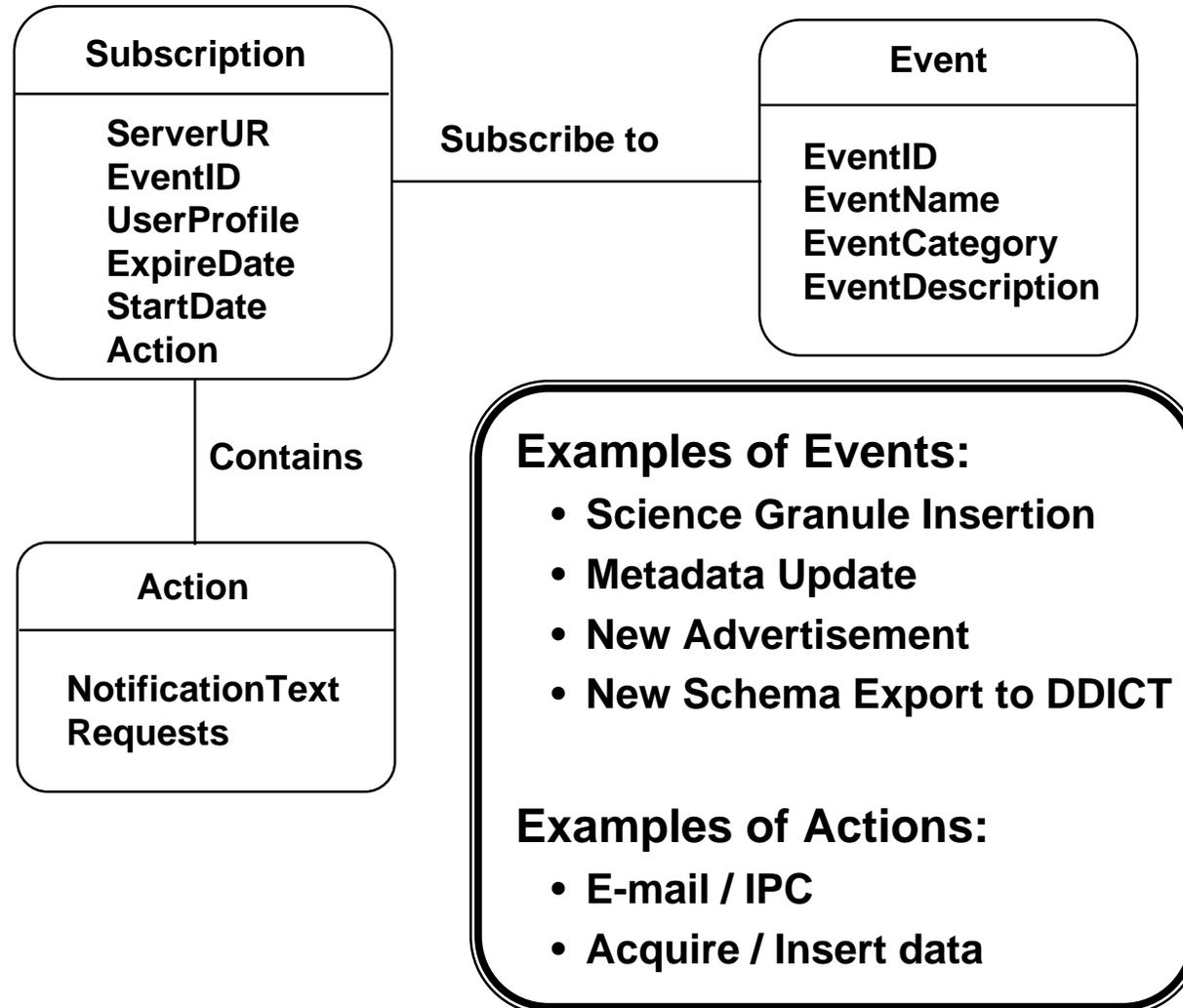
- Data Server
- Advertising Service
- Data Dictionary Service

## ECS Subscribers

- Client on behalf of science users
- Data Processing
- Planning
- Operator Interfaces on behalf of operators



