Platform EGO™ Developer's Guide

Platform EGO version 1.2.3
December, 2008

Comments to: doc@platform.com
Support: support@platform.com

Contents

CHAPTER 1

Introduction to the Platform EGO SDK

The Platform Enterprise Grid Orchestrator™ (EGO) is a service that virtualizes a distributed heterogeneous computing environment enabling users, administrators, and developers to treat a collection of distributed software and hardware resources as components of a virtual computer. Platform EGO enables all applications, services, and workloads to access a shared infrastructure. Like a node operating system, Platform EGO takes physical resources and virtualizes them creating a virtual environment. It allocates the resources according to policy and simplifies the management and improves availability of the entire environment.

Contents

EGO concepts and terms

EGO master

The bulk of the intelligence in Platform EGO resides on the EGO master, which receives requests from clients and interacts with the underlying nodes to gather resource information.

EGO hosts contain the local information collection and execution agents taking instructions from the EGO master.

Resources

Resources are physical and logical entities that are used by applications in order to run. A resource of a particular type is associated with attributes. For example, a host has attributes of memory, CPU utilization, operating system type, etc. Platform EGO deals with resource allocation at the granularity of physical hosts, logical sub-divisions of the physical host known as cpu slots, software license features, and includes an extensible resource model to cover storage space, network bandwidth, or data sets as resources whose use is controlled under policies

Consumers

A consumer is a generalized notion of something that uses a resource. A consumer may be an individual user, user group, application, project, department, or an entire company. Consumers are organized hierarchically to model the nature of an organization that wants to access compute resources.

Resource allocation requests

An allocation request is a request for a set of resources made by a client to Platform EGO. The client must identify the originating consumer that this request is for in order for Platform EGO to apply its policies on resource allocation.

A client for each unique consumer wanting to run work should issue a resource allocation request; this allows Platform EGO to coordinate the sharing of resources amongst multiple consumers across different applications. Separate allocation requests should be made for each application or set of services that needs resources, since the allocation resource requirements will likely be different between different applications or sets of services.

Services

Platform EGO is an operating environment for hosting distributed services. Platform EGO provides a facility to define services that must be running and manages the lifecycle of their execution. Putting services under EGO management provides centralized control, virtualization of service placement, and failover.

Platform EGO comes with a set of standard services that would be commonly used within any environment. These include Service Controller, Service Director, web portal service, calendar service, and logging service. These services can be used together with application specific services to manage different workloads and environments.

Containers (Activities)

Services, in general, require some sort of execution context to be established; this may include a virtual machine (VM) or a J2EE application server, or some OS-level construct. To abstract this concept we introduce the notion of an activity or container as a hosting environment for services. The container is the main "unit of execution" from the point of view of the EGO kernel. Within the context of Platform EGO, container and activity have the same meaning. The container term is a legacy of Platform EGO's native C APIs whereas activity is a term that is favored within the context of Web Services. This guide uses both terms in keeping with this philosophy.

The relationship between services and containers is, in general, many to one, i.e., multiple services will run in the same container.

Platform EGO SDK

The EGO SDK enables developers to create services or applications that run on top of Platform EGO by accessing its C API or Web Service interface. The SDK is intended for Independent Software Vendors (ISVs) or system vendors looking to integrate their applications into the EGO environment. The SDK comes with a full runtime enabling developers to develop and test in the same environment that exists in the production version of Platform EGO.

Major components

Here is a list of the major SDK components:

EGO C SDK (com.platform.ego.c plug-in)

- header files
  - vem.api.h
  - vem.errno.h
  - vem.common.h
  - vem.version.h
  - esc.api.h
  - esd.h
  - esdplugin.h
- dynamic libraries
  - libvem.so (EGO stub library)
  - sec_ego_default.so (EGO security plug-in)
  - libesc.so (Service Controller library)
  - esd_ego_default.so (Service Director plug-in)
  - libxml.so (XML utility library)
  - libesd.so
- static libraries
  - libvem.a
  - libesc.a
  - libxml.a

EGO Web Service SDK

- com.platform.ego.soap plug-in

Help

- EGO Developer's Guide
- C API Reference
- Web Service XML Schema Reference
- Web Service WSDL files (note: the Web Service WSDL documentation is embedded in the WSDL files)

Code samples

- C client
- Java client for web services

ego.conf configuration file

The `ego.conf` configuration file is required for the client application to connect to Platform EGO because it contains the master host name and port numbers for the agents. The location of this file is passed to the API when a request is made to open a connection to the EGO master. This file is included in the SDK.

EGO client development tools

C API plug-in

The C language integration enables developers to create projects that contain C language code and make files. This provides the developer with a quick means of creating a new project suitable for use as an EGO client.

Web Service plug-in

The Java / Web Service language integration is used to create projects that contain Java language code and build files. This provides the programmer with a quick means of creating a new project suitable for use as an EGO Web Service client.

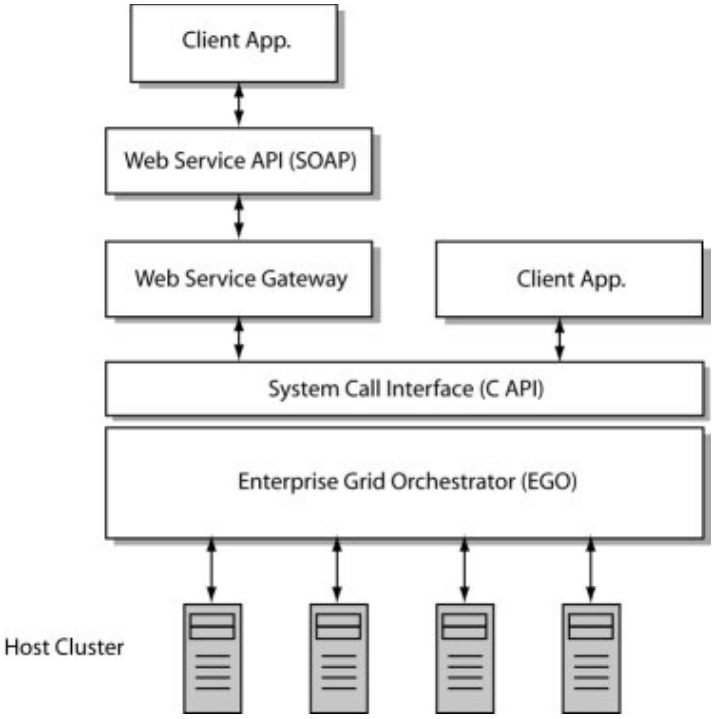
Eclipse

Eclipse is an open source community whose projects are focused on providing an extensible development platform and application frameworks for building software. Eclipse provides extensible tools and frameworks that span the software development lifecycle, including support for modeling, language development environments for Java, C/C++ and others, testing and performance, business intelligence, rich client applications and embedded development. The EGO client development tools were specifically designed as plug-ins to the Eclipse IDE.

EGO API

The EGO Application Programming Interface (API) is a collection of programming interfaces used by a client application to access EGO functionality. A client can be either registered or unregistered. Unregistered clients can perform queries for host and consumer information. Registered clients have access to additional functionality such as resource allocation and container execution.

Clients can interact with EGO through either the native C API or the Web Service interface.



EGO functionality

In order to take advantage of what the API has to offer, it is important to understand the core functions of the EGO kernel. These functions can be categorized as information, allocation, execution, and administration.

Information

Platform EGO aggregates information from all the hosts in the cluster providing a single point from which clients can request information about the state of any host. This includes the state of individual resources, the status of allocation requests, the consumer hierarchy including current resources assigned to each consumer, as well as activities that have been started in the distributed environment.

Allocation

A critical function of Platform EGO is to manage the allocation requests that come from different clients. Clients request and release CPU slots through the allocate and release interfaces identifying the number and type of resources they need based on various host attributes and the consumer they are being allocated on behalf of.

Based on the availability of resources and the amount of resources a consumer is entitled to, Platform EGO applies policies to determine how many resources to allocate to a given request. The client is notified asynchronously as the resources become available. In this case, CPU slots are identified by the physical host that they reside on and this host information is passed back to the client.

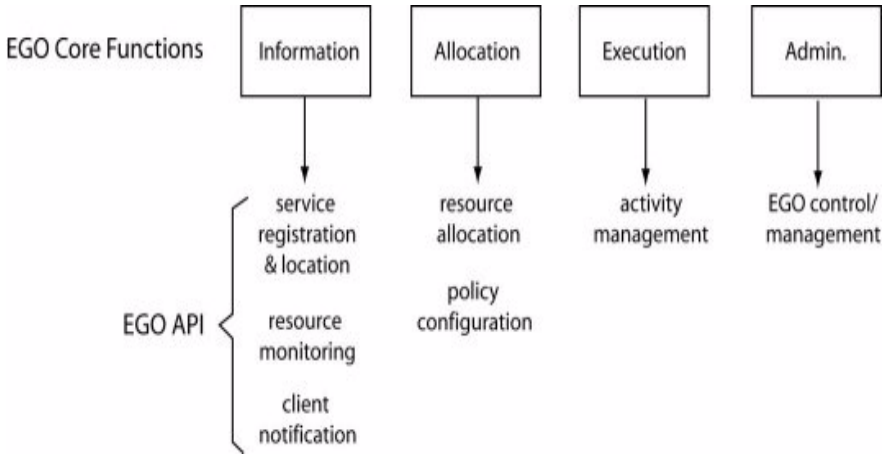
Execution

Once a client has obtained resources from Platform EGO, it will want to utilize those resources to perform work. Platform EGO provides a mechanism to allow clients to execute actions on the resources that have been allocated to them. In the typical case where compute hosts or CPU slots are the resources, execution action involves starting, stopping, and controlling execution activities.

Platform EGO instantiates OS-level processes on the hosts representing the container/activity and sets the execution context around the activity such as environment variables, working directories, and resource limits. Changes in the status of the activity on any host are reported asynchronously to the client enabling the client to handle these changes.

EGO API

This section describes the external interfaces available to a client application for invoking EGO services.



Client registration interface

Registration is required for clients to be able to request allocations and start activities. The client must also be registered in order to receive notifications (callbacks) for events related to the client's allocations, assigned resources, and activities.

Resource allocation interface

The EGO client uses the Resource Allocation Interface to request Platform EGO to allocate, change, release, replace or list resources. The client must register with Platform EGO using the client registration interface before being able to use any of the resource allocation interface operations.

Container management interface

The Container Management Interface supports the execution of activities on hosts managed by Platform EGO. An EGO client uses this interface to start its activities once it has the resources it needs through the resource allocation interface. This interface is also used to suspend, resume, and collect the status of a container.

Policy configuration interface

The Policy Configuration Interface allows for the setting of resource allocation policies in Platform EGO. The interface allows administrators to define consumers and place them in the consumer hierarchy. Policies are related to how resources should be divided between consumers at any given level of the hierarchy.

The policies at each consumer level specify parameters that control the division of resources among its children.

Resource monitoring interface

This interface supports operations to query information from the EGO master and cluster managed by the EGO master. The interface can also be used to collect information pertaining to a group of resources or pertaining to activities that were started by the EGO kernel.

Client notification interface

This interface supports a number of client notification operations such as informing the client to shut itself down or to informing the client that a resource has been added to the allocation. This interface is also used by the EGO kernel to notify the client of status changes in activities, named resources, or membership in resource groups.

Administration interface

This interface supports the control and management functions of EGO. This includes the vast number of operations that enable the cluster administrator to set up and manage entities such as resource host groups, users, and consumers.

API calls, callbacks, and notifications

API calls

All API calls are synchronous, meaning that a message sent by the client is expected to have a corresponding reply, i.e., the calling operation code has one and only one reply code. The client side blocks execution until the matching reply is received.

Synchronous call sequence: (callback is received while an API is blocking for reply)

1. During an API call, the function waits for matching reply code.
2. A message is received but API determines that it is not the reply to its call. The API places this message on a message queue internal to the client library.
3. The function continues to wait until a matching reply code is received. The API pushes all other messages on the message queue.

EGO callbacks

Some calls to the C API, such as vem_alloc(), may result in asynchronous callbacks. At the time of registration, a client specifies callback functions that get executed when certain events occur in Platform EGO. The time when these functions get executed is not determined by when the functions are supplied so they are considered asynchronous. These callback messages can occur at any time, even when the client is waiting for a reply to another call.

Asynchronous callback sequence: (callback received outside of an API call)

1. Client uses vem_select() to determine if there is a pendinging message from Platform EGO to be processed.
2. Client uses vem_read() to retrieve the pending message.

EGO notifications

Instead of callbacks, the Web Service API uses notifications. Through the Web Service API, the client specifies a communication endpoint where Platform EGO sends a message when certain events happen.

API and Web Service interface reference documentation

Reference documentation for the C API and Web Service interface is available through the Eclipse Help menu.

C API

The C API documentation provides detailed descriptions of the API functions, structures, enumerations, and preprocessor directives. To access the C API documentation, select the reference documentation under EGO Developer Documentation in the Eclipse Help Contents pane.

Web Service interface

The Web Service reference documentation consists of WSDL and schema information. The WSDLs describe the Web Service interface (operations) and messages whereas the schemas describe the data types of the input/output arguments and their structure. To access the Web Service documentation, select the appropriate reference documentation under EGO Developer Documentation in the Eclipse Help Contents pane. The WSDL and schema documentation is also embedded in the individual WSDL and schema files.

About Web Services

Web Services provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks. This section begins with a description of the major components and concepts of a Web Service. Later, we describe these concepts within the EGO context.

Web Service components

XML

XML is used in the Web Services architecture as the platform-independent format for transferring information between the Web Service and the Web Service client. The XML format ensures uniform data representation and exchange. XML 1.0 was released as a W3C Recommendation; refer to document REC-xml-19980210 for further information.

WSDL

The Web Services Description Language (WSDL) describes the message syntax associated with the invocation and response of a Web Service. A WSDL file is an XML document that defines the Web Service operations and associated input/output parameters. In a way, the WSDL can be considered a contract between the Web Services client and the Web Services server.

Basically, a WSDL document describes three fundamental properties of a Web Service:

- The operations (methods) that the service provides including input arguments needed to invoke them and the response.
- Details of the data formats and protocols required to access the service's operations.
- Service location details such as a URL.

WSDL 1.1 was suggested in a note to W3C as an XML format for describing Web Services; refer to http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

XML Schema

XML schemas are used to specify the structure of WSDL documents and the data type of each element/attribute. XML schemas describe the documents that serve as the body of the SOAP messages traversing the EGO Web Service interface. XML Schema was approved as a W3C Recommendation; refer to http://www.w3.org/TR/xmlschema-1/.

SOAP

SOAP is the protocol used for communication between the Web Service and the client application. SOAP uses the Hypertext Transfer Protocol (HTTP or HTTPS) as the underlying protocol for transporting the data. SOAP 1.1 was suggested in a

note to W3C as a protocol for exchanging information in a distributed environment. The EGO Web Services implementation supports SOAP version 1.2; refer to http://www.w3.org/TR/soap12-part1/.

Web Service gateway

The Web Service gateway is a runtime component of Platform EGO. The gateway provides a standards-based Web Services interface for Web Service clients to access EGO functionality. The Web Service client sends its request to the gateway via SOAP protocol. The gateway calls the EGO C APIs in order to perform the required operations on behalf of the Web Service client and returns the results.

A closer look at an EGO WSDL and schema

This section looks at some key features of the EGO WSDLs and schemas.

SOAP binding style

SOAP supports two invocation models: Remote Procedure Calls (RPC) and document style.

In the RPC-style model, clients invoke the Web Service by sending parameters and receiving return values that are wrapped inside the SOAP body. These procedure calls are synchronous, which means that the client sends the request and waits for the response.

In the document style model, the client sends the parameters to the Web Service within an XML document. The Web Service receives the entire document, processes it and possibly returns a response message. Using the document style, the body of the SOAP message is interpreted as straight XML. Hence, this combination of sending a document with a literal XML infoset as a payload is referred to as document/literal. This is opposed to the RPC style that uses RPC conventions for the SOAP body as defined in the SOAP specification. One advantage of using the document-centric approach is that document messaging rules are more flexible than the RPC style, which allows for changes to the XML schema without breaking the calling applications.

The type of binding model that is implemented is determined by an attribute in the WSDL. The style attribute within the SOAP protocol binding can be set to either rpc or document. Here is an example of WSDL binding element that is set to document style.

```
<soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
```

Here is an example of the encoding technique that is specified on the soap:body element's use attribute. In this case, it is set to literal.

```
<wsdl:input>
    <soap:body parts="RequestAllocationRequest" use="literal"/>
</wsdl:input>
```

The Java code samples provided in the SDK use document style binding and that is why a document must be created for the request and response messages.