# Attributes



# Subscription Service Capabilities



## Register New Events

- Stored persistently
- Made available through Advertisement Service

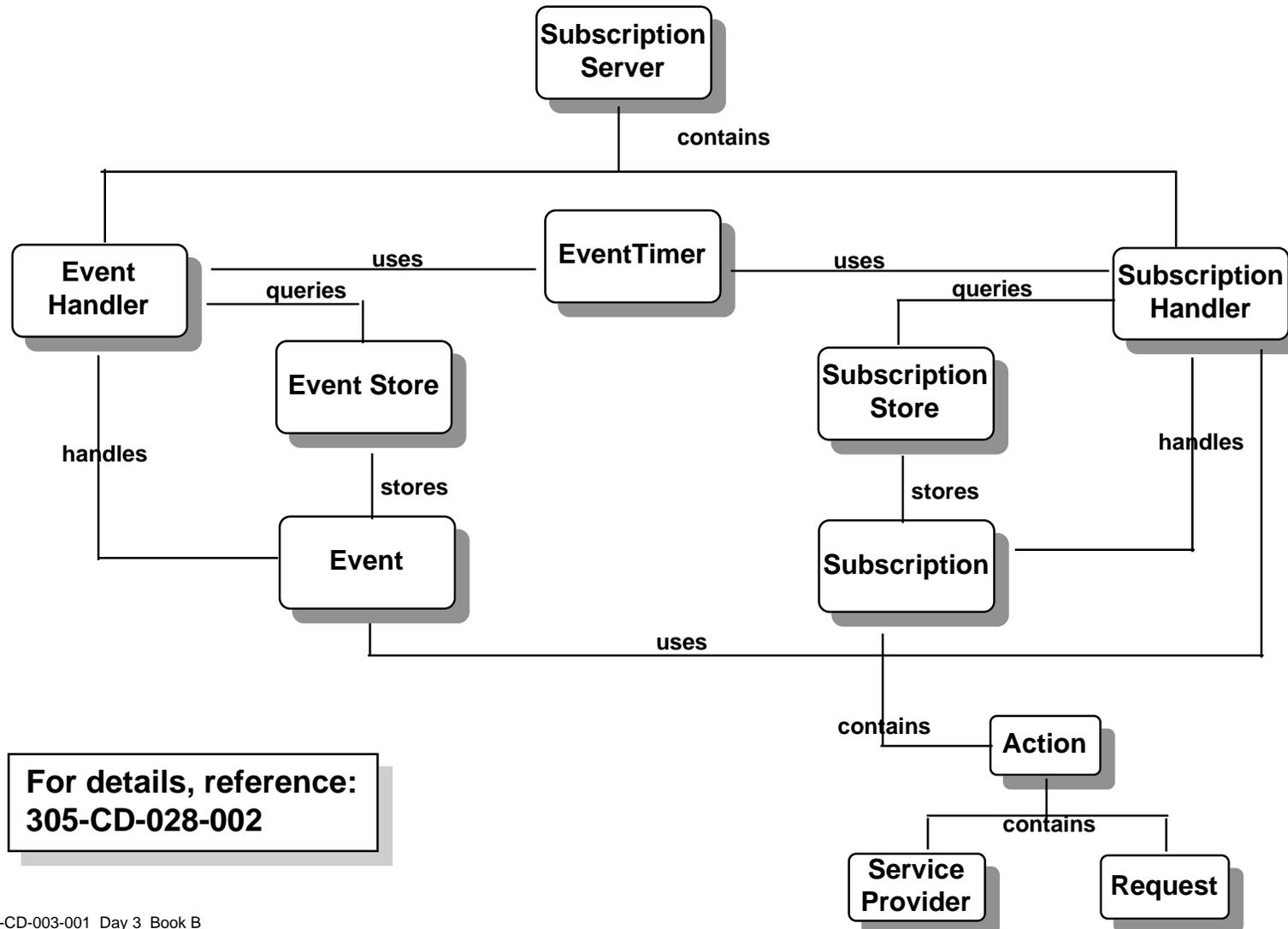
## Accept Subscriptions

- Accept new Subscription Requests that specify an action to be taken and an event to initiate the action
- Accept Subscription Update Requests to update stored Subscriptions
- Validate Subscription Requests

## Process Subscriptions upon Event Notification

- Identify all subscriptions to the specified event
- Process the actions defined in the Subscriptions
  - E-mail notification
  - direct program interface to other service providers