Passing parameters to a Web Service

The following example shows a portion of the WSDL file for the EGO MonitoringService Web Service and its associated schema file.

```
...
<wsdl:types>
<xsd:schema
 targetNamespace="http://www.platform.com/ego/2005/05/wsdl/monitoring"
 elementFormDefault="qualified">
...
<xsd:element name="ResourceInfoRequest">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="ego:ResourceRequirement" minOccurs="0" />
<xsd:element ref="ego:ResourceName" minOccurs="0" maxOccurs="unbounded" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name="ResourceInfoResponse">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="ego:Resource" minOccurs="0" maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
...
<wsdl:message name="ResourceInfoRequestMessage">
  <wsdl:part element="tns:ResourceInfoRequest" name="ResourceInfoRequest" />
  </wsdl:message>
<wsdl:message name="ResourceInfoResponseMessage">
  <wsdl:part element="tns:ResourceInfoResponse" name="ResourceInfoResponse"
 />
  </wsdl:message>
...
<wsdl:portType name="MonitoringPortType">
...
<wsdl:operation name="ResourceInfo">
<wsdl:input message="tns:ResourceInfoRequestMessage">
</wsdl:input>
<wsdl:output message="tns:ResourceInfoResponseMessage">
</wsdl:output>
  </wsdl:operation>
```

The operation name used in the example in this section is ResourceInfo. This operation takes a single input argument defined as a message of type ResourceInfoRequestMessage. Now we need to determine the number of parameters in this message and their data types.

In the message element named ResourceInfoRequestMessage, the part element is named ResourceInfoRequest. If you look up this message in the types element, you will find it contains two parameters: ResourceRequirement and ResourceName.

```
<xsd:element name="ResourceName" type="xsd:string">

</xsd:element>

<xsd:element name="ResourceRequirement" type="xsd:string">

</xsd:element>
```

In the schema file, you can see that the ResourceName and ResourceRequirement parameters have string data types. When you call the ResourceInfo operation, you pass both parameters as input.

Return values from a Web Service

Web Service operations often return information back to the client application. You can determine the name and data type of returned information by examining the WSDL and schema files for the Web Service.

Referring to the previous portion of the WSDL file for the EGO MonitoringService Web Service, we find that the operation named ResourceInfo returns a message of type ResourceInfoResponseMessage. In the message element named ResourceInfoResponseMessage, the part element is named ResourceInfoResponse. If you look up this message in the types element, you will find it contains a return argument called Resource. In the schema file, you can see that the Resource parameter has a complex data type. Complex data types are serialized as XML and returned from the Web Service as the result. The variable used to store the result must match the structure of the complex data type.

```
<xsd:element name="Resource">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="ego:ResourceState" minOccurs="0"/>
<xsd:element ref="ego:ConsumableAttribute" minOccurs="0"/>
<xsd:element ref="ego:Attribute" minOccurs="0" maxOccurs="unbounded"/>
<xsd:any namespace="##other" minOccurs="0" processContents="lax"/>
</xsd:sequence>
<xsd:attribute name="ResourceName" type="xsd:string" use="required"/>
<xsd:attribute name="ResourceType" type="xsd:anyURI"/>
</xsd:complexType>
</xsd:element>
```

Building a Web Service client

A Web Services client is an application capable of sending and receiving SOAP messages. Such an application serializes or deserializes the SOAP messages to a programming language type system enabling programmatic processing.

Here is the sequence for invoking a Web Service:

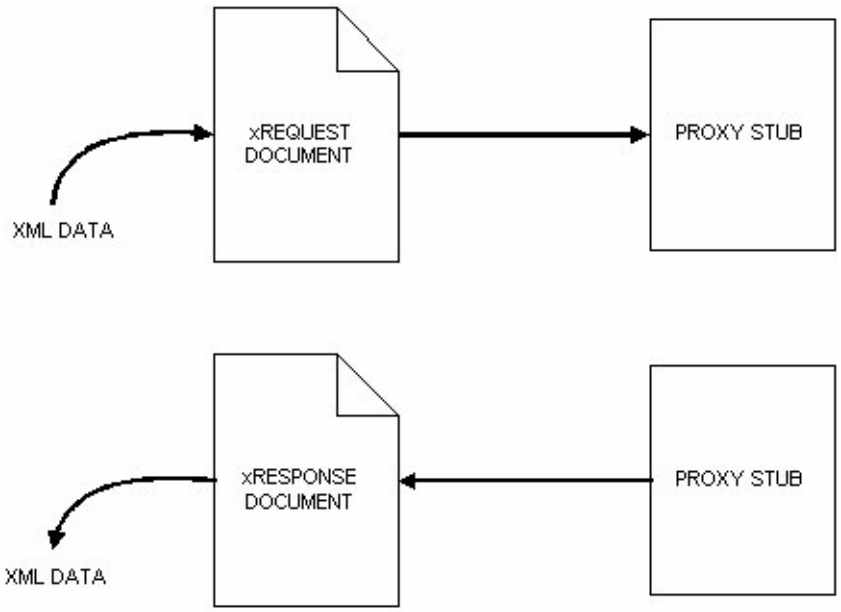
- Client serializes the arguments of the method call into the XML payload of the SOAP message
- Send the message to the Web Service
- Wait for a response (or timeout)
- Deserialize the XML payload in the response message to a local type/structure
- Return that type/structure as a value from the method call.

Using Axis2 to Develop Java Web Service Clients

As a client to a Web Service, encoding your requests in XML to the Web Service and decoding the responses you get back would be tedious (not to mention implementing the logic that deals with accepting requests and sending responses).

Apache Axis2 is an implementation of the SOAP protocol and it shields the developer from the details of dealing with SOAP and WSDL. You can use Axis on the client side to greatly facilitate the development of your client. All code samples included in this guide were developed with Axis2. Bear in mind that there are several tools available to aid in the development of a Web Service client and Platform does not endorse any particular one.

When using Axis2 to write your client, you don't need to deal directly with SOAP and XML. Axis creates a proxy (or stub) for your clients to abstract away SOAP. All you need to do is make the method calls on the Web Service proxy as if it were a local object.



The client calls the stub, the stub translates the call into a SOAP message, and the stub sends it to the Web Service. The listening server receives the SOAP message and translates it into a method call at the server. Since the server is written in Java, the SOAP message is turned into a Java call. The server's return values are translated back to SOAP and then returned to the stub, which translates the returned SOAP message into a Java response.

A sample Bash shell script is provided that creates client-side classes for consuming services described in the WSDL files. Run this script from the directory containing the EGO WSDL files.

```
#!/bin/bash
# Add the location of Java tools to PATH
export PATH=/usr/local/jdk/bin/:$PATH
# Set the location of Axis2 binary installation
AXIS2_HOME=/home/ACCOUNT DIRECTORY/axis2-0.92-bin
# Build Axis2 classpath
AXIS2_CLASSPATH=.
for i in $AXIS2_HOME/lib/*.jar; do AXIS2_CLASSPATH=$AXIS2_CLASSPATH:$i;
done

# Generate the Java classes
for i in *.wsdl; do echo $i; j=`echo $i | sed -e 's/\(.*\)\..*/\1/'`; echo $j ;
java -classpath $AXIS2_CLASSPATH org.apache.axis2.wsdl.WSDL2Java -uri $i $i
done

# Compile the generated classes
for i in codegen codegen/databinding/com/platform/www
 codegen/databinding/org/w3/www codegen/databinding/org/xmlsoap/schemas;
do
javac -classpath $AXIS2_CLASSPATH $i/*.java;
javac -classpath $AXIS2_CLASSPATH $i/impl/*.java;
done

# Create the Jar file
jar cvf ego.jar ./codegen ./schemaorg_apache_xmlbeans/

# Use the generated jar file in classpath of your application
exit 0
```

CHAPTER

2

Creating EGO Client Projects

What is a client?

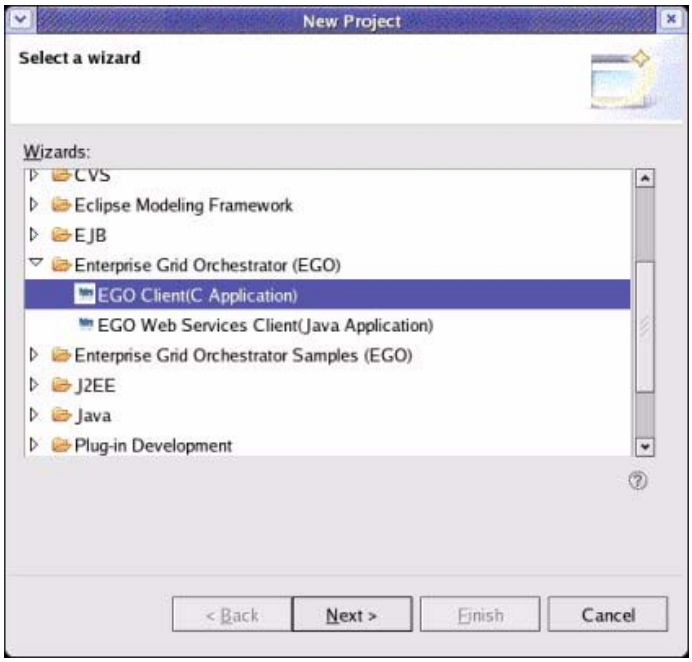
A client is an applicaton written in either the C programming language or as a Web Service, that communicates with Platform EGO through the EGO API for the purpose of querying for information, requesting and managing computing resources, and monitoring resource loading, amongst others.

Contents

Create a C client project

1. Launch Eclipse.
2. Select **File > New > Project**.
3. In the New Project dialog, expand **Enterprise Grid Orchestrator (EGO)** and select **EGO Client (C Application)**. Click **Next**.
4. In the **EGO C Project** dialog, enter a project name. Click **Finish** (to use default C project settings) or click **Next** to adjust the settings. Click **Finish** when settings are complete.

   Eclipse displays the C/C++ perspective.

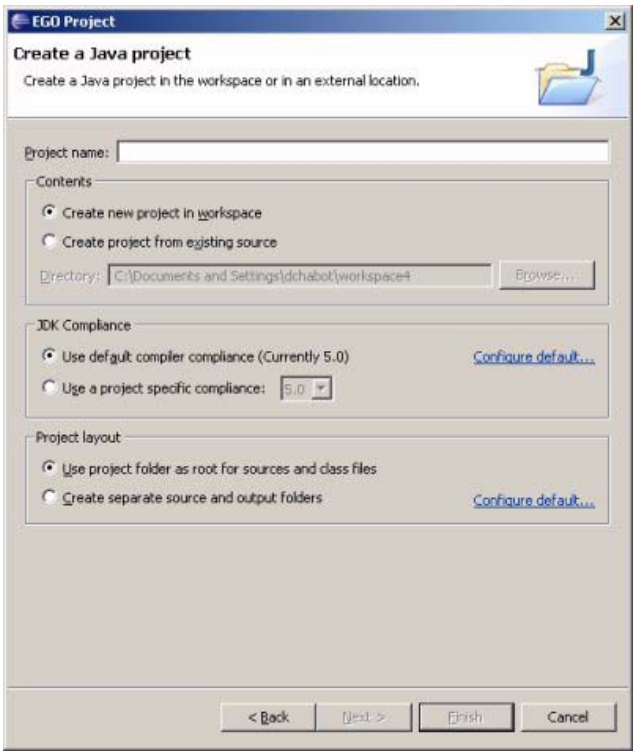


Create a Web Service project

Prerequisites: In order to develop code compliant with J2SE 5.0, you will need the J2SE 5.0 Java Runtime Environment (JRE).

1. Launch Eclipse.
2. Select **File > New > Project**.
3. In the New Project dialog, expand **Enterprise Grid Orchestrator (EGO)** and select **EGO Web Services Client (Java Application)**. Click **Next**.
4. In the EGO Project dialog, enter a project name. Set the JDK Compliance level to 5.0. Click **Finish** (to use default Java project settings) or click **Next** to adjust the settings. Click **Finish** when settings are complete. .

   Eclipse displays the Java perspective.



CHAPTER

3

Getting Started with the C Client: A Collection of Tutorials

Before you begin the tutorials

Before starting these tutorials, ensure that the EGO C API plug-in is installed in Eclipse and the EGO runtime has been installed, configured, and running on a host cluster. The sample programs use host name and port numbers to communicate with the master host. This data is stored in a configuration file (ego.conf) provided with the code samples. The configuration file must be updated so that the host name and port numbers match the values configured in the master host. Consult your cluster administrator to determine the configuration details of the master host.

Contents

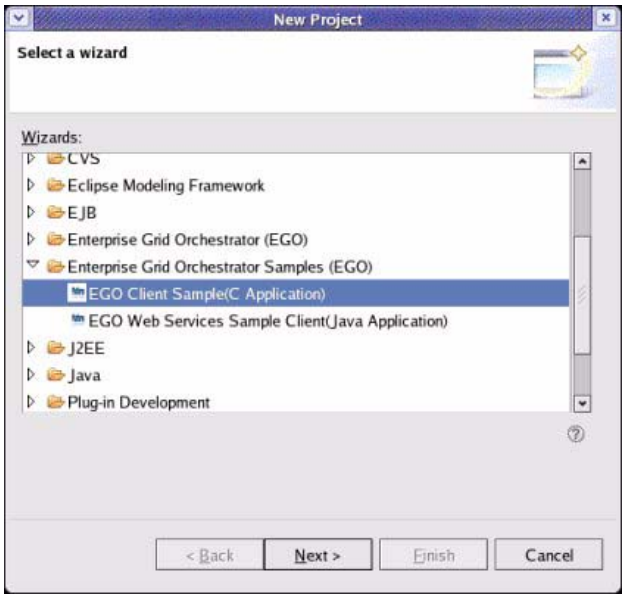
Locate the code samples

1. Launch Eclipse.
2. Select **File > New > Project**.
3. In the New Project dialog, expand **Enterprise Grid Orchestrator Samples (EGO)** and select **EGO Client Sample (C Application)**. Click **Next**.
4. In the EGO C Sample Project dialog, enter a project name. To use default C project settings, click **Finish**, or, to adjust the settings, click **Next** . Click **Finish** when settings are complete.

   If Eclipse asks you if you want to open the C/C++ perspective, click **Yes**.
5. In the Navigator view, expand the project to see the list of code samples.

   Note: If the Navigator is not visible, in the Window menu, select **Show View > Navigator.**
6. Double-click the sample file.

   The sample code appears in the main view.



Tutorial 1: Request Information About Hosts in a Cluster

This tutorial describes the minimum amount of code required to create an unregistered EGO client that connects to a host cluster.

Using this tutorial, you will …

- Open a connection to Platform EGO
- Print out cluster info
- Initialize structures to hold host information
- Retrieve and print out info about all hosts in a cluster
- Retrieve and print out host summary (host availability and utilization) information

Step 1: Preprocessor directives

The first step is to include a reference to the system and API header files. The samples.h header file contains the declaration of methods that are implemented in the samples.

```
#include <stdlib.h>
#include <stdio.h>
#include "vem.api.h"
#include "samples.h"
```

Step 2: Implement the principal method

Lines: 4-10: define and initialize a data structure that is used to request a connection with the EGO host cluster. The data structure contains a reference to a configuration file where the master host name and port numbers are stored.

Line 17: the data structure is passed as an argument to the vem_open () method, which opens a connection to the master host. If the connection attempt is successful, a handle is returned; otherwise the method returns NULL. The handle, which is unique to each client, acts as a communication channel to the master host and all subsequent communication occurs through this handle.

Lines 25-33: the vem_name_t structure is initialized with NULL. This structure holds the cluster name, system name, and version. The vem_uname () method is passed the communication handle and, if successful, returns a valid vem_name_t structure (defined as clusterName); otherwise the method returns NULL.

Lines 34 and 35: the cluster info is printed out and the memory allocated to the structure (clusterName) is freed.

Lines 37-42: define and initialize a data structure that holds a list of host names. The hostlist member is initialized with NULL to indicate all hosts should be queried.

```
1   int
2   sample1()
3   {
4     vem_openreq_t orequest;
5     vem_handle_t *vhandle = NULL;
6     /* setup the open request structure with the filename
7      * that includes the cluster information: master host and port
8      */
9     orequest.file = "ego.conf"; // configuration file
10    orequest.flags=0;
11
12    /* this opens a connection to the vemkd using the master host
13     * and port specified in the configuration file. Returns NULL
14   * if unsucessful. All other interactions occur through this
15     * vem_handle_t returned.
16     */
17    vhandle = vem_open(&orequest);
18
19    if (vhandle == NULL) {
20    // error opening
21     fprintf(stderr, "Error opening cluster: %s\n",  vem_strerror(vemerrno));
22     return -1;
23    }
24
25    vem_name_t *clusterName = NULL;
26
27    /* Retrieves the identification of the cluster, name and version */
28    clusterName = vem_uname(vhandle);
29    if (clusterName == NULL) {
30    // error connecting
31     fprintf(stderr, "Error connecting to cluster: %s\n",  vem_strerror(vemerrno));
32     return -2;
33    }
34    printf(" Connected... %s %s %4.2f\n\n", clusterName->clustername,
    clusterName->sysname, clusterName->version);
35    vem_free_uname(clusterName); // free memory
36
37    int hin;
38    vem_hostinforeq_t hinforeq;
39    hinforeq.resreq = " ";        // resource string, unimplemented
40    hinforeq.hostlist = NULL;     // all hosts
41    vem_hostinfo_t *hinfo = NULL; // out parameter, set by vem_getHostInfo
42    char **attrs = NULL;          // out parameter, set by vem_getHostInfo
```

Lines 46-49: the vem_getHostInfo () method retrieves information such as hostname, status, and attributes from all hosts in the cluster. If successful, the method returns the number of hosts; otherwise it returns a negative value and an error is flagged.