# Software Design – High Level Class Model



For details, reference:  
305-CD-028-002

# Class Descriptions



<b>SubscriptionServer</b>	<b>Provides access to the processing of events and subscriptions</b>
<b>Event</b>	<b>Implementation of an event</b>
<b>EventHandler</b>	<b>Provides an interface to manage events</b>
<b>EventStore</b>	<b>Provides an interface to interact with a database of events</b>
<b>Subscription</b>	<b>Implementation of a single subscription</b>
<b>SubscriptionHandler</b>	<b>Provides an interface to manage subscriptions</b>
<b>SubscriptionStore</b>	<b>Provides an interface to interact with a database of subscriptions</b>
<b>Action</b>	<b>Actions to be performed when events fire</b>
<b>EventTimer</b>	<b>Time keeper to maintain and generate timed events</b>

# Object Model



The following object models will be reviewed:

<u>Model Name</u>	<u>Document Reference</u>	<u>Section</u>
Subscription Client	305-CD-028-002	4.5.3.3
Subscription Server	305-CD-028-002	4.5.3.4
Subscription Attributes	305-CD-028-002	4.5.3.5
Notification Object	305-CD-028-002	4.5.3.6

# Dynamic Model



The following event traces will be reviewed:

<u>Event Trace Name</u>	<u>Document Reference</u>	<u>Section</u>
Submitting a Subscription	305-CD-028-002	4.5.3.5.1
Registering a Subscribable Event	305-CD-028-002	4.5.3.5.6
Fulfilling a One Time Subscription	305-CD-028-002	4.5.3.5.4



# Event Traces

## Submitting a Subscription

### Scenario

- In this scenario the client has retrieved a subscription advertisement from the Advertising Server and has decided that he/she would like to be notified upon the future occurrence of that advertised event.

### Functional Description

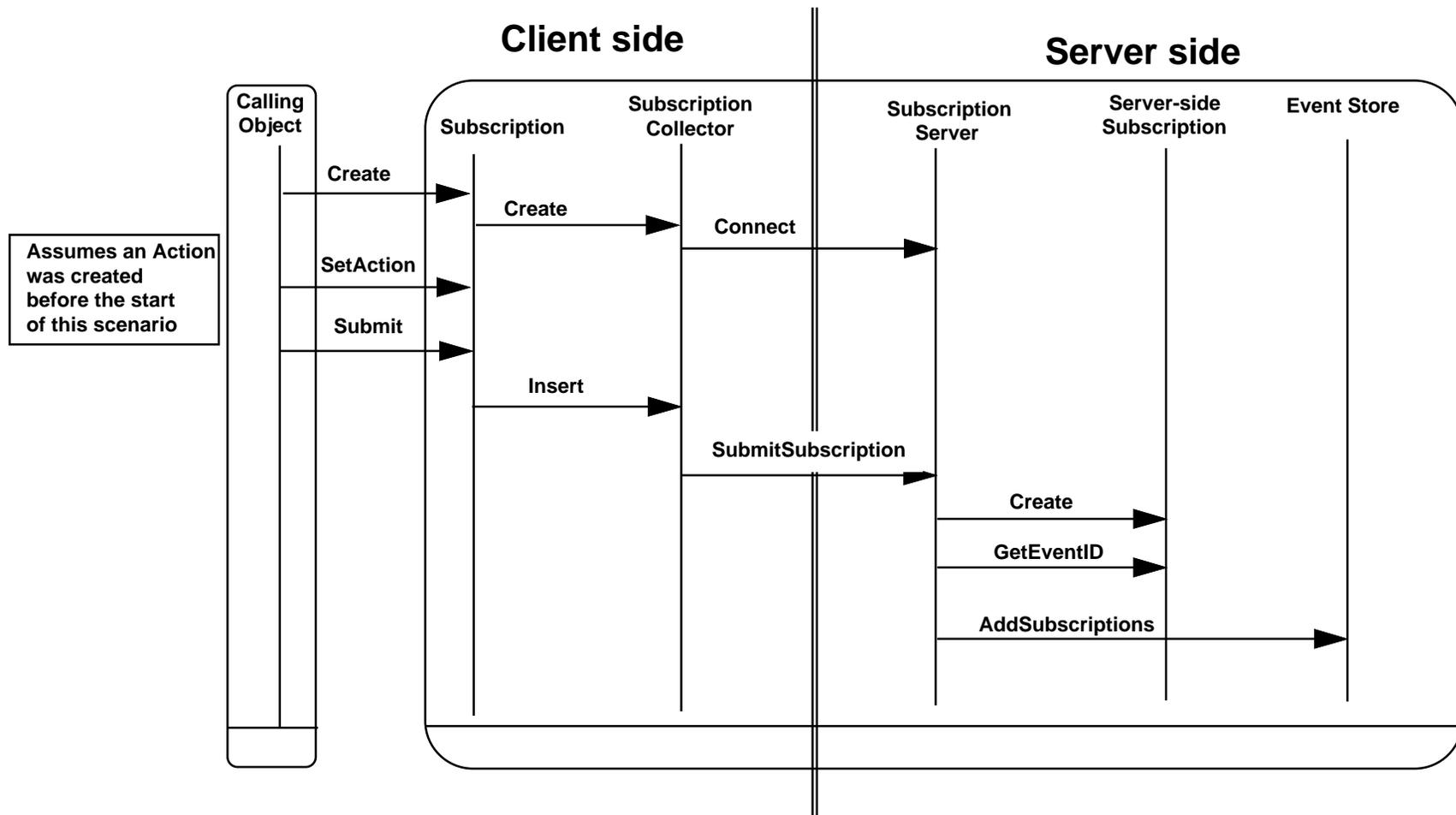
- Client application creates a subscription, passing the advertisement (event information) and user information.
- The client then creates an action which includes a request and an optional notification text string (user-entered data).
- Next the EcCISubscription objects SetAction() operation is used to register the user defined action.
- Finally, the server uses AddSubscriptions() to add the new subscription to its list of managed subscriptions.