Lines 52-53: the print_hostInfo () method prints out the host information. Once the host information is printed, memory allocated to the information is freed.

Lines 56-72: the host summary structure is defined and passed to the vem_getHostSummary () method, which retrieves brief information about host availability and utilization. If the method call is successful, a positive integer is returned and the host summary info is printed out and the memory allocated to the host summary structure is freed. The sample1 program then closes the connection to the master host and terminates.

```
43   /* Retrieves information about all hosts in the cluster
44     * returns the number of hosts, negative if unsuccessful
45     */
46    hin = vem_getHostInfo(vhandle, &hinforeq, &hinfo, &attrs);
47    if (hin < 0) {
48    // error
49  fprintf(stderr, "Error getting hostinfo: %s %d\n",  vem_strerror(vemerrno), hin);
50    } else {
51      // print the host names and attributes
52      print_hostInfo(hin, hinfo, attrs);
53      vem_free_hostinfo(hinfo, hin, attrs);
54    }
55
56    vem_hostsummary_t hsummary;
57
58    /* get brief information about the host availabilty and utilization (in 10%
    intervals)
59     * returns negative on error.
60     */
61    int rc = vem_getHostSummary(vhandle, &hsummary);
62    if (rc < 0) {
63     fprintf(stderr, "Error getting hostsummary: %s\n",  vem_strerror(vemerrno));
64    } else {
65      print_hostsummary(&hsummary);
66      vem_clear_hostsummary(&hsummary);
67    }
68
69    /* this closes the connection to the vemkd */
70    vem_close(vhandle);
71    return 0;
72  }
```

Step 3: Send host information to the console

This method prints out the host names and status. Lines 97-107 iterate through the attrs[] array, printing the attribute names, followed by a for loop that prints out the attribute value for each host.

```
73 /**
74  * Prints vem_hostinfo_t type array with hin elements
75  */
76 void
77 print_hostInfo(int hin, vem_hostinfo_t *hinfo, char **attrs)
78 {
79   int i=0, j=0;
80   printf("%-12s\t", "Attribute");
81   for (i=0; i < hin; i++) {
82     printf("%-12s\t", hinfo[i].name);
83   }
84   printf("\n");
85
86
87   printf("%-12s\t", "Status");
88   for (i=0; i < hin; i++) {
89     print_host_status(hinfo[i].status);
90   }
91   printf("\n");
92
93   j=0;
94   while(attrs[j] != NULL) {
95     printf("%-12s\t", attrs[j]);
96     vem_value_t value;
97     for (i=0; i < hin; i++) {
98       value = hinfo[i].attributes[0];
99       int status = value.value.v_int32;
100      if(status != HOST_OK) {
101        value.type = VEM_TYPE_STRING;
102        value.value.v_string = " ";
103      } else {
104        value = hinfo[i].attributes[j];
105      }
106      print_vem_value(&value);
107    }
108    j++;
109    printf("\n");
110  }
111 }
```

Step 4: Get the host status

This method formats the host status message for printing to the console.

```
void
print_host_status(int status)
{
  switch(status) {
    case HOST_OK:      printf("%-12s\t", "Ok"); break;
   case HOST_UNAVAIL: printf("%-12s\t", "Unavailable"); break;
    case HOST_CLOSE:   printf("%-12s\t", "Closed"); break;
    case HOST_REMOVED: printf("%-12s\t", "Removed"); break;
  }
}
```

Step 5: Format output according to data type

This method formats each attribute value according to its data type so that it is properly displayed.

```
void
print_vem_value(vem_value_t *vem_value)
{
  switch(vem_value->type) {
    case VEM_TYPE_NULL:    printf("%-12s\t",                    "NULL"); break;
    case VEM_TYPE_CHAR:    printf("%-12c\t",   vem_value->value.v_char);    break;
   case VEM_TYPE_UCHAR:  printf("%-12c\t",  vem_value->value.v_uchar);  break;
    case VEM_TYPE_INT16:   printf("%-12d\t",   vem_value->value.v_int16);   break;
    case VEM_TYPE_UINT16:  printf("%-12u\t",   vem_value->value.v_uint16);  break;
    case VEM_TYPE_INT32:   printf("%-12d\t",   vem_value->value.v_int32);   break;
   case VEM_TYPE_UINT32:  printf("%-12u\t",  vem_value->value.v_uint32); break;
    case VEM_TYPE_INT64:   printf("%-12lld\t", vem_value->value.v_int64);   break;
    case VEM_TYPE_UINT64:  printf("%-12llu\t", vem_value->value.v_uint64);  break;
    case VEM_TYPE_FLOAT32: printf("%-12.2f\t",   vem_value->value.v_float32); break;
    case VEM_TYPE_FLOAT64: printf("%-12.2lf\t",  vem_value->value.v_float64); break;
    case VEM_TYPE_BOOL:    printf("%-12d\t",   vem_value->value.v_bool);    break;
    case VEM_TYPE_TIME:    printf("%-12ld\t",  vem_value->value.v_time);    break;
    case VEM_TYPE_STRING:  printf("%-12s\t",   vem_value->value.v_string);  break;
    case VEM_TYPE_PTR:     printf("%-12p\t",   vem_value->value.v_ptr);     break;
  }
}
```

Step 6: Send host summary to the console

This method is passed the host summary structure. The method iterates through the structure's arrays and prints out a list of host status definitions and the number of hosts corresponding to each status definition. The method also prints out the utilization intervals, i.e., 10%, 20%, etc., and the number of hosts that correspond to each interval.

```
void
print_hostsummary(vem_hostsummary_t *hsummary)
{
int i;
printf("\nHost Summary:\n");
for(i=0; i<hsummary->statusC; i++){
  printf("%-12s %d\n", hsummary->statusV[i], hsummary->statusSummary[i]);
}
printf("\nUT Summary:\n");
for(i=0; i<hsummary->utC; i++){
  printf("%-12s %d\n", hsummary->utV[i], hsummary->utSummary[i]);
}
}
```

Run the client application

1. Select **Run > Run**.

   The Run dialog appears.
2. In the Configurations list, either select an EGO C Client Application or click **New** for a new configuration.

   For a new configuration, enter the configuration name.
3. Enter the project name and C/C++ Application name.
4. Click **Apply** and then **Run**.

Sample output

```
 Connected... cluster1 hb01b13 1.00

Attribute       hb01b14         hb01b16         mfrisch1        hb01b13
Status          Ok              Ok              Unavailable     Ok
status          0               0                               0
type            NTX86           NTX86                           NTX86
model           PC450           PC450                           PC450
ncpu            1               1                               1
cpuf            13.20           13.20                           13.20
mem             782.00          761.00                          520.00
swp             2300.00         2290.00                         1914.89
maxmem          1023            1023                            1023
maxswp          2462            2462                            2462
tmp             33792.00        32112.00                        23604.00
ut              0.00            0.00                            0.05
it              1105.00         1099.00                         1123.92
io              8.48            6.45                            104.99
pg              0.03            0.00                            1.78
r1m             2.79            2.09                            2.15
r15s            2.77            2.04                            2.01
r15m            2.62            2.25                            3.00
ls              1.00            1.00                            1.00
slot            1               1                               1
freeslot        1               1                               1
```

```
Host Summary:
OK              3
UNAVAIL         1
CLOSED          0
REMOVED         0

UT Summary:
10%             3
20%             0
30%             0
40%             0
50%             0
60%             0
70%             0
80%             0
90%             0
100%            0
```

Tutorial 2: Request Host Allocation in a Cluster with Synchronous Notifications

This tutorial describes how to create a registered EGO client that requests host allocation in a cluster and starts a container on the host. The client also reads notifications synchronously from the cluster regarding resource changes.

Using this tutorial, you will …

- Open a connection to Platform EGO
- Print out cluster information
- Check if there are any registered clients connected to Platform EGO
- Log on to Platform EGO
- Register the client with Platform EGO
- Print out allocation and container reply info from a previous connection
- Print out host group information
- Request resource allocation from Platform EGO and print out the allocation ID
- Check for an incoming resource allocation message from Platform EGO on the open connection and print message
- Start a container on Platform EGO and print out the container ID
- Check for registered clients connected to Platform EGO and print out information

Step 1: Preprocessor directives

The first step is to include a reference to the system and API header files. The samples.h header file contains the declaration of methods that are implemented in the samples.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include "vem.api.h"
#include "samples.h"
```

Step 2: Implement the principal method

Lines 4-8: define and initialize a data structure that is used to request a connection with the EGO host cluster. The data structure contains a reference to a configuration file where the master host name and port numbers are stored.

Line 10: pass the data structure as an argument to the vem_open () method, which opens a connection to the master host. If the connection attempt is successful, a handle is returned; otherwise the method returns NULL. The handle acts as a communication channel to the master host and all subsequent communication occurs through this handle.

Lines 18-19: the vem_name_t structure is initialized with NULL. This structure holds the cluster name, system name, and version. The vem_uname () method is passed the communication handle and, if successful, returns a valid vem_name_t structure (defined as clustername); otherwise the method returns NULL.

Line 26: the cluster info is printed out to the screen.

Lines 29-46: define the client info structure. Use vem_locate() to get all registered clients. Since NULL is provided as the client name, all registered clients will be located and the method returns the number of registered clients. Note that Platform EGO is equipped with a number of default clients (services) such as the Service Controller, so as a minimum, the info relevant to these clients is printed out and the associated memory is released.

Lines 47-49: authenticate the user to Platform EGO.

```
1   int
2   sample2()
3   {
4     vem_openreq_t orequest;
5     vem_handle_t *vhandle = NULL;
6
7     orequest.file = "ego.conf"; // default libvem.conf
8     orequest.flags=0;
9
10    vhandle = vem_open(&orequest);
11
12    if (vhandle == NULL) {
13    // error opening
14     fprintf(stderr, "Error opening cluster: %s\n",  vem_strerror(vemerrno));
15     return -1;
16    }
17
18    vem_name_t *clusterName = NULL;
19    clusterName = vem_uname(vhandle);
20    if (clusterName == NULL) {
21    // error connecting
22     fprintf(stderr, "Error connecting to cluster: %s\n",  vem_strerror(vemerrno));
23     return -2;
24    }
25
26    fprintf(stdout, " Connected... %s %s %4.2f\n", clusterName->clustername,
27   clusterName->sysname, clusterName->version);
28
29    vem_clientinfo_t *clients;
30    int  rc = vem_locate(vhandle, NULL, &clients);
31    if (rc >=0) {
32      if (rc == 0) {
33       printf("No registered clients exist\n");
34      } else {
35      int i=0;
36      for (i=0; i<rc; i++) {
37         printf("%s %s %s\n", clients[i].name, clients[i].description,
38         clients[i].location);
39      }
40      // free
41      vem_clear_clientinfo(clients);
42      }
43    } else {
44    // error connecting
45     fprintf(stderr, "Error geting clients: %s\n",  vem_strerror(vemerrno));
46    }
47  } if (login(vhandle, username, password)<0) {
48     fprintf(stderr, "Error logon: %s\n",  vem_strerror(vemerrno));
49    }
```

Lines 50-63: define the vem_allocation_info_reply_t and vem_container_info_reply_t structures. If a client gets disconnected and then re-registers, its existing allocations and containers are returned to these structures. If the client had never registered before, the structures would be empty. Define and initialize a structure (rreq) that holds client info for registration purposes. Note that on line 58, the callback member (cb) is set to NULL. This means that it is the client's

responsibility to periodically check the open connection via vem_select()/vem_read() to get incoming messages and take action accordingly. Register with Platform EGO via the open connection using vem_register().

Lines 64-67: print out information related to the allocation requests and containers. Once the info is printed out, the memory for the allocations is freed.

Lines 73-79: the method collects the information for the requested hostgroup. In this case, the requested hostgroup in the input argument is set to NULL, which means that information about all hostgroups is requested. If the method call is successful, hostgroup information is printed out to the screen.

```
50  vem_allocation_info_reply_t aireply;
51   vem_container_info_reply_t  cireply;
52   vem_registerreq_t rreq;
53
54   rreq.name = "sample2_client";
55   rreq.description = "Sample2";
56   rreq.flags = VEM_REGISTER_TTL;
57   rreq.ttl = 3;
58   rreq.cb = NULL; // would need to read messages explicitly;
59
60   rc = vem_register(vhandle, &rreq, &aireply, &cireply);
61   if (rc < 0) {
62     fprintf(stderr, "Error registering: %s\n",  vem_strerror(vemerrno));
63   }
64 print_vem_allocation_info_reply(&aireply);
65   print_vem_container_info_reply(&cireply);
66   // freeup any previous allocations
67   release_vem_allocation(vhandle, &aireply);
68
69   vem_hostgroupreq_t hgroupreq;
70   hgroupreq.grouplist = NULL;
71   vem_hostgroup_t *hgroup;
72
73   rc = vem_gethostgroupinfo(vhandle, &hgroupreq, &hgroup);
74   if (rc < 0) {
75     fprintf(stderr, "Error getting hostgroup: %s\n",  vem_strerror(vemerrno));
76   } else {
77   printf("%s %s %d %d\n", hgroup->groupName, hgroup->members, hgroup->free,
78  hgroup->allocated);
79   }
```

Lines 80-101: initialize the data structure (vem_allocreq_t) that specifies the allocation request. Method vem_alloc() requests resource allocation using the allocation request info (vem_allocreq_t structure) as one of the input arguments. If the request is successful, the allocation ID is printed out to the screen.

```
80    vem_allocreq_t areq;
81    areq.name = "Sample2Alloc";
82    areq.consumer = "/SampleApplications/EclipseSamples”;
83    areq.hgroup = "ComputeHosts";
84  #ifndef WIN32_RESOURCE
85    areq.resreq = "LINUX86";
86  #else
87    areq.resreq = "NTX86";
88  #endif
89    areq.minslots = 1;
90    areq.maxslots = 1;
91    areq.tile = 0;
92  vem_allocation_id_t alocid;
93    vem_allocfreereq_t afree;
94    rc = vem_alloc(vhandle, &areq, &alocid);
95    if (rc < 0) {
96      fprintf(stderr, "Error allocating: %s\n",  vem_strerror(vemerrno));
97      goto bailout;
98
99    } else {
100     printf("allocated: %s\n",  alocid);
101   }
```

Lines 102-123: define and initialize a container specification including the setting of its resource limits to default values. The container specification essentially defines a job that the user wants to be executed. The conspec.command method specifies the actual binary that should be executed. In the sample, we want the program "sleep" to be executed. The UNIX sleep command takes the number of seconds to sleep as an input argument.

```
102  vem_container_spec_t conspec;
103   memset(&conspec, 0, sizeof(vem_container_spec_t));
104 #ifndef WIN32_RESOURCE
105   conspec.command = "sleep 240";
106   conspec.execUser = "lsfadmin"; // "egoadmin";
107   conspec.umask = 0777;
108   conspec.execCwd = "/tmp";
109   conspec.envC = 0;
110 #else
111   // sleep needs to be installed on the cluster NT hosts
112   // or if ping is available, use something like ping -n xxx 127.0.0.1 > nul
113   conspec.command = "sleep 240";
114   conspec.execUser = "lsf\\lsfadmin"; //"egouser"; // "lsfadmin"; // "egoadmin";
115   conspec.umask = 0777;
116   conspec.execCwd = "c:\\";
117   conspec.envC = 0;
118 #endif
119   int i;
120   for (i=0; i<VEM_RLIM_NLIMITS; i++) {
121   conspec.rlimits[i].rlim_cur = VEM_RLIM_DEFAULT;
122     conspec.rlimits[i].rlim_max = VEM_RLIM_DEFAULT;
123   }
```

Lines 124-130: define and initialize various structures and assign container and allocation IDs.

Lines 132-163: check to see if there is any incoming data on an open connection for up to 60 seconds (configurable timeout). If successful, the message is read from the open connection. A switch statement is used to interpret the message code enumeration and the corresponding message is printed out to the screen. If the message cannot be read, free the memory for the allocation ID.

```
124 vem_startcontainerreq_t conreq;
125   vem_container_id_t      conid = NULL;
126   conreq.allocId = alocid;
127 struct timeval tv;
128   struct vem_message msg;
129   struct vem_allocreply *rep = NULL;
130   struct vem_allocreclaim *reclaim = NULL;
131
132   tv.tv_sec = 60; // 60 seconds timeout
133   rc = vem_select(vhandle, &tv);
134   if(rc < 0) {
135      printf("vem_select error\n");
136      goto cleanup;
137   }
138   if(rc == 0) {
139      printf("vem_select may have problem, please set longer timeout \n");
140      goto cleanup;
141   }
142   rc = vem_read(vhandle, &msg);
143   if(rc < 0) {
144      printf("Read message failed\n");
145      goto cleanup;
146   }
147   switch(msg.code) {
148      case RESOURCE_ADD:
149           rep = (struct vem_allocreply *)msg.content;
150           printf("Got alloc reply for %s %d hosts\n", rep->consumer, rep->nhost);
151           break;
152      case RESOURCE_RECLAIM:
153           reclaim = (struct vem_allocreclaim*)msg.content;
154           printf("vem wants its resources back for allocation %s\n",
155  reclaim->reclaim->consumer);
156           rc = -1;
157           goto cleanup;
158           break;
159      default:
160           printf("unknown message code %d\n", msg.code);
161           goto cleanup;
162           break;
163   } /* switch() */
```

Lines 164-168: get the hostname for the allocation and print it out to the screen. Initialize the workload container request structure (conreq) with the hostname, container name, and the container specification (conspec).

Lines 170-175: start the workload container on the specified host and, if successful, print out the container ID.

Lines 178-193: use vem_locate() to get all registered clients. Since NULL is provided as the client name, all registered clients will be located and the method returns the number of registered clients. Note that Platform EGO is equipped with a number of default clients (services) such as the Service Controller, so as a

minimum, the info relevant to these clients is printed out and the associated memory is released. If successful, print out the client info and free the associated memory.

```
164 char *host = rep->host[0].name;
165   printf("Allocated host: %s\n",  host);
166   conreq.hostname = host;
167   conreq.name = "Sample2Container";
168   conreq.spec = &conspec;
169
170   rc = vem_startcontainer(vhandle, &conreq, &conid);
171   if (rc < 0) {
172     fprintf(stderr, "Error starting container: %s\n",  vem_strerror(vemerrno));
173     goto cleanup;
174   }
175   printf("Started container %s\n", conid);
176 // Currently no way to get container from id.
177   //print_vem_container(vem_container_t *container);
178 rc = vem_locate(vhandle, NULL, &clients);
179   if (rc >=0) {
180     if (rc == 0) {
181      printf("No registered clients exist\n");
182     } else {
183     int i=0;
184     for (i=0; i<rc; i++) {
185        printf("%s %s %s\n", clients[i].name, clients[i].description,
186        clients[i].location);
187     }
188       vem_clear_clientinfo(clients);
189     }
190   } else {
191   // error connecting
192    fprintf(stderr, "Error geting clients: %s\n",  vem_strerror(vemerrno));
193   }
194 // wait for job to finish
195 #ifdef WIN32
196     Sleep(60000);
197 #else
198     sleep(30);
199 #endif
```

```
200 cleanup:
201   afree.allocId = alocid;
202   rc = vem_allocfree(vhandle, &afree);
203   if (rc < 0) {
204     fprintf(stderr, "Error freeing allocation: %s\n",  vem_strerror(vemerrno));
205   }
206 bailout:
207   rc = vem_unregister(vhandle);
208   if (rc < 0) {
209     fprintf(stderr, "Error unregistering: %s\n",  vem_strerror(vemerrno));
210   }
211 if (logout(vhandle)<0) {
212    fprintf(stderr, "Error logoff: %s\n",  vem_strerror(vemerrno));
213   }
214 // free memory
215   vem_free_containerId(conid);
216   //vem_free_containerSpec(&conspec);  // crashes
217
218 leave:
219   vem_free_uname(clusterName);
220   vem_close(vhandle);
221 if(host != NULL)
222   free(host);
223
224   return 0;
225 }
```

Step 3: Free all resource allocations

This method iterates through each allocation, as identified by its allocation ID, and frees its memory. Freeing an allocation is the same as cancelling it, i.e., all resources associated with the allocation are released.

```
void
release_vem_allocation(vem_handle_t *vhandle, vem_allocation_info_reply_t *aireply)
{
int i;
for(i=0; i<aireply->nallocation; i++){
  // free alocid memory
      vem_allocfreereq_t afree;
      afree.allocId = aireply->allocation[i].allocId;
      int rc = vem_allocfree(vhandle, &afree);
      if (rc < 0) {
    fprintf(stderr, "Error freeing allocation: %s\n",  vem_strerror(vemerrno));
      }
}
}
```

Step 4: Print allocation info

These three methods iterate through each allocation, printing out the allocation ID, allocation request info, host name, host slots, and a list of host attributes.

```
void
print_vem_allocation_info_reply(vem_allocation_info_reply_t *aireply)
{
int i;
for(i=0; i<aireply->nallocation; i++){
print_vem_allocation(&aireply->allocation[i]);
}
}

void
print_vem_allocation(vem_allocation_t *alloc)
{
printf("AllocId=%s\n", alloc->allocId);
print_vem_allocreq(alloc->allocReq);
    int i, j;
    for(i=0; i<alloc->nhost; i++){
    printf("Name=%s Slots=%d Attributes ",
      alloc->host[i].name,
      alloc->host[i].slots);
    for(j=0; j<alloc->hostattr[i].attrC; j++){
      vem_attribute_t *attr = &alloc->hostattr[i].attrV[j];
      printf("%s=", attr->name);
      print_vem_value(&attr->value_t);
    }
    printf("\n");
    }
}
void
print_vem_allocreq(vem_allocreq_t *allocreq)
{
printf("AllocReq %s %s %s %s %d %d %d\n",
allocreq->name,
allocreq->consumer,
allocreq->hgroup,
allocreq->resreq,
allocreq->maxslots,
allocreq->minslots,
allocreq->flags
);
}
```

Step 5: Print container info

These four methods iterate through each container, printing out the container ID, state, and other container-related fields. The print_vem_container_state () and print_vem_container_exit_reason methods () use switch statements to interpret the meaning of the enumeration members.

```
void
print_vem_container_info_reply(vem_container_info_reply_t  *cireply)
{
int i;
for(i=0; i<cireply->ncontainer; i++){
print_vem_container(cireply->container);
}
}
void
print_vem_container(vem_container_t *container)
{
printf("Container\n");
printf("Id=%s\nState=",container->id);
print_vem_container_state(container->state);
printf("\nName=%s\nAllocId=%s\nConsumer=%s Start=%ld, End=%ld\nHost=%s ExitStatus=%d
ExitReason=",
  container->name,
  container->allocId,
  container->consumer,
  container->startTime,
  container->endTime,
  container->host,
  container->exitStatus);
print_vem_container_exit_reason(container->exitReason);
//TODO add the rest of the fields
// print rest
}
void
print_vem_container_state(vem_container_state_t state)
{
switch(state) {
      case CONTAINER_NULL:       printf(" 0, internal state"); break;
      case CONTAINER_START:      printf(" 1, start"); break;
      case CONTAINER_RUN:        printf(" 2, running"); break;
      case CONTAINER_SUSPEND:    printf(" 3, suspend"); break;
      case CONTAINER_FINISH:     printf(" 4, finish"); break;
      case CONTAINER_UNKNOWN:    printf(" 5, unknown, host unreachable "); break;
     case CONTAINER_ZOMBIE:    printf(" 6, zombie, unknown container is terminated");
 break;
      case CONTAINER_MAX_STATE:  printf(" Number of container state"); break;
}
}
```

```
void
print_vem_container_exit_reason (vem_container_exit_reason_t rcode)
{
switch(rcode) {
     case ER_NULL:                printf("  0, no reason"); break;
      case ER_SETUP_NO_MEM:          printf("  1, exit bacause of setup fail");break;
      case ER_SETUP_FORK:            printf("  2, fork fail");break;
      case ER_SETUP_PGID:            printf("  3, fail to setpgid"); break;
      case ER_SETUP_ENV:             printf("  4, fail to set env variables");break;
      case ER_SETUP_LIMIT:           printf("  5, fail to set process limits");break;
      case ER_SETUP_NO_USER:         printf("  6, user account doesn't exist");break;
     case ER_SETUP_PATH:           printf("  7, fail to change container cwd");break;
      case ER_SIG_KILL:              printf("  8, terminated by sigkill");break;
      case ER_UNKNOWN:               printf("  9, unknown reason ");break;
      case ER_PEM_UNREACH:           printf("  10, fail to reach pem host");break;
      case ER_PEM_SYN:               printf("  11, vemkd and pem sync issue");break;
case ER_BAD_ALLOC_HOST:        printf("  14, host is not allocated");break;
      case ER_NOSUCH_CLIENT:         printf("  15, client doesn't exist");break;
      case ER_START:                 printf("  16, container start fails");break;
      case LAST_EXIT_REASON:         printf(" last exit reason ");break;
}
printf("\n");
}
```

Run the client application

1. Select **Run > Run**.

   The Run dialog appears.
2. In the Configurations list, either select an EGO C Client Application or click **New** for a new configuration.

   For a new configuration, enter the configuration name.
3. Enter the project name and C/C++ Application name.
4. Click **Apply** and then **Run**.

Sample output

```
 Connected... demo2 archtsmst.noam.corp.platform.com 1.00
sample7_client Sample7 Client 53182@172.25.244.92
sample6_client Sample6 Client
sample5_client Sample5 Client
moviestore MovieStoreController 55358@172.25.244.143
EGO_SERVICE_CONTROLLER url=archtsmst.noam.corp.platform.com:7872 47408@172.25.244.143
Compute egonode-a egonode-b egonode-c egonode-d  12 4
allocated: 30
Got alloc reply for  1 hosts
AllocId=30 Consumer= Hosts=1
Name=egonode-c Slots=1 Attributes: cpuf=8.30           type=LINUX86
Allocated host: egonode-c
Started container 376
sample2_client Sample2 55252@172.25.244.92
sample7_client Sample7 Client 53182@172.25.244.92
sample6_client Sample6 Client
sample5_client Sample5 Client
moviestore MovieStoreController 55358@172.25.244.143
EGO_SERVICE_CONTROLLER url=archtsmst.noam.corp.platform.com:7872 47408@172.25.244.143
```

Tutorial 3: Request Host Allocation in a Cluster with Asynchronous Callback Notifications

This tutorial describes how to create a registered EGO client that requests host allocation in a cluster and starts a container on the host. The sample uses callbacks for notifications from the cluster about resource change and container/host state change.

Using this tutorial, you will …

- Open a connection to Platform EGO
- Print out cluster information
- Check if there are any registered clients connected to Platform EGO
- Log on to Platform EGO
- Register the client with Platform EGO
- Print out allocation and container reply info from a previous connection
- Print out host group information
- Request resource allocation from Platform EGO and print the allocation ID
- Start a container on Platform EGO and print container ID
- Check for registered clients connected to Platform EGO and print out information
- Implement client callback methods.

Step 1: Preprocessor directives and method declarations

The first step is to include a reference to the system and API header files. The samples.h header file contains the method declarations that are common to all of the samples. In addition, we declare the methods that are specific to this sample.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include "vem.api.h"
#include "samples.h"

static int addResourceCB(vem_allocreply_t *areply);
static int reclaimForceCB(vem_allocreclaim_t *areclaim);
static int containerStateChgCB(vem_containerstatechg_t *cschange);
static int hostStateChangeCB(vem_hoststatechange_t *hschange);
// holds allocation information
static vem_allocreply_t *allocReply = NULL;
static char *allocated_host_name = NULL;
static int barrier = 0;
static vem_container_id_t jobContainerId = NULL;
static int jobFinished = 0;
```

Step 2: Implement the principal method

Lines 4-7: define and initialize a data structure that is used to request a connection with the EGO host cluster. The data structure contains a reference to a configuration file where the master host name and port numbers are stored.

Line 8: pass the data structure as an argument to the vem_open () method, which opens a connection to the master host. If the connection attempt is successful, a handle is returned; otherwise the method returns NULL. The handle acts as a communication channel to the master host and all subsequent communication occurs through this handle.

Lines 14-15: the vem_name_t structure (defined as clusterName) is initialized with NULL. This structure holds the cluster name, system name, and version. The vem_uname () method is passed the communication handle and, if successful, returns a valid vem_name_t structure ; otherwise the method returns NULL

Line 21: the cluster info is printed out to the screen.

Lines 22-39: define the client info structure. Use vem_locate() to get all registered clients. Since NULL is provided as the client name, all registered clients will be located and the method returns the number of registered clients. Note that Platform

EGO is equipped with a number of default clients (services) such as the Service Controller, so as a minimum, the info relevant to these clients is printed out and the associated memory is released.

```
1  int
2  sample3()
3  {
4  vem_openreq_t orequest;
5  vem_handle_t *vhandle = NULL;
6  orequest.file = "ego.conf"; // default libvem.conf
7  orequest.flags=0;
8  vhandle = vem_open(&orequest);
9  if (vhandle == NULL) {
10 // error opening
11 fprintf(stderr, "Error opening cluster: %s\n",  vem_strerror(vemerrno));
12 return -1;
13 }
14 vem_name_t *clusterName = NULL;
15 clusterName = vem_uname(vhandle);
16 if (clusterName == NULL) {
17 // error connecting
18 fprintf(stderr, "Error connecting to cluster: %s\n",  vem_strerror(vemerrno));
19 return -2;
20 }
21  fprintf(stdout, " Connected... %s %s %4.2f\n", clusterName->clustername,
   clusterName->sysname, clusterName->version);
22 vem_clientinfo_t *clients;
23   int  rc = vem_locate(vhandle, NULL, &clients);
24   if (rc >=0) {
25     if (rc == 0) {
26      printf("No registered clients exist\n");
27     } else {
28     int i=0;
29     for (i=0; i<rc; i++) {
30        printf("%s %s %s\n", clients[i].name, clients[i].description,
31        clients[i].location);
32     }
33     // free
34     vem_clear_clientinfo(clients);
35     }
36   } else {
37   // error connecting
38    fprintf(stderr, "Error geting clients: %s\n",  vem_strerror(vemerrno));
39   }
```

Lines 40-42: authenticate the user to Platform EGO.

Lines 43-47: define and initialize a structure for callback methods. These callback methods are invoked by Platform EGO when resources are added or reclaimed, or when a change occurs to host status or a container. When Platform EGO wants to communicate about these events, it invokes these methods thereby calling back to the client.

Lines 48-59: Define the vem_allocation_info_reply_t and vem_container_info_reply_t structures. If a client gets disconnected and then re-registers, its existing allocations and containers are returned to these structures. If the client had never registered before, the structures would be empty. Define and initialize a structure (rreq) that holds client info for registration purposes. (This

includes assigning the client callback structure (cbf) to the callback member of the rreq structure; see Step 3: Client callback methods on page 56.) Register with Platform EGO via the open connection using vem_register().

```
40  if (login(vhandle, username, password)<0) {
41    fprintf(stderr, "Error logon: %s\n",  vem_strerror(vemerrno));
42   }
43   vem_clientcallback_t cbf;
44   cbf.addResource = addResourceCB;
45   cbf.reclaimForce = reclaimForceCB;
46   cbf.containerStateChg = containerStateChgCB;
47   cbf.hostStateChange = hostStateChangeCB;
48  vem_allocation_info_reply_t aireply;
49   vem_container_info_reply_t  cireply;
50   vem_registerreq_t rreq;
51 rreq.name = "sample3_client";
52   rreq.description = "Sample3";
53   rreq.flags = VEM_REGISTER_TTL;
54   rreq.ttl = 3;
55   rreq.cb = &cbf; // NULL, would need to read messages explicitly;
56  rc = vem_register(vhandle, &rreq, &aireply, &cireply);
57   if (rc < 0) {
58     fprintf(stderr, "Error registering: %s\n",  vem_strerror(vemerrno));
59  }
```

Lines 60-63: print out information related to the allocation requests and containers. Once the info is printed out, the memory for the allocations is freed.

Lines 65-75: the vem_gethostgroupinfo() method collects the information for the requested hostgroup. In this case, the requested hostgroup in the input argument is set to NULL, which means that information about all hostgroups is requested. If the method call is successful, hostgroup information is printed out to the screen.

Lines 76-96: initialize the data structure (vem_allocreq_t) that specifies the allocation request. vem_alloc() requests resource allocation using the allocation request info (vem_allocreq structure) as one of the input arguments. If the request is successful, the allocation ID is printed out to the screen.

```
60 print_vem_allocation_info_reply(&aireply);
61   print_vem_container_info_reply(&cireply);
62   // freeup any previous allocations
63   release_vem_allocation(vhandle, &aireply);
64
65   vem_hostgroupreq_t hgroupreq;
66   hgroupreq.grouplist = NULL;
67   vem_hostgroup_t *hgroup;
68   rc = vem_gethostgroupinfo(vhandle, &hgroupreq, &hgroup);
69   if (rc < 0) {
70     fprintf(stderr, "Error getting hostgroup: %s\n",
71  vem_strerror(vemerrno));
72   } else {
73   printf("%s %s %d %d\n", hgroup->groupName, hgroup->members, hgroup->free,
74  hgroup->allocated);
75   }
76   vem_allocreq_t areq;
77   areq.name = "Sample2Alloc";
78   areq.consumer = "/SampleApplications/EclipseSamples";
79   areq.hgroup = "ComputeHosts";
80 #ifndef WIN32_RESOURCE
81   areq.resreq = "LINUX86";
82 #else
83   areq.resreq = "NTX86";
84 #endif
85   areq.minslots = 1;
86   areq.maxslots = 1;
87   areq.flags = VEM_ALLOC_EXCLUSIVE;
88   vem_allocation_id_t alocid;
89   vem_allocfreereq_t afree;
90   rc = vem_alloc(vhandle, &areq, &alocid);
91   if (rc < 0) {
92     fprintf(stderr, "Error allocating: %s\n",  vem_strerror(vemerrno));
93     goto bailout;
94   } else {
95     printf("allocated: %s\n",  alocid);
96   }
```

Lines 97-121: define and initialize a container specification including the setting of its resource limits to default values. The container specification essentially defines a job that the user wants to be executed. The conspec.command method specifies the actual binary that should be executed. In the sample, we want the program "sleep" to be executed. The UNIX sleep command takes the number of seconds to sleep as an input argument.