### Assumptions/Preconditions

- A user has selected an advertised subscription from the Advertiser.



# Submitting a Subscription





# Event Traces

## Registering a Subscribable Event

### Scenario

- The purpose of this scenario is to show how a subscribable event gets established.

### Functional Description

- Client Application creates a Client Event (EcClientEvent) with a name, category, and description. This causes a creation of the event (EcServerEvent) within the Subscription Server process space.
- Client Application then invokes the Register() method which stores the event in database, and notifies the Subscription Server that this is now a known event.
- The Subscription Server coordinates with the Data Management Subsystem to advertise the fact that this event is subscribable.

# Event Traces



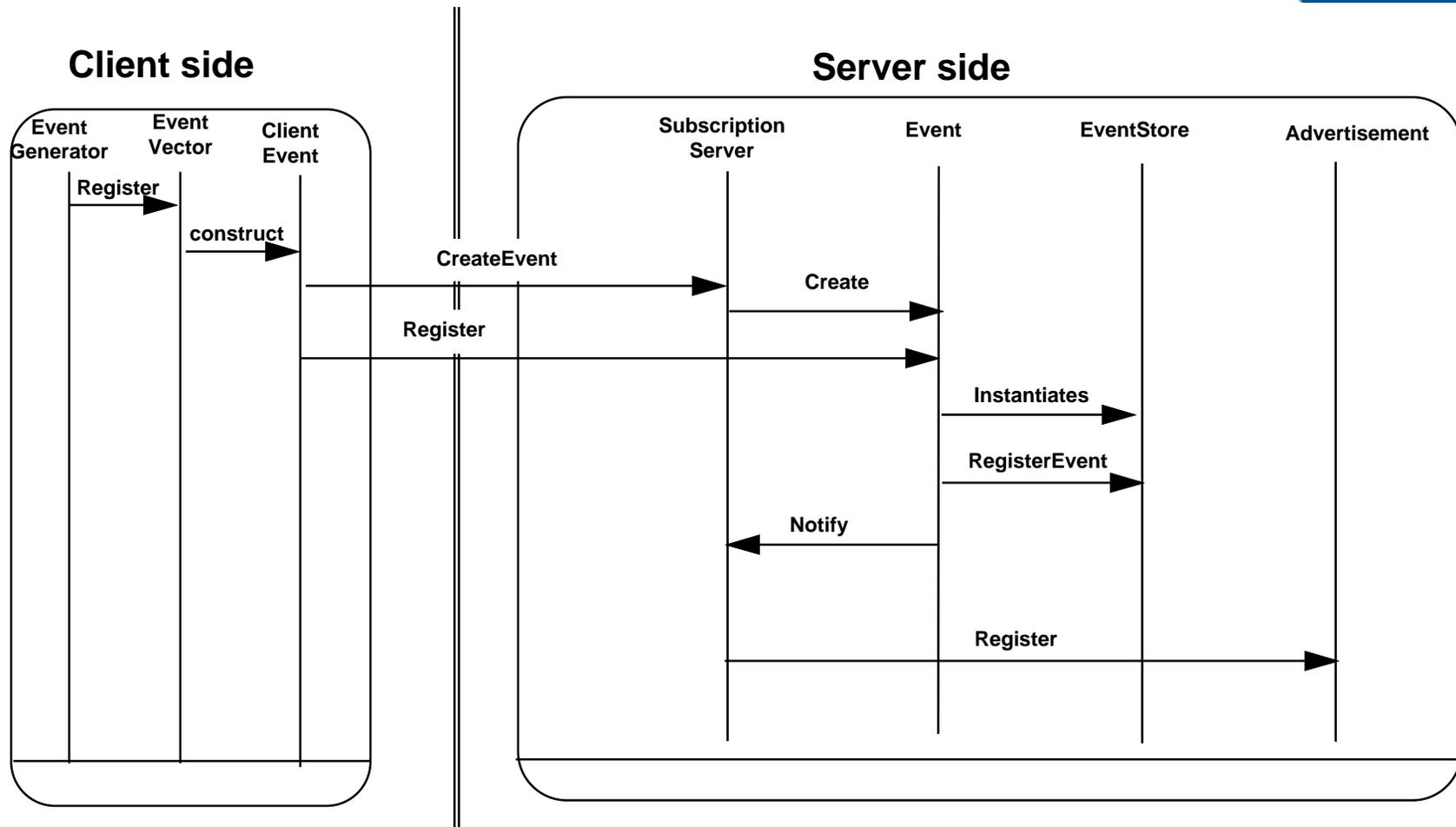
## Scenario (Continued)

### Assumptions/Preconditions

- The Science Data Server has the UR of its Subscription Server Factory.
- The Subscription Server is active.
- There is just one event being registered.



# Registering an Event





# Event Traces

## Fulfilling a One Time Subscription

### Scenario

- This scenario shows how a server fulfills a previously submitted subscription.

### Functional Description

- Client Application instantiates the event (EcCIEvent) with the known eventID.
- Client Application calls EcCIEvent.Trigger() method, which in turn calls the Trigger() method of EcSbEvent on the server side.
- EcSbEvent.Trigger() instantiates the EcSbSubscription Handler Object.
- EcSbSubscriptionHandler.ProcessEvent() method will retrieve all subscriptions of this event from the Subscription Store and execute them.