Lines 122-124: define and initialize various structures and assign container and allocation IDs.

Lines 126-135: a while loop suspends program execution until a hostname for the allocation is found. The barrier variable is set when the notification from Platform EGO arrives after which it can proceed to run a container on the allocated resource.. The hostname is printed out.

Lines 136-138: initialize the workload container request structure (conreq) with the hostname, container name, and the container specification (conspec).

```
97 vem_container_spec_t conspec;
98   memset(&conspec, 0, sizeof(vem_container_spec_t));
99
100 #ifndef WIN32_RESOURCE
101   conspec.command = "sleep 120";
102   conspec.execUser = "lsfadmin"; // "egoadmin";
103   conspec.umask = 0777;
104   conspec.execCwd = "/tmp";
105   conspec.envC = 0;
106 #else
107   // sleep needs to be installed on the cluster NT hosts
108   // or if ping is available, use something like ping -n xxx 127.0.0.1 > nul
109   conspec.command = "sleep 120";
110   conspec.execUser = "lsf\\lsfadmin"; //"egouser"; // "lsfadmin"; //
111  "egoadmin";
112   conspec.umask = 0777;
113   conspec.execCwd = "c:\\";
114   conspec.envC = 0;
115 #endif
116
117   int i;
118   for (i=0; i<VEM_RLIM_NLIMITS; i++) {
119   conspec.rlimits[i].rlim_cur = VEM_RLIM_DEFAULT;
120     conspec.rlimits[i].rlim_max = VEM_RLIM_DEFAULT;
121   }
122   vem_startcontainerreq_t conreq;
123   vem_container_id_t      conid = NULL;
124   conreq.allocId = alocid;
125   // find the hostname for allocation from the CB fn
126   while (barrier == 0) {
127   // wait until we have a host allocated
128    sleep(1);
129   }
130   if (allocReply == NULL || allocReply->nhost ==0) {
131    fprintf(stderr, "Error allocating host: %s\n",  vem_strerror(vemerrno));
132     goto cleanup;
133   }
134   char *host = allocated_host_name;
135   printf("Allocated host: %s\n",  host);
136   conreq.hostname = host; // allocReply->host[0].name;
137   conreq.name = "Sample2Container";
138   conreq.spec = &conspec;
```

Lines 139-146: start the workload container on the specified host and, if successful, print out the container ID.

Lines 147-168: use vem_locate() to get all registered clients. Since NULL is provided as the client name, all registered clients will be located and the method returns the number of registered clients. If successful, print out the client info and free the associated memory.

```
139 rc = vem_startcontainer(vhandle, &conreq, &conid);
140   if (rc < 0) { fprintf(stderr, "Error starting container: %s\n",
141  vem_strerror(vemerrno));
142     jobContainerId = "INVALID";
143     goto cleanup;
144   }
145   jobContainerId = conid;
146   printf("Started container %s\n", conid);
147 rc = vem_locate(vhandle, NULL, &clients);
148   if (rc >=0) {
149     if (rc == 0) {
150      printf("No registered clients exist\n");
151     } else {
152     int i=0;
153     for (i=0; i<rc; i++) {
154        printf("%s %s %s\n", clients[i].name, clients[i].description,
155        clients[i].location);
156     }
157       vem_clear_clientinfo(clients);
158     }
159   } else {
160   // error connecting
161    fprintf(stderr, "Error geting clients: %s\n",  vem_strerror(vemerrno));
162   }
163 // wait for job to be finished
164   while (!jobFinished) {
165   //wait
166   sleep(10);
167   }
168   vem_free_containerId(conid);
```

Step 3: Client callback methods

These callback methods are invoked by Platform EGO when resources are added or reclaimed, or when a change occurs to host status or a container. When Platform EGO wants to communicate about these events, it invokes these methods thereby calling back to the client.

Lines 169-179: this method is called by Platform EGO when resources have been added to an allocation in order to tell the client which resources have been provided for its use. This method prints out the allocation and consumer IDs, the number of hosts allocated, host names and number of slots, and host attributes.

Lines 180-186: this method is called by Platform EGO when resources need to be reclaimed. Resources may be reclaimed either for policy reasons, or because a resource has been found to be down or unavailable. The method prints out the host info including host name and slots for each host being reclaimed.

Lines 187-200: this method is called by Platform EGO in order to communicate status changes in containers to the clients that started them. The method prints out the container ID and its associated state; the container state is enumerated in the vem.common.h file.

Lines 201-207: this method is called by Platform EGO when a host changes state. The method prints out the host name and its new host state.

```
169 int
170 addResourceCB(vem_allocreply_t *areply)
171 {
172 printf("addResource Call Back\n");
173 allocReply = areply;
174 allocated_host_name = malloc(strlen(allocReply->host[0].name));
175 strcpy(allocated_host_name, allocReply->host[0].name);
176 barrier = 1;
177     print_vem_allocreply(areply);
178     return 0;
179 }
180 int
181 reclaimForceCB(vem_allocreclaim_t *areclaim)
182 {
183 printf("reclaimForce Call Back\n");
184 print_vem_allocreclaim(areclaim);
185 return 0;
186 }
187 int
188 containerStateChgCB(vem_containerstatechg_t *cschange)
189 {
190 printf("containerStateChg Call Back\n");
191 printf("%s %d\n", cschange->containerId, cschange->newState);
192  while(jobContainerId == NULL) {sleep(1);} // wait until container has been
193  created
194     if(jobContainerId && !strcmp(cschange->containerId, jobContainerId)) {
195     if(cschange->newState == CONTAINER_FINISH) {
196     jobFinished = 1;
197     }
198     }
199 return 0;
200 }
201 int
202 hostStateChangeCB(vem_hoststatechange_t *hschange)
203 {
204 printf("hostStateChange Call Back\n");
205 printf("%s %d\n", hschange->name, hschange->newState);
206     return 0;
207 }
```

Run the client application

1 Select **Run > Run**.

The Run dialog appears.

2 In the Configurations list, either select an EGO C Client Application or click **New** for a new configuration.

   For a new configuration, enter the configuration name.

3 Enter the project name and C/C++ Application name.

4 Click **Apply** and then **Run**.

Sample output

```
 Connected... demo2 archtsmst.noam.corp.platform.com 1.00
sample7_client Sample7 Client 53182@172.25.244.92
moviestore MovieStoreController 55358@172.25.244.143
EGO_SERVICE_CONTROLLER url=archtsmst.noam.corp.platform.com:7872 47408@172.25.244.143
Container
Id=376
State= 4, finish
Name=Sample2Container
AllocId=30
Consumer=StoreFront Start=1147807844, End=1147807844
Host=egonode-c ExitStatus=0 ExitReason=  6, user account doesn't exist
Compute egonode-a egonode-b egonode-c egonode-d  12 4
allocated: 31
addResource Call Back
AllocId=31 Consumer= Hosts=1
Name=egonode-c Slots=1 Attributes: cpuf=8.30            type=LINUX86
Allocated host: egonode-c
Started container 377
sample2_client Sample2 55350@172.25.244.92
sample7_client Sample7 Client 53182@172.25.244.92
moviestore MovieStoreController 55358@172.25.244.143
EGO_SERVICE_CONTROLLER url=archtsmst.noam.corp.platform.com:7872 47408@172.25.244.143
containerStateChg Call Back
377 1
containerStateChg Call Back
377 4
```

Tutorial 4: Request Resource Allocation in a Cluster and Start Containers Using Threads

This tutorial describes how to create a registered EGO client that requests resource allocation in a cluster and starts containers on the hosts. This sample program uses threads to make the allocation requests to Platform EGO and start containers on the hosts allocated by Platform EGO.

Using this tutorial, you will …

- Open a connection to Platform EGO
- Print out cluster information
- Check if there are any registered clients connected to Platform EGO
- Log on to Platform EGO
- Initialize a structure for client callback methods
- Register the client with Platform EGO
- Print out allocation and container reply info from a previous connection
- Print out host group information
- Define and initialize structures for the work, resource, and monitor threads
- Create and run the three threads
- Determine the number of available host slots and make a resource allocation request for half of them
- Store allocation requests in a resource queue and make allocation requests to Platform EGO
- Retrieve the allocation reply from the work queue and start a container on each host slot
- Calculate the average host load
- Check for registered clients connected to Platform EGO and print out info
- Unregister the client

Underlying principles

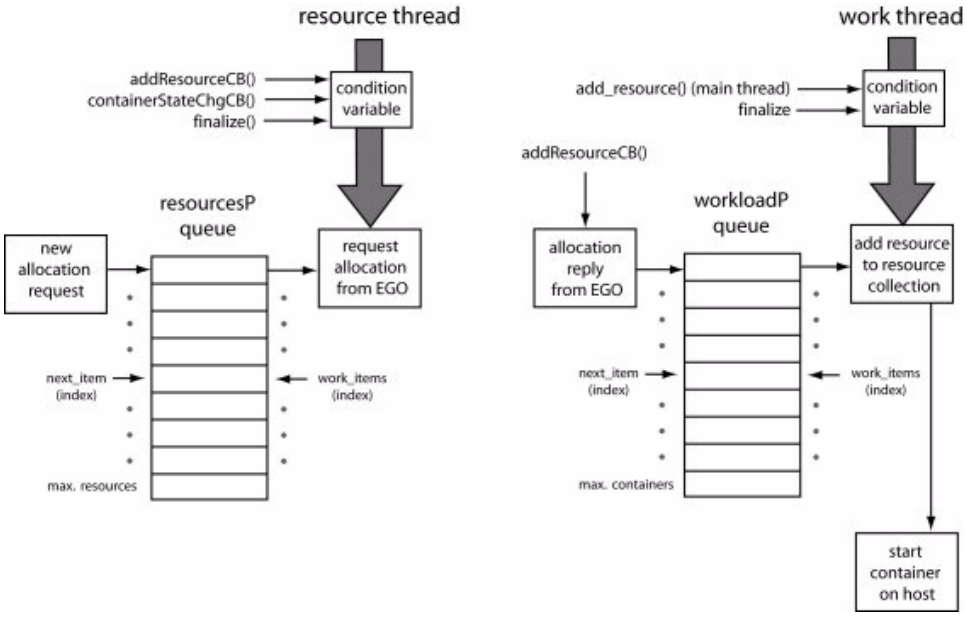
This code sample uses global data structures (workloadP, resourcesP, and monitorP) that are accessible by different threads. Since these data structures are considered shared resources, a mutex (mutual exclusion) object is used to prevent simultaneous modification of the data. The mutex object can be locked and

unlocked by individual threads, thereby controlling access to the respective data structure. In conjunction with the mutex object, a condition variable enables threads to wait for the data to enter a defined state before accessing the data.

This sample implements three threads: resource, work, and monitor, in addition to the main thread.

The resource thread is responsible for getting the allocation request from the resource queue and making an allocation request to Platform EGO. Once the resource thread enters the wait state, the addResourceCB() and containerStateChgCB() callback methods and finalize() method set the condition variable that enables the resource thread to resume execution. The resource thread cycles through the queue until all allocation requests have been processed.

The work thread is responsible for getting the allocation reply from the work queue, adding the resource to the resource collection structure, and starting a container on the allocated host slot. Once the work thread enters the wait state, the add_resources() (called from the main thread) and finalize() methods set the condition variable that enables the work thread to resume execution. The thread cycles through the queue until containers have been started on all allocated host slots.



Step 1: Preprocessor directives and global variable declarations

The first step is to include a reference to the system and API header files, followed by the declaration of global variables and structures that are implemented in the sample.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
#include <errno.h>
#include "vem.api.h"
#include "samples.h"

int samples_shutdown = 0;
work_state_t *workloadP;
resource_state_t *resourcesP;
monitor_state_t *monitorP;
```

Step 2: Implement the principal method

Lines 4-7: define and initialize a data structure that is used to request a connection with the EGO host cluster. The data structure contains a reference to a configuration file where the master host name and port numbers are stored.

Line 8: pass the data structure as an argument to the vem_open () method, which opens a connection to the master host. If the connection attempt is successful, a handle is returned; otherwise the method returns NULL. The handle acts as a communication channel to the master host and all subsequent communication occurs through this handle.

Lines 15-16: the vem_name_t structure (defined as clusterName) is initialized with NULL. This structure holds the cluster name, system name, and version. The vem_uname () method is passed the communication handle and, if successful, returns a valid vem_name_t structure ; otherwise the method returns NULL.

Line 24: the cluster info is printed out to the screen.

Lines 26-43: define the client info structure. Use vem_locate() to get all registered clients. Since NULL is provided as the client name, all registered clients will be located and the method returns the number of registered clients. Note that Platform

EGO is equipped with a number of default clients (services) such as the Service Controller, so as a minimum, the info relevant to these clients is printed out and the associated memory is released.

```
1   int
2   sample4()
3   {
4     vem_openreq_t orequest;
5     vem_handle_t *vhandle = NULL;
6     orequest.file = "ego.conf";
7     orequest.flags=0;
8   vhandle = vem_open(&orequest);
9     if (vhandle == NULL) {
10    // error opening
11     fprintf(stderr, "Error opening cluster: %s\n",  vem_strerror(vemerrno));
12     return -1;
13    }
14
15    vem_name_t *clusterName = NULL;
16    clusterName = vem_uname(vhandle);
17    if (clusterName == NULL) {
18    // error connecting
19     fprintf(stderr, "Error connecting to cluster: %s\n",
20   vem_strerror(vemerrno));
21     return -2;
22    }
23
24    fprintf(stdout, " Connected... %s %s %4.2f\n", clusterName->clustername,
25   clusterName->sysname, clusterName->version);
26  vem_clientinfo_t *clients;
27    int  rc = vem_locate(vhandle, NULL, &clients);
28    if (rc >=0) {
29      if (rc == 0) {
30       printf("No registered clients exist\n");
31      } else {
32      int i=0;
33      for (i=0; i<rc; i++) {
34         printf("%s %s %s\n", clients[i].name, clients[i].description,
35         clients[i].location);
36      }
37      // free
38      vem_clear_clientinfo(clients);
39      }
40    } else {
41    // error connecting
42     fprintf(stderr, "Error geting clients: %s\n",  vem_strerror(vemerrno));
43    }
```

Lines 44-47: authenticate the user to Platform EGO.

Lines 48-52: define and initialize a structure for callback methods. These callback methods are invoked by Platform EGO when resources are added or reclaimed, or when a change occurs to host status or a container. When Platform EGO wants to communicate about these events, it invokes these methods thereby calling back to the client.

```
44   if (login(vhandle, username, password)<0) {
45    fprintf(stderr, "Error logon: %s\n",  vem_strerror(vemerrno));
46    goto leave;
47 }
48   vem_clientcallback_t cbf;
49   cbf.addResource = addResourceCB;
50   cbf.reclaimForce = reclaimForceCB;
51   cbf.containerStateChg = containerStateChgCB;
52   cbf.hostStateChange = hostStateChangeCB;
```

Lines 53-67: define the vem_allocation_info_reply_t and vem_container_info_reply_t structures. If a client gets disconnected and then re-registers, its existing allocations and containers are returned to these structures. If the client had never registered before, the structures would be empty. Define and initialize a structure (rreq) that holds client info for registration purposes. (This includes assigning the client callback structure (cbf) to the callback member of the rreq structure.) Register with Platform EGO via the open connection using vem_register().

```
53 vem_allocation_info_reply_t aireply;
54   vem_container_info_reply_t  cireply;
55   vem_registerreq_t rreq;
56
57   rreq.name = "sample4_client";
58   rreq.description = "Sample4 Client";
59   rreq.flags = VEM_REGISTER_TTL;
60   rreq.ttl = 3;
61   rreq.cb = &cbf;
62
63   rc = vem_register(vhandle, &rreq, &aireply, &cireply);
64   if (rc < 0) {
65     fprintf(stderr, "Error registering: %s\n",  vem_strerror(vemerrno));
66    goto leave;
67   }
```

Lines 68-71: print out information related to the allocation requests and containers. Once the info is printed out, the memory for the allocations is freed.

Lines 75-82: the vem_gethostgroupinfo() method collects the information for the requested hostgroup. In this case, the requested hostgroup in the input argument is set to NULL, which means that information about all hostgroups is requested. If the method call is successful, hostgroup information is printed out to the screen.

```
68   print_vem_allocation_info_reply(&aireply);
69   print_vem_container_info_reply(&cireply);
70   // freeup any previous allocations
71   release_vem_allocation(vhandle, &aireply);
72 vem_hostgroupreq_t hgroupreq;
73   hgroupreq.grouplist = NULL;
74   vem_hostgroup_t *hgroup;
75   rc = vem_gethostgroupinfo(vhandle, &hgroupreq, &hgroup);
76   if (rc < 0) {
77     fprintf(stderr, "Error getting hostgroup: %s\n",
78  vem_strerror(vemerrno));
79   } else {
80   printf("%s %s %d %d\n", hgroup->groupName, hgroup->members, hgroup->free,
81  hgroup->allocated);
82   }
```

Lines 83-95: define and initialize structures for the workload, resources and monitor threads. These structures are global in scope.

Lines 96-105: create and run the three threads.

Line 107-115: get half the number of available host slots and make a corresponding number of resource allocation requests via the add_resources() method. This method adds a new allocation request to the resource queue and increments the queue index (next_item). The add_resources() method also sets the condition variable, which tells the waiting resource_thread that a new allocation request has been added to the resource queue. The resource_thread resumes execution and the resource_mutex object is unlocked.

When a resource is added, the addResourceCB() callback method is executed. The callback method adds the allocation reply structure to the workload queue at position next_item and increments the index (next_item). The condition variable

is set, which tells the waiting work_thread that a new allocation reply has been added to the workload queue. The work_thread resumes execution and the work_mutex object is unlocked. The allocation reply is also printed out.

```
83  pthread_t worker_thread, resource_thread, monitor_thread;
84   work_state_t workload;
85   resource_state_t resources;
86   monitor_state_t monitor;
87
88   // globals so that callback functions can find the queues/lock/cond var
89   workloadP = &workload;
90   resourcesP = &resources;
91   monitorP = &monitor;
92
93   initialize_workload(&workload, vhandle);
94   initialize_resources(&resources, vhandle);
95   initialize_monitor(&monitor, vhandle);
96 if (pthread_create(&worker_thread, NULL, work_thread_fn, &workload)) {
97     perror("Error creating worker thread: ");
98   }
99 if (pthread_create(&resource_thread, NULL, resource_thread_fn,
100 &resources)) {
101    perror("Error creating resource thread: ");
102  }
103 if (pthread_create(&monitor_thread, NULL, monitor_thread_fn, &monitor)) {
104    perror("Error creating monitor thread: ");
105  }
106 // Request half of them, one if just one is available
107  int numavailable = getNumberOfHostSlotsAvailable(vhandle);
108  fprintf(stderr, "Available Slots=%d\n", numavailable);
109  if(numavailable > 0) {
110  int num_request = (numavailable / 2) > 1 ? (numavailable / 2): 1; //3;
111    vem_allocreq_t *aloc_spec = get_alloc_spec();
112    // aloc_spec->maxslots = 1;
113 // add to request Q
114    add_resources(num_request, aloc_spec);
115  }
```

Lines 116-117: pause the main thread for 180 milliseconds. The finalize() method sets the samples_shutdown flag to 1 and sets the condition variable for all three threads. The shutdown flag causes the three threads to end execution.

Lines 119-126: block the main thread until all three threads have finished. Clean up the thread states by destroying the mutex object and condition variable associated with each thread.

Lines 128-143: use vem_locate() to get all registered clients. Since NULL is provided as the client name, all registered clients will be located and the method returns the number of registered clients. Note that Platform EGO is equipped with

a number of default clients (services) such as the Service Controller, so as a minimum, the info relevant to these clients is printed out and the associated memory is released.

```
116 sleep(180);
117 finalize();
118 // wait for worker, resource, monitor threads to finish
119 pthread_join(worker_thread, NULL);
120   pthread_join(resource_thread, NULL);
121   pthread_join(monitor_thread, NULL);
122
123   // clean up thread states
124   finalize_workload(workloadP);
125   finalize_resources(resourcesP);
126   finalize_monitor(monitorP);
127
128   rc = vem_locate(vhandle, NULL, &clients);
129   if (rc >=0) {
130     if (rc == 0) {
131      printf("No registered clients exist\n");
132     } else {
133     int i=0;
134     for (i=0; i<rc; i++) {
135        printf("%s %s %s\n", clients[i].name, clients[i].description,
136        clients[i].location);
137     }
138       vem_clear_clientinfo(clients);
139     }
140   } else {
141   // error connecting
142    fprintf(stderr, "Error geting clients: %s\n",  vem_strerror(vemerrno));
143   }
144 bailout:
145   rc = vem_unregister(vhandle);
146   if (rc < 0) {
147     fprintf(stderr, "Error unregistering: %s\n",  vem_strerror(vemerrno));
148   }
149   if (logout(vhandle)<0) {
150    fprintf(stderr, "Error logoff: %s\n",  vem_strerror(vemerrno));
151   }
152
153 leave:
154   // free memory
155   vem_free_uname(clusterName);
156   vem_close(vhandle);
157
158   return 0;
159 }
```

Step 3: Make resource allocation requests to Platform EGO (resource thread)

Lock the resource_mutex and wait for the condition variable to be set. Once the condition variable is set by the addResourceCB() callback method and thread execution resumes, get the allocation request from the resource queue using the

work_items index. Increment the index. Make an allocation request to Platform EGO and retrieve and store the allocation ID. Continue this cycle until all the allocation requests in the resource queue have been processed.

```
void *resource_thread_fn(resource_state_t *resourcesP)
{
while(!samples_shutdown) {
     pthread_mutex_lock(&resourcesP->resource_mutex);
     resourcesP->ready = 1;
     pthread_cond_wait(&resourcesP->resource_cond,
 &resourcesP->resource_mutex);
     fprintf(stderr, "Need to allocate?\n");
     while(resourcesP->work_items < resourcesP->next_item) {
    // deque alocspec
      vem_allocreq_t *aloc_spec =
 resourcesP->queue[resourcesP->work_items++].alocreq;
       vem_allocation_id_t alocid = get_resource(resourcesP->vhandle,
 aloc_spec);
       if(alocid != NULL) {
     // add to the allocated ids
     if(resourcesP->num_resources < MAX_RESOURCES) {
           resourcesP->alocids[resourcesP->num_resources++] = alocid;
      } else {
    // TODO Grow
    fprintf(stderr, "Exceeded Limit\n");
    finalize();
      }
       } else {
       fprintf(stderr, "Could not allocate\n");
       }
     } /* else {
    // no new request, maybe shutdown request
       }*/
   pthread_mutex_unlock(&resourcesP->resource_mutex);
// if done
   }
  // free alocid memory
fprintf(stderr, "ResourceThread Shutdown\n");
   return NULL;
}
```

Step 4: Get resource allocation reply from Platform EGO and start containers (work thread)

Lock the work_mutex and wait for the condition variable to be set. Once the condition variable is set and thread execution resumes, retrieve the allocation reply from the work queue using the work_items index. Increment the index and retrieve the container specification.

The addto_monitor_resource() method is called for each allocated host, which adds the host name, allocation ID, and allocation state to the resource collection. The method also increments the resources num counter (monitorP->resources->num).

For each slot in each host, start a container using the allocation ID, host name, and container specification. If successful, increment the running containers counter. Unlock the work_mutex.

```
void *work_thread_fn(work_state_t *workloadP)
{

   while(!samples_shutdown /* && workloadP->num_containers_running > 0 */) {
   // wait until a resource host is allocated
    pthread_mutex_lock(&workloadP->work_mutex);
    workloadP->ready = 1;
   pthread_cond_wait(&workloadP->work_cond, &workloadP->work_mutex);

    fprintf(stderr, "Received Resource?\n");
    int rc = 0;
    // pick out from queue
    while (workloadP->work_items < workloadP->next_item) {
      vem_allocreply_t * allocReply =
 &workloadP->queue[workloadP->work_items++];
      int i=0, j=0;
      vem_container_spec_t *conspec = get_container_spec();

      for(i=0; i<allocReply->nhost; i++) {

      addto_monitor_resource(monitorP, allocReply->host[i].name,
 allocReply->allocId);

      for(j=0; j<allocReply->host[i].slots; j++) {
          rc= startContainer(workloadP->vhandle, allocReply->allocId,
 allocReply->host[i].name, conspec);
          // if successful
          if (!rc) {
            ++workloadP->num_containers_running;
          }
      }
      }
     } /* else {
     // probably woken up as a container has changed state
    fprintf(stderr, "Nothing to do\n");
     } */
   pthread_mutex_unlock(&workloadP->work_mutex);
   }
   fprintf(stderr, "WorkThread Shutdown\n");
   return NULL;
}
```

Step 5: Calculate the average host load (monitor thread)

Lock the monitor_mutex and get the current time. The thread now waits for either a host state change to be signalled by Platform EGO or the wait time to expire. If a state change occurs, the corresponding callback method (hostStateChangeCB) is invoked by Platform EGO, which updates the host state in the resource collection.

The condition variable is then set to reactivate the monitor thread. If the thread resumes execution due to wait time expiration, the average computer load is calculated and printed out.

```
void *monitor_thread_fn(monitor_state_t *monitorP)
{
   struct timespec timeout;
   struct timeval  now;
   int rc;

   while(!samples_shutdown) {
   // wait until change in host/container status is received
   pthread_mutex_lock(&monitorP->monitor_mutex);
    monitorP->ready = 1;
gettimeofday(&now);
    timeout.tv_sec = now.tv_sec + 30;
    timeout.tv_nsec = now.tv_usec *1000;
rc = pthread_cond_timedwait(&monitorP->monitor_cond,
 &monitorP->monitor_mutex, &timeout);

    // Currently no way to get container from id.
    //print_vem_container(vem_container_t *container);
    if(rc == ETIMEDOUT) {
      vem_hostinfo_t *hinfo = NULL;
      char **attrs;
      int numh;
      double *loads;
     double load = computeAverageLoad(monitorP, &numh, &hinfo, &attrs, &loads);
     fprintf(stderr, "\nMonitor: Avg. Load =%6.2f\n", load);
     if(hinfo != NULL) {
     free(loads);
       vem_free_hostinfo(hinfo, numh, attrs);
     }
    } else {
    // we were signaled
      fprintf(stderr, "Received Event?\n");
    }
   pthread_mutex_unlock(&monitorP->monitor_mutex);

   // update activity information
   }
   fprintf(stderr, "MonitorThread Shutdown\n");
   return NULL;
}
```

Step 6: Client callback methods

These callback methods are invoked by Platform EGO when resources are added or reclaimed, or when a change occurs to host status or a container. When Platform EGO wants to communicate about these events, it invokes these methods thereby calling back to the client.

The addResourceCB() method locks the work_mutex object. The method then adds the allocation reply structure to the workload queue at position next_item and increments the index (next_item). The condition variable is set, which tells the waiting work_thread that a new allocation reply has been added to the workload queue. The work_mutex object is unlocked and the allocation reply is printed out.

```
int
addResourceCB(vem_allocreply_t *areply)
{
printf("addResource Call Back\n");

    pthread_mutex_lock(&workloadP->work_mutex);
    // check if thread is ready?
    while(!workloadP->ready) {
      pthread_mutex_unlock(&workloadP->work_mutex);
      sleep(1);
      pthread_mutex_lock(&workloadP->work_mutex);
    }

    // add to queue
    if(workloadP->next_item < MAX_CONTAINERS) {
      vem_allocreply_clone_deep(&workloadP->queue[workloadP->next_item++],
 areply);
    } else {
    //TODO increase capacity
    fprintf(stderr, "Exceeded limit");
    finalize();
    }
    pthread_cond_signal(&workloadP->work_cond);
    pthread_mutex_unlock(&workloadP->work_mutex);

    print_vem_allocreply(areply);
    return 0;
}
```

The containerStateChgCB() method prints out the host name and the new host state.

The work_mutex is locked and the method cycles through the list of containers and finds the container ID associated with the state change. If the new state indicates that the container has finished running, the workloadP->container_state is updated

with the new state and the total number of running containers is decremented. The condition variable is set to signal the work_thread to resume execution and the work_mutex object is unlocked.

```
int
containerStateChgCB(vem_containerstatechg_t *cschange)
{
printf("containerStateChg Call Back\n");
printf("%s %d\n", cschange->containerId, cschange->newState);

pthread_mutex_lock(&workloadP->work_mutex);
update_container_state(workloadP, cschange);
    pthread_cond_signal(&workloadP->work_cond);
    pthread_mutex_unlock(&workloadP->work_mutex);

    return 0;
}
```

The hostStateChangeCB() method prints out the host name and the new host state. After locking the monitor_mutex, the method calls the update_host_state() method which, in turn, updates the resource collection with the new host state info. The condition variable is set, which signals to the waiting monitor_thread to resume execution.

```
int
hostStateChangeCB(vem_hoststatechange_t *hschange)
{
printf("hostStateChange Call Back\n");
printf("%s %d\n", hschange->name, hschange->newState);
pthread_mutex_lock(&monitorP->monitor_mutex);
update_host_state(monitorP, hschange);
    pthread_cond_signal(&monitorP->monitor_cond);
    pthread_mutex_unlock(&monitorP->monitor_mutex);

    return 0;
}
```

Step 7: Calculate the average activity load on the resources

In order to determine if a resource is too busy to receive jobs, a load index value is calculated and compared to a corresponding load threshold parameter. The following code retrieves and processes the load index value for r1m, the 1-minute CPU run queue length.

The method cycles through each resource in the resource collection and checks the allocation status. If the resource is allocated, a counter is incremented that keeps track of the number of allocated resources. The resource name is also stored (hostlist array), which is assigned to the hinforeq structure. Since the host list is used as an input to vem_getHostInfo(), the method will only update info for hosts in the list, as well as return the number of allocated hosts.

The getLoadAttribute() method cycles through the list of host attributes looking for the r1m load index. (The r1m load index represents the average number of processes ready to use the CPU during a one-minute interval.) This method returns

the attribute array index corresponding to load index r1m. The value for r1m is retrieved for each allocated resource and converted to a double data type. These values are stored in an array as well as added to a variable (load), which is used as an accumulator. The total sum is then divided by the number of allocated resources to yield the average load index value of r1m.

```
double computeAverageLoad(monitor_state_t *monitorP, int *nhostsP, vem_hostinfo_t
**hinfoP, char*** host_attributesP, double **loadsP)
{
  double load = 0.0;
  double *loads = NULL;

  // assumes caller has acquired the lock
  // finds allocated hosts
  resource_collection_t *resources = monitorP->resources;
  int i, j=0, count=0;
  for(i=0; i<resources->num; i++){
  if(resources->allocstate[i] == RESOURCE_ALLOCATED) {
    count++;
  }
  }
/* No hosts allocated */
  if(count == 0) {
  *nhostsP          = 0;
    *host_attributesP = NULL;
   *hinfoP = NULL;
   *loadsP = NULL;
  return 0;
  }
char **hostlist              = calloc(count, sizeof(char*));

  for(i=0; i<resources->num; i++){
  if(resources->allocstate[i] == RESOURCE_ALLOCATED) {
    hostlist[j]     = resources->names[i];
    j++;
  }
  }
```

```
int hin;
  vem_hostinforeq_t hinforeq;
  hinforeq.resreq = "";        // resource string, unimplemented
  hinforeq.hostlist = hostlist;
  vem_hostinfo_t *hinfo = NULL; // out parameter, set by vem_getHostInfo
  char **attrs = NULL;          // out parameter, set by vem_getHostInfo
hin = vem_getHostInfo(monitorP->vhandle, &hinforeq, &hinfo, &attrs);
if (hin < 0) {
  // error
   fprintf(stderr, "Error getting hostinfo: %s %d\n",  vem_strerror(vemerrno),
 hin);
    *nhostsP    = hin;
    *host_attributesP = attrs;
   *hinfoP = NULL;
   *loadsP = loads;
   return -1.0;
  } else {
    //TODO are the hosts in hinfo, in the same order as hinforeq?
    print_hostInfo(hin, hinfo, attrs);
    // find load attribute
    int index = get_load_attribute("r1m", attrs);
    loads = calloc(hin, sizeof(double));
    double l;
    for(i=0; i<hin; i++) {
    l = get_load(&hinfo[i], index);
    loads[i] = l;
    load += l;
    }
free(hostlist);
  }
  *nhostsP    = hin;
  *hinfoP     = hinfo;
  *host_attributesP = attrs;
  *loadsP = loads;
  return (count == 0) ? 0 : load / count;
}
```

```
double get_load(vem_hostinfo_t *hinfo, int index)
{
double rv=-1.0;
    vem_value_t value;
    value = hinfo->attributes[index];
    rv = value.value.v_float32;
    return rv;
}

int get_load_attribute(char *load, char **attrs)
{
  int j=-1;
  while(attrs[++j] != NULL) {
  if(!strcmp(load, attrs[j])) {
  return j;
  }
  }
  return j;
}
```

Run the client application

1. Select **Run > Run**.

   The Run dialog appears.
2. In the Configurations list, either select an EGO C Client Application or click **New** for a new configuration.

   For a new configuration, enter the configuration name.
3. Enter the project name and C/C++ Application name.
4. Click **Apply** and then **Run**.

Sample output