# Event Traces

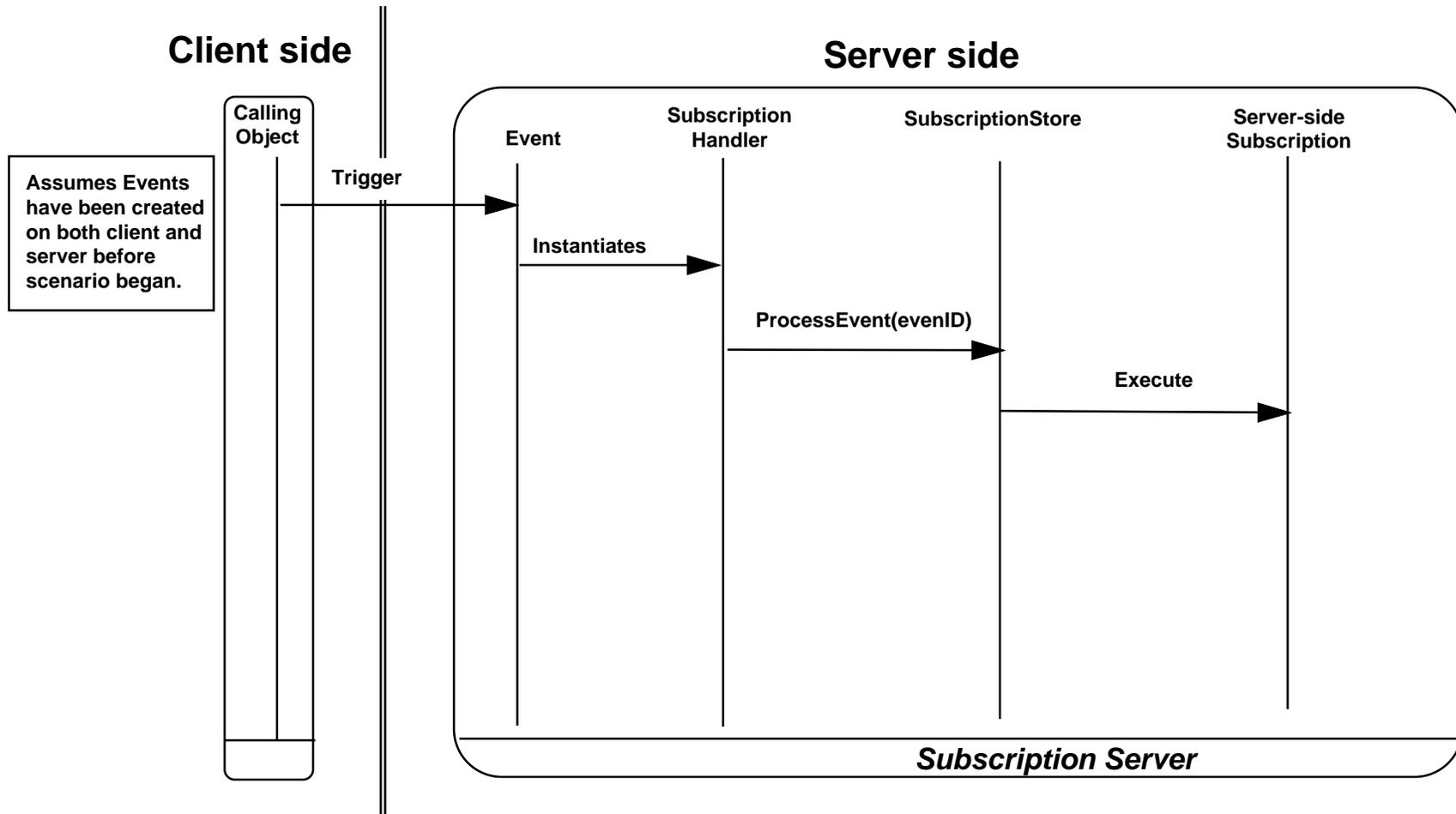
## Scenarios (Continued)

- The **EcSbSubscription** object contains the receivers (via **GetNotify()**) and the notification text (via **GetText()**)
- The notification is then sent via the **EcSbNotification** object.

## Assumptions/Preconditions

- The event has been registered, and is subscribable.
- The event has occurred.

# Event Notification & Subscription Execution





# Selected Public Interfaces

## **EcUtStatus EcCIEvent:Register()**

- This method is called from the client application.

## **Functional Description**

- Check if all event data has been set.
- IF status indicates failure
  - Return failed status
- Call Register() method of the Event Proxy distributed object.
- Check status
- Return



# Selected Public Interface

**EcUtStatus EcSbEvent:Trigger(GIPParameterList&)**

- This method is called at the occurrence of a given event.

## Functional Description

- Instantiates **EcSbSubscriptionHandler** object.
- Call **EcSbSubscriptionHandler->ProcessEvent(eventID,GIPParam)** to fire all the subscriptions for this event.
- **IF status indicates failure**
  - Return failed status



# Selected Public Interface

**EcUtStatus EcCISubscription::Submit()**

## Functional Description

- Check if all data has been set.
- IF status indicates failure
  - Return failed status
- Invoke **EcScSubscriptionCollector::Insert(\*this)** method to add subscription into the collection vector.
- Invoke **EcScSubscriptionCollector::SubmitSubscription()** method, to submit the provided subscription to subscription server.
- Check status



# Design Drill Down

Client Application Main

```
main()
{
...
// Instantiates client subscription(EcCISubscription). The underlying factory will automatically
// create this subscription on the server side.
EcScSubscription* mySubscription = new EcScSubscription(serverUR, EventID,
user, expDate, startDate, action);
mySubscription->Submit();
...
}
```