```
 Connected... demo2 archtsmst.noam.corp.platform.com 1.00
sample7_client Sample7 Client 53182@172.25.244.92
moviestore MovieStoreController 55358@172.25.244.143
EGO_SERVICE_CONTROLLER url=archtsmst.noam.corp.platform.com:7872 47408@172.25.244.143
Container
Id=377
State= 4, finish
Name=Sample2Container
AllocId=31
Consumer=StoreFront Start=1147808029, End=1147808029
Host=egonode-c ExitStatus=0 ExitReason=  6, user account doesn't exist
Compute egonode-a egonode-b egonode-c egonode-d  12 4
Attribute       egonode-a       egonode-b       archtsmst.noam.corp.platform.com        egonode-c
Status          Ok              Ok              Ok              Ok              Ok
status          0               0               0               0               0
type            LINUX86         LINUX86         LINUX86         LINUX86         LINUX86
model           PC300           PC300           PC450           PC300           PC300
ncpu            1               1               1               1               1
cpuf            8.30            8.30            13.20           8.30            8.30
mem             156.00          160.00          66.00           163.00          161.00
swp             544.50          544.50          781.17          717.00          552.50
maxmem          250             250             377             250             250
maxswp          556             556             823             729             564
tmp             9984.00         10032.00        5606.81         15968.00        10112.00
ut              0.00            0.00            0.01            0.00            0.00
it              6036.00         6032.00         5964.13         6036.00         6036.00
io              17.08           17.95           780.66          31.56           39.38
pg              1.14            1.20            52.41           2.09            2.65
```

```
allocated: 33

allocated: 34

allocated: 35

allocated: 36

allocated: 37

allocated: 38
addResource Call Back
AllocId=32 Consumer= Hosts=1
Name=egonode-c Slots=1 Attributes: cpuf=8.30            type=LINUX86
addResource Call Back
AllocId=33 Consumer= Hosts=1
Name=egonode-d Slots=1 Attributes: cpuf=8.30            type=LINUX86
Received Resource?
Allocated host: egonode-c
containerStateChg Call Back
378 1
Started container 378
Allocated host: egonode-d
Started container 379
containerStateChg Call Back
379 1
containerStateChg Call Back
378 4
```

```
378 4
containerStateChg Call Back
379 4
Attribute       egonode-c       egonode-d
Status          Ok              Ok
status          0               0
type            LINUX86         LINUX86
model           PC300           PC300
ncpu            1               1
cpuf            8.30            8.30
mem             163.00          161.00
swp             717.00          552.50
maxmem          250             250
maxswp          729             564
tmp             15968.00        10112.00
ut              0.00            0.00
it              6036.00         6036.00
io              18.42           24.39
pg              1.22            1.64
r1m             0.00            0.00
r15s            0.00            0.00
r15m            0.00            0.00
ls              0               0
slot            4               4
freeslot        3               3
RESOURCES       (linux)         (linux)

Monitor: Avg. Load =  0.00
```

```
tmp             15968.00        10112.00
ut              0.00            0.00
it              6040.00         6040.00
io              27.30           23.28
pg              1.82            1.55
r1m             0.00            0.00
r15s            0.00            0.00
r15m            0.00            0.00
ls              0               0
slot            4               4
freeslot        3               3
RESOURCES       (linux)         (linux)

Monitor: Avg. Load =  0.00
containerStateChg Call Back
377 0
Received Resource?
Received Resource?
WorkThread Shutdown
Need to allocate?
ResourceThread Shutdown
Received Event?
MonitorThread Shutdown
sample4_client Sample4 Client 55439@172.25.244.92
sample7_client Sample7 Client 53182@172.25.244.92
moviestore MovieStoreController 55358@172.25.244.143
EGO_SERVICE_CONTROLLER url=archtsmst.noam.corp.platform.com:7872 47408@172.25.244.143
```

Tutorial 5: Request Resource Allocation in a Cluster and Start Containers Based on Host Loading

This tutorial describes how to create a registered EGO client that requests resource allocation in a cluster and starts containers on the hosts. Based on host loading, the program either adds more resources or releases some.

Using this tutorial, you will …

- Open a connection to Platform EGO
- Print out cluster information
- Check if there are any registered clients connected to Platform EGO
- Log on to Platform EGO
- Register the client with Platform EGO
- Print out allocation and container reply information from a previous connection
- Print out host group information
- Determine the number of available host slots and make a resource allocation request for half of them
- Store allocation requests in a resource queue and make allocation requests to Platform EGO
- Retrieve the allocation reply from the work queue and start a container on each host slot
- Add or release a resource depending on host loading
- Check for registered clients connected to Platform EGO and print out info
- Unregister the client

Underlying principles

Refer to Tutorial 4: Underlying principles on page 59 for a description of the multi-thread technique used in this sample.

Step 1: Preprocessor directives

The first step is to include a reference to the system and API header files. The samples.h header file contains the method declarations that are common to all of the samples.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
#include <errno.h>
#include <sys/time.h>
#include "vem.api.h"
#include "samples.h"
```

Step 2: Implement the principal method

Lines 4-7: define and initialize a data structure that is used to request a connection with the EGO host cluster. The data structure contains a reference to a configuration file where the master host name and port numbers are stored.

Line 8: pass the data structure as an argument to the vem_open () method, which opens a connection to the master host. If the connection attempt is successful, a handle is returned; otherwise the method returns NULL. The handle acts as a communication channel to the master host and all subsequent communication occurs through this handle.

Lines 16-17: the vem_name_t structure (defined as clusterName) is initialized with NULL. This structure holds the cluster name, system name, and version. The vem_uname () method is passed the communication handle and, if successful, returns a valid vem_name_t structure ; otherwise the method returns NULL

Line 24: the cluster info is printed out to the screen.

Lines 27-43: locate all the registered clients and print out the client info (name, description, and location). Define the client info structure. Use vem_locate() to get all registered clients. Since NULL is provided as the client name, all registered clients will be located and the method returns the number of registered clients. Note

that Platform EGO is equipped with a number of default clients (services) such as the Service Controller, so as a minimum, the info relevant to these clients is printed out and the associated memory is released.

```
1  int
2  sample5()
3  {
4    vem_openreq_t orequest;
5    vem_handle_t *vhandle = NULL;
6    orequest.file = "ego.conf";
7    orequest.flags=0;
8    vhandle = vem_open(&orequest);
9
10   if (vhandle == NULL) {
11   // error opening
12    fprintf(stderr, "Error opening cluster: %s\n",  vem_strerror(vemerrno));
13    return -1;
14   }
15
16   vem_name_t *clusterName = NULL;
17   clusterName = vem_uname(vhandle);
18   if (clusterName == NULL) {
19   // error connecting
20    fprintf(stderr, "Error connecting to cluster: %s\n",
21  vem_strerror(vemerrno));
22    return -2;
23   }
24 fprintf(stdout, " Connected... %s %s %4.2f\n", clusterName->clustername,
25  clusterName->sysname, clusterName->version);
26 vem_clientinfo_t *clients;
27   int  rc = vem_locate(vhandle, NULL, &clients);
28   if (rc >=0) {
29     if (rc == 0) {
30      printf("No registered clients exist\n");
31     } else {
32     int i=0;
33     for (i=0; i<rc; i++) {
34        printf("%s %s %s\n", clients[i].name, clients[i].description,
35        clients[i].location);
36     }
37     // free
38     vem_clear_clientinfo(clients);
39     }
40   } else {
41   // error connecting
42    fprintf(stderr, "Error geting clients: %s\n",  vem_strerror(vemerrno));
43   }
```

Lines 44-47: authenticate the user to Platform EGO.

Lines 48-52: define and initialize a structure for callback methods. These callback methods are invoked by Platform EGO when resources are added or reclaimed, or when a change occurs to host status or a container. When Platform EGO wants to communicate about these events, it invokes these methods thereby calling back to the client.

Lines 53-67: define the vem_allocation_info_reply_t and vem_container_info_reply_t structures. If a client gets disconnected and then re-registers, its existing allocations and containers are returned to these structures. If the client had never registered before, the structures would be empty. Define and initialize a structure (rreq) that holds client info for registration purposes. (This includes assigning the client callback structure (cbf) to the callback member of the rreq structure.) Register with Platform EGO via the open connection using vem_register().

Lines 68-71: print out information related to the allocation requests and containers. Once the info is printed out, the memory for the allocations is freed.

Lines 75-82: the vem_gethostgroupinfo() method collects the information for the requested hostgroup. In this case, the requested hostgroup in the input argument is set to NULL, which means that information about all hostgroups is requested. If the method call is successful, hostgroup information is printed out to the screen.

```
44   if (login(vhandle, username, password)<0) {
45    fprintf(stderr, "Error logon: %s\n",  vem_strerror(vemerrno));
46    goto leave;
47   }
48 vem_clientcallback_t cbf;
49   cbf.addResource = addResourceCB;
50   cbf.reclaimForce = reclaimForceCB;
51   cbf.containerStateChg = containerStateChgCB;
52   cbf.hostStateChange = hostStateChangeCB;
53 vem_allocation_info_reply_t aireply;
54   vem_container_info_reply_t  cireply;
55   vem_registerreq_t rreq;
56
57   rreq.name = "sample5_client";
58   rreq.description = "Sample5 Client";
59   rreq.flags = VEM_REGISTER_TTL;
60   rreq.ttl = 3;
61   rreq.cb = &cbf;
62
63   rc = vem_register(vhandle, &rreq, &aireply, &cireply);
64   if (rc < 0) {
65     fprintf(stderr, "Error registering: %s\n",  vem_strerror(vemerrno));
66    goto leave;
67   }
68   print_vem_allocation_info_reply(&aireply);
69   print_vem_container_info_reply(&cireply);
70   // freeup any previous allocations
71   release_vem_allocation(vhandle, &aireply);
72 vem_hostgroupreq_t hgroupreq;
73   hgroupreq.grouplist = NULL;
74   vem_hostgroup_t *hgroup;
75   rc = vem_gethostgroupinfo(vhandle, &hgroupreq, &hgroup);
76   if (rc < 0) {
77     fprintf(stderr, "Error getting hostgroup: %s\n",
78  vem_strerror(vemerrno));
79   } else {
80   printf("%s %s %d %d\n", hgroup->groupName, hgroup->members, hgroup->free,
81  hgroup->allocated);
82   }
```

Lines 83-95: define and initialize structures for the workload, resources and monitor threads. These structures are global in scope.

Lines 97-107: create and run the three threads. Refer to Tutorial 4: Underlying principles on page 59 for further details of the workload and resource threads.

Lines 109-117: get half the number of available host slots and make a corresponding number of resource allocation requests via the add_resources() method. This method adds a new allocation request to the resource queue and increments the queue index (next_item). The add_resources() method also sets the condition variable, which tells the waiting resource_thread that a new allocation request has been added to the resource queue. The resource_thread resumes execution and the resource_mutex object is unlocked.

```
83 pthread_t worker_thread, resource_thread, monitor_thread;
84   work_state_t workload;
85   resource_state_t resources;
86   monitor_state_t monitor;
87
88   // globals so that callback functions can find the queues/lock/cond var
89   workloadP = &workload;
90   resourcesP = &resources;
91   monitorP = &monitor;
92
93   initialize_workload(&workload, vhandle);
94   initialize_resources(&resources, vhandle);
95   initialize_monitor(&monitor, vhandle);
96
97 if (pthread_create(&worker_thread, NULL, work_thread_fn, &workload)) {
98     perror("Error creating worker thread: ");
99   }
100 if (pthread_create(&resource_thread, NULL, resource_thread_fn,
101  &resources)) {
102    perror("Error creating resource thread: ");
103  }
104 if (pthread_create(&monitor_thread, NULL, monitor_thread_extended_fn,
105  &monitor)) {
106    perror("Error creating monitor thread: ");
107  }
108 // Request half of them, one if just one is available
109  int numavailable = getNumberOfHostSlotsAvailable(vhandle);
110  fprintf(stderr, "Available Slots=%d\n", numavailable);
111  if(numavailable > 0) {
112  int num_request = 4; //(numavailable / 2) > 1 ? (numavailable / 2): 1; //3;
113    vem_allocreq_t *aloc_spec = get_alloc_spec();
114    // aloc_spec->maxslots = 1;
115 // add to request Q
116    add_resources(num_request, aloc_spec);
117  }
```

When a resource is added, the addResourceCB() callback method is executed. The callback method adds the allocation reply structure to the workload queue at position next_item and increments the index (next_item). The condition variable is set, which tells the waiting work_thread that a new allocation reply has been added to the workload queue. The work_thread resumes execution and the work_mutex object is unlocked. The allocation reply is also printed out.

Lines 118-119: pause the main thread for 180 milliseconds. The finalize() method sets the shutdown flag to 1 and sets the condition variable for all three threads. The shutdown flag causes the three threads to end execution.

Lines 122-128: block the main thread until all three threads have finished. Clean up the thread states by destroying the mutex object and condition variable associated with each thread.

Lines 130-145: use vem_locate() to get all registered clients. Since NULL is provided as the client name, all registered clients will be located and the method returns the number of registered clients. If successful, print out the client info and free the associated memory.

```
118 sleep(180);
119 finalize();
120   // wait for worker, resource, monitor threads to be finish
121
122   pthread_join(worker_thread, NULL);
123   pthread_join(resource_thread, NULL);
124   pthread_join(monitor_thread, NULL);
125 // clean up thread states
126   finalize_workload(workloadP);
127   finalize_resources(resourcesP);
128   finalize_monitor(monitorP);
129
130   rc = vem_locate(vhandle, NULL, &clients);
131   if (rc >=0) {
132     if (rc == 0) {
133      printf("No registered clients exist\n");
134     } else {
135     int i=0;
136     for (i=0; i<rc; i++) {
137        printf("%s %s %s\n", clients[i].name, clients[i].description,
138        clients[i].location);
139     }
140       vem_clear_clientinfo(clients);
141     }
142   } else {
143   // error connecting
144    fprintf(stderr, "Error geting clients: %s\n",  vem_strerror(vemerrno));
145   }
146 bailout:
147   rc = vem_unregister(vhandle);
148   if (rc < 0) {
149     fprintf(stderr, "Error unregistering: %s\n",  vem_strerror(vemerrno));
150   }
151 if (logout(vhandle)<0) {
152    fprintf(stderr, "Error logoff: %s\n",  vem_strerror(vemerrno));
153   }
154 leave:
155   // free memory
156   vem_free_uname(clusterName);
157   vem_close(vhandle);
158 return 0;
159 }
```

Step 3: Add or release resources based on average host load (monitor thread)

Lock the monitor_mutex and get the current time. The thread now waits for either a host state change to be signalled by Platform EGO or the wait time to expire. If a state change occurs, the corresponding callback method (hostStateChangeCB) is invoked by Platform EGO, which updates the host state in the resource collection. The condition variable is then set to reactivate the monitor thread. If the thread resumes execution due to wait time expiration, the average computer load is calculated and printed out; refer to Tutorial 4: Step 7: Calculate the average activity load on the resources on page 71

If the average host load is greater than a predetermined threshold, an additional resource is allocated via the add_resources() method; refer to Tutorial 4: Step 6: Client callback methods on page 69 for further details of the add_resources() method. If the average computer load is less than a predetermined threshold and the number of allocated hosts is greater than 0, release a resource via the release_resources() method; refer to the next step.

```
void *monitor_thread_extended_fn(monitor_state_t *monitorP)
{
   struct timespec timeout;
   struct timeval  now;
   int rc;

   while(!samples_shutdown) {
   // wait until change in host/container status is received
   pthread_mutex_lock(&monitorP->monitor_mutex);
    monitorP->ready = 1;

    gettimeofday(&now, NULL);
    timeout.tv_sec = now.tv_sec + 30;
    timeout.tv_nsec = now.tv_usec *1000;

   rc = pthread_cond_timedwait(&monitorP->monitor_cond, &monitorP->monitor_mutex,
&timeout);

    // Currently no way to get container from id.
    //print_vem_container(vem_container_t *container);
    if (rc == ETIMEDOUT) {
      char **attrs = NULL;
      int numh = -1;
      vem_hostinfo_t *hinfo = NULL;
      double *loads = NULL;
```

```
   double load = computeAverageLoad(monitorP, &numh, &hinfo, &attrs, &loads);
      fprintf(stderr, "\n Monitor: Avg. Load =%6.2f\n", load);

     // update resources based on load
     int delta = 1;
  vem_allocreq_t *aloc_spec = get_alloc_spec();
      if(load > UPPER_THRESHOLD) {
       add_resources(delta, aloc_spec);
     } else if (load < LOWER_THRESHOLD && numh > 0) {
       release_resources(monitorP->vhandle, delta, aloc_spec, numh, hinfo, attrs,
loads);
     }
      if(hinfo != NULL) {
        vem_free_hostinfo(hinfo, numh, attrs);
      }
     if(loads != NULL) {
       free(loads);
      }
    } else {
      fprintf(stderr, "Received Event?\n");
    }
    pthread_mutex_unlock(&monitorP->monitor_mutex);
   }
   fprintf(stderr, "MonitorThread Shutdown\n");
   return NULL;
}
```

Step 4: Release resources from Platform EGO

The release_resources() method determines the least loaded host and makes a request to Platform EGO to release this allocated resource.