# Design Drill Down

EcCISubscription

```
//Constructor of EcCISubscription. This implementation uses the concept of Factory to  
// create subscription object dynamically. In this example, the auto-filled constructor is  
// used to create subscription.
```

```
EcCISubscription::EcCISubscription(EcUrUR server,  
                                   EsTSbEventID eventId,  
                                   MsAcUserProfile& user,  
                                   RWDate expDate,  
                                   RWDate startDate,  
                                   EcCIAction& action)
```

```
{  
// Instantiates the distributed client side factory interface(EcCIFactoryProxy). This will in turn  
// instantiate the client side factory object factoryIDL_1_0 using the provided server UR  
myFactory = new EcCIFactoryProxy(server);
```

```
// Pack the subscription data into byte data
```

```
....
```

# Design Drill Down



(EcClSubscription Continued)

```
// Call method CreateNewSubscription() to send data across the network and create
// subscription on the server side. This method will also register this subscription
// interface with GSO, and return a DCE object reference of the interface.
mySubscriptionProxyRef = myFactory->CreateNewSubscription(data);
...

// Use the object reference to construct client subscription proxy.
mySubscriptionProxy = new subscriptionIDL_1_0(mySubscriptionProxyRef);

// Save the object uuid for later use (for example, to destroy the object)
mySubscriptionProxyUuid = mySubscriptionProxyRef->objId;

}
```



# Design Drill Down

Submit

```
// This method is called by client application main to submit a subscription  
// to the system. The subscription will be stored in the subscription store  
// database.
```

```
EcUtStatus EcScSubscription::Submit()  
{  
...  
status = mySubscriptionProxy->Submit();  
...  
}
```



# Design Drill Down

## Subscription Server

```
// EcSbFactoryConcrete inherits from the factory manager object, in addition,  
// it maintains a list of subscriptions that are being created.
```

```
extern EcSbSubscriptionServer *mySubscriptionServer;
```

```
DCEObjRefT* EcSbFactoryConcrete::CreateNewSubscription(data)
```

```
{
```

```
// Unpack data
```

```
...
```

```
// Subscription Server is used to create a server side (local) subscription
```

```
EcSbSubscription* mySubscription = mySubscriptionServer->CreateSubscription(  
eventID, user, expDate, startDate, action);
```



# Design Drill Down

Subscription Server (Continued)

```
// Instantiates server side subscription interface (EcSdSubscriptionConcrete).  
// EcSdSubscriptionConcrete inherits from subscription manager object  
// subscriptionIDL_1_0_Mgr. In addition, it contains a pointer to its server side (local)  
// subscription object. This will be used to identify its corresponding local server subscription  
// object when the object is destroyed.  
EcSdSubscriptionConcrete* myNewSub =  
           new EcSdSubscriptionConcrete(mySubscription);  
  
// Add this subscription to the list that the factory maintains  
AddSubscription(myNewSub);  
  
theServer->RegisterObject(*myNewSub), true);  
  
return myNewSub->GetObjectReference();  
}
```

# Design Drill Down



Trigger

```
EcUtStatus EcSbEvent::Trigger()  
{  
...  
EcSbSubscriptionHandler* myHandler = new EcSbSubscriptionHandler();  
  
// get all subscriptions of this event, execute the corresponding action  
status = myHandler->ProcessEvent(GetID());  
...  
return status;  
}
```



# Performance Considerations

- **Submitting subscriptions occurs so infrequently, the impact on the system is negligible**
- **E-Mail flow rate not a concern**
- **Even in case of abnormal events where every user needs to be notified Performance Modeling indicates that the impact will not be catastrophic**
- **Throttling of Message Handler always an option. Isolation from a hardware standpoint**

## Infrastructure Review Issues

- **Need to understand the overhead of SendMail is now much less important**

# Nominal E-Mail Usage



**E-mail frequency is dependent on the number of external subscription notifications per day.**

**The frequency of external subscription notifications was determined from the user modeling nominal pull subscription requests by DAAC.**

**All numbers are daily.**

**EDC - 1023**

**LaRC - 241**

**GSFC - 706**

**NSIDC - 13**

**ASF - 82**

**JPL - 36**

**TOTAL = 2101**