Define an array of structures to hold the individual host loads and associated index. Use the qsort() method to sort the host loads in ascending order.

For each resource to be released and starting with the least loaded host, get the host name and number of slots.

Get the index of the least loaded host in the resource collection. The find_host_index() method looks for the matching host name in the resource collection, and if allocated, returns the index.

Define the release request structure with information such as the allocation ID of the host to release, as well as number of hosts/slots to release. Pass this structure to the vem_release() method, which causes Platform EGO to release the resource(s).

The remove_monitor_resource() method searches in the resource collection for the name of the host to be released. When a match is found, the host's allocation state is updated to released status, i.e., unallocated.

```
int release_resources(vem_handle_t* vhandle, int delta, vem_allocreq_t *alocreq, int
num_hosts, vem_hostinfo_t *hinfo, char **attrs, double *loads)
{
  // which ones to release? Sort the load and release the ones that are least
 used
  // create a struct of load and index to sort
  int e, index, hindex, i;
  sort_element_t *element_array = calloc(num_hosts, sizeof(sort_element_t));

  for(e=0; e<num_hosts; e++) {
    element_array[e].load  = loads[e];
    element_array[e].index = e;
  }
qsort(element_array, num_hosts, sizeof(sort_element_t), sort_fn);
vem_host_t             host;
   vem_releasereq_t       release;
   resource_collection_t *resources = monitorP->resources;
for(i=0; i<delta; i++) {
     index = element_array[i].index;
     host.name  = hinfo[index].name;
     host.slots = 1;                      // should be made generic
     hindex = find_host_index(resources, hinfo[index].name);
     release.allocId = resources->allocId[hindex];
     release.nhosts  = 1;
     release.hosts   = &host;
     release.flags   = VEM_RELEASE_AUTOADJ;
int rc = vem_release(vhandle, &release);
     if (rc < 0) {
     fprintf(stderr, "Error releasing resource: %s\n",
 vem_strerror(vemerrno));
     }

     //mark them as released in the monitor host list
     if(rc == 0) {
      remove_monitor_resource(monitorP, host.name);
    }
   }
free(element_array);
   return 0;
}
```

```
int sort_fn(const void *e1, const void *e2)
{
  sort_element_t *d1 = (sort_element_t *)e1;
  sort_element_t *d2 = (sort_element_t *)e2;
  return (d1->load == d2->load) ? 0 : ( (d1->load < d2->load) ? -1 : +1);
}
```

```
int find_host_index(resource_collection_t *resources, char *hname)
{
int i, index=-1;

if (hname == NULL) return index;

for(i=0; i<resources->num; i++) {
if(!strcmp(hname, resources->names[i]) && (resources->allocstate[i] ==
 RESOURCE_ALLOCATED)) {
return i;
}
}
  return index;
}
```

```
void remove_monitor_resource(monitor_state_t *monitorP, char *hostname)
{
fprintf(stderr, "remove host\n");
// assumes the caller acquired the lock
int i;
resource_collection_t *resources = monitorP->resources;
if(hostname == NULL) {
return;
}

for(i=0; i< resources->num; i++){
      if(!strcmp(hostname, resources->names[i]) && (resources->allocstate[i]
 != RESOURCE_RELEASED)) {
        resources->allocstate[i] = RESOURCE_RELEASED;
  }
    }
}
```

Run the client application

1. Select **Run > Run**.

   The Run dialog appears.
2. In the Configurations list, either select an EGO C Client Application or click **New** for a new configuration.

   For a new configuration, enter the configuration name.
3. Enter the project name and C/C++ Application name.
4. Click **Apply** and then **Run**.

Sample output

```
 Connected... demo2 archtsmst.noam.corp.platform.com 1.00
sample7_client Sample7 Client 53182@172.25.244.92
moviestore MovieStoreController 55358@172.25.244.143
EGO_SERVICE_CONTROLLER url=archtsmst.noam.corp.platform.com:7872 47408@172.25.244.143
Compute egonode-a egonode-b egonode-c egonode-d  12 4
Attribute       egonode-a       egonode-b       archtsmst.noam.corp.platform.com        egonode-c
Status          Ok              Ok              Ok              Ok              Ok
status          0               0               0               0               0
type            LINUX86         LINUX86         LINUX86         LINUX86         LINUX86
model           PC300           PC300           PC450           PC300           PC300
ncpu            1               1               1               1               1
cpuf            8.30            8.30            13.20           8.30            8.30
mem             156.00          160.00          65.00           163.00          161.00
swp             544.50          544.50          781.17          717.00          552.50
maxmem          250             250             377             250             250
maxswp          556             556             823             729             564
tmp             9984.00         10032.00        5606.82         15968.00        10112.00
ut              0.00            0.00            0.02            0.00            0.00
it              6044.00         6040.00         5971.15         6044.00         6044.00
io              11.84           14.15           702.24          14.96           16.16
pg              0.79            0.94            46.85           1.00            1.07
r1m             0.00            0.00            1.14            0.00            0.00
r15s            0.00            0.00            0.01            0.00            0.00
r15m            0.00            0.00            0.09            0.00            0.00
ls              1               1               1               0               0
slot            4               4               8               4               4
freeslot        2               2               3               4               4
RESOURCES       (linux)         (linux)         (linux)         (linux)         (linux)
```

```
Available Slots=15
Need to allocate?
2               2               3               4               4
allocated: 39

allocated: 40

allocated: 41

allocated: 42
addResource Call Back
AllocId=39 Consumer= Hosts=1
Name=egonode-c Slots=1 Attributes: cpuf=8.30            type=LINUX86
addResource Call Back
AllocId=40 Consumer= Hosts=1
Name=egonode-d Slots=1 Attributes: cpuf=8.30            type=LINUX86
Received Resource?
Allocated host: egonode-c
Started container 380
Allocated host: egonode-d
Started container 381
containerStateChg Call Back
380 1
containerStateChg Call Back
381 1
containerStateChg Call Back
380 4
containerStateChg Call Back
```

```
Received Resource?
Attribute       egonode-c       egonode-d
Status          Ok              Ok
status          0               0
type            LINUX86         LINUX86
model           PC300           PC300
ncpu            1               1
cpuf            8.30            8.30
mem             163.00          161.00
swp             717.00          552.50
maxmem          250             250
maxswp          729             564
tmp             15968.00        10112.00
ut              0.00            0.00
it              6044.00         6044.00
io              14.96           16.16
pg              1.00            1.07
r1m             0.00            0.00
r15s            0.00            0.00
r15m            0.00            0.00
ls              0               0
slot            4               4
freeslot        3               3
RESOURCES       (linux)         (linux)

 Monitor: Avg. Load =  0.00
remove host
```

```
containerStateChg Call Back
382 1
Started container 382
containerStateChg Call Back
382 4
Attribute       egonode-c       egonode-d
Status          Ok              Ok
status          0               0
type            LINUX86         LINUX86
model           PC300           PC300
ncpu            1               1
cpuf            8.30            8.30
mem             163.00          161.00
swp             717.00          552.50
maxmem          250             250
maxswp          729             564
tmp             15968.00        10112.00
ut              0.00            0.00
it              6044.00         6044.00
io              37.19           40.78
pg              2.48            2.71
r1m             0.00            0.00
r15s            0.00            0.00
r15m            0.00            0.00
ls              0               0
slot            4               4
freeslot        3               3
RESOURCES       (linux)         (linux)
```

```
tmp             10112.00
ut              0.00
it              6044.00
io              25.78
pg              1.71
r1m             0.00
r15s            0.00
r15m            0.00
ls              0
slot            4
freeslot        3
RESOURCES       (linux)

 Monitor: Avg. Load =  0.00
remove host

 Monitor: Avg. Load =  0.00
Received Resource?
WorkThread Shutdown
Need to allocate?
ResourceThread Shutdown
Received Event?
MonitorThread Shutdown
sample5_client Sample5 Client 55649@172.25.244.92
sample7_client Sample7 Client 53182@172.25.244.92
moviestore MovieStoreController 55358@172.25.244.143
EGO_SERVICE_CONTROLLER url=archtsmst.noam.corp.platform.com:7872 47408@172.25.244.143
```

Tutorial 6: Create an EGO Service

This tutorial describes how to create and run an EGO service.

Using this tutorial, you will …

- Open a connection to Platform EGO
- Print out cluster information
- Check if there are any registered clients connected to Platform EGO
- Log on to Platform EGO
- Register the client with Platform EGO
- Print out allocation and container reply information from a previous connection
- Print out host group information
- Create and run an EGO service
- Use mutex objects to synchronize service query requests between the client (main thread) and the service thread
- Query for the IP address of the host where the service is running
- Update the Service Director with a new entry for service instance location
- Disable and remove an EGO service

Underlying principles

This sample uses two threads, main and service, and two sets of mutex objects and condition variables to synchronize the functions of the service thread. One of the tasks of the service thread is to create a service. The service thread can also be asked to query a service by a client; this only happens when the client requests it. This synchronization is achieved by using the client and service mutexes and condition variables. The service thread waits on the service condition variable, and when the client wants to query the service, it signals the service condition variable. Once the service thread obtains the information, it tells the client that the information is ready by signaling the client's condition variable.

Step 1: Preprocessor directives and declarations

The first step is to include a reference to the system and API header files, followed by the declaration of global structures that are implemented in the sample.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <pthread.h>
#include "vem.api.h"
#include "esc.api.h"
#include "samples.h"

static int addResourceCB(vem_allocreply_t *areply);
static int reclaimForceCB(vem_allocreclaim_t *areclaim);
static int containerStateChgCB(vem_containerstatechg_t *cschange);
static int hostStateChangeCB(vem_hoststatechange_t *hschange);
static char *get_service_def_xml();
service_state_t *service_stateP;
esc_service_info_reply_t *service_info_reply;
```

Step 2: Implement the principal method

Lines 4-7: define and initialize a data structure that is used to request a connection with the EGO host cluster. The data structure contains a reference to a configuration file where the master host name and port numbers are stored.

Line 8: pass the data structure as an argument to the vem_open () method, which opens a connection to the master host. If the connection attempt is successful, a handle is returned; otherwise the method returns NULL. The handle acts as a communication channel to the master host and all subsequent communication occurs through this handle.

Lines 14-15: the vem_name_t structure (defined as clusterName) is initialized with NULL. This structure holds the cluster name, system name, and version. The vem_uname () method is passed the communication handle and, if successful, returns a valid vem_name_t structure ; otherwise the method returns NULL

Line 23: the cluster info is printed out to the screen.

Lines 26-42: locate all the registered clients and print out the client info (name, description, and location). Define the client info structure. Use vem_locate() to get all registered clients. Since NULL is provided as the client name, all registered clients will be located and the method returns the number of registered clients. Note

that Platform EGO is equipped with a number of default clients (services) such as the Service Controller, so as a minimum, the info relevant to these clients is printed out and the associated memory is released.

```
1   int
2   sample6()
3   {
4     vem_openreq_t orequest;
5     vem_handle_t *vhandle = NULL;
6     orequest.file = "ego.conf";
7     orequest.flags=0;
8     vhandle = vem_open(&orequest);
9     if (vhandle == NULL) {
10    // error opening
11     fprintf(stderr, "Error opening cluster: %s\n",  vem_strerror(vemerrno));
12     return -1;
13    }
14    vem_name_t *clusterName = NULL;
15    clusterName = vem_uname(vhandle);
16    if (clusterName == NULL) {
17    // error connecting
18     fprintf(stderr, "Error connecting to cluster: %s\n",
19   vem_strerror(vemerrno));
20     return -2;
21    }
22
23    fprintf(stdout, " Connected... %s %s %4.2f\n", clusterName->clustername,
24   clusterName->sysname, clusterName->version);
25    vem_clientinfo_t *clients;
26    int  rc = vem_locate(vhandle, NULL, &clients);
27    if (rc >=0) {
28      if (rc == 0) {
29       printf("No registered clients exist\n");
30      } else {
31      int i=0;
32      for (i=0; i<rc; i++) {
33         printf("%s %s %s\n", clients[i].name, clients[i].description,
34         clients[i].location);
35      }
36      // free
37      vem_clear_clientinfo(clients);
38      }
39    } else {
40    // error connecting
41     fprintf(stderr, "Error geting clients: %s\n",  vem_strerror(vemerrno));
42    }
```

Lines 43-47: authenticate the user to Platform EGO.

Lines 48-52: define and initialize a structure for callback methods. These callback methods are invoked by Platform EGO when resources are added or reclaimed, or when a change occurs to host status or a container. When Platform EGO wants to communicate about these events, it invokes these methods thereby calling back to the client.

Lines 54-68: define the vem_allocation_info_reply_t and vem_container_info_reply_t structures. If a client gets disconnected and then re-registers, its existing allocations and containers are returned to these structures.

If the client had never registered before, the structures would be empty. Define and initialize a structure (rreq) that holds client info for registration purposes. (This includes assigning the client callback structure (cbf) to the callback member of the rreq structure.) Register with Platform EGO via the open connection using vem_register().

```
43 if (login(vhandle, username, password)<0) {
44    fprintf(stderr, "Error logon: %s\n",
45  vem_strerror(vemerrno));
46    goto leave;
47   }
48 vem_clientcallback_t cbf;
49   cbf.addResource = addResourceCB;
50   cbf.reclaimForce = reclaimForceCB;
51   cbf.containerStateChg = containerStateChgCB;
52   cbf.hostStateChange = hostStateChangeCB;
53
54   vem_allocation_info_reply_t aireply;
55   vem_container_info_reply_t  cireply;
56   vem_registerreq_t rreq;
57
58   rreq.name = "sample6_client";
59   rreq.description = "Sample6 Client";
60   rreq.flags = VEM_REGISTER_TTL;
61   rreq.ttl = 3;
62   rreq.cb = &cbf;
63 rc = vem_register(vhandle, &rreq, &aireply, &cireply);
64   if (rc < 0) {
65     fprintf(stderr, "Error registering: %s\n",
66  vem_strerror(vemerrno));
67    goto leave;
68   }
```

Lines 69-72: print out information related to the allocation requests and containers. Once the info is printed out, the memory for the allocations is freed.

Lines 77-82: the vem_gethostgroupinfo() method collects the information for the requested hostgroup. In this case, the requested hostgroup in the input argument is set to NULL, which means that information about all hostgroups is requested. If the method call is successful, hostgroup information is printed out to the screen.

```
69  print_vem_allocation_info_reply(&aireply);
70   print_vem_container_info_reply(&cireply);
71   // freeup any previous allocations
72   release_vem_allocation(vhandle, &aireply);
73
74 vem_hostgroupreq_t hgroupreq;
75   hgroupreq.grouplist = NULL;
76   vem_hostgroup_t *hgroup;
77 rc = vem_gethostgroupinfo(vhandle, &hgroupreq, &hgroup);
78   if (rc < 0) {
79     fprintf(stderr, "Error getting hostgroup: %s\n",  vem_strerror(vemerrno));
80   } else {
81   printf("%s %s %d %d\n", hgroup->groupName, hgroup->members, hgroup->free,
   hgroup->allocated);
82   }
```

Lines 83-86: initialize the service_stateP and client_stateP structures. These structures contain the respective mutex objects and condition variables.

Lines 87-88: create a service definition in XML format. Each service is described by an XML file that contains information about the service such as the type of resources required to run the service instances and how to start and monitor them. Store the service definition in the service_stateP structure.

Lines 89-93: create and run the service_thread. The service thread is responsible for defining and creating the EGO service.

Lines 94-109: use vem_locate() to get all registered clients. Since NULL is provided as the client name, all registered clients will be located and the method returns the number of registered clients. If successful, print out the client info (name, description, and location) and free the associated memory.

```
83 service_stateP = calloc(1, sizeof(service_state_t));
84   initialize_service(service_stateP);
85   client_state_t *client_stateP = calloc(1, sizeof(client_state_t));
86   initialize_client(client_stateP);
87   char *xml = get_service_def_xml();
88   service_stateP->xml = xml;
89   pthread_t service_thread;
90   if (pthread_create(&service_thread, NULL, service_thread_fn,
91   service_stateP)) {
92     perror("Error creating service thread: ");
93   }
94   rc = vem_locate(vhandle, NULL, &clients);
95   if (rc >=0) {
96     if (rc == 0) {
97      printf("No registered clients exist\n");
98     } else {
99     int i=0;
100    for (i=0; i<rc; i++) {
101       printf("%s %s %s\n", clients[i].name, clients[i].description,
102       clients[i].location);
103    }
104      vem_clear_clientinfo(clients);
105    }
106  } else {
107  // error connecting
108   fprintf(stderr, "Error geting clients: %s\n",  vem_strerror(vemerrno));
109  }
```

Lines 114-120: lock the service_mutex object and wait until the service is running. Set the service condition variable to unblock the service thread, which causes the service thread to query the service.

Lines 124-127: lock the client_mutex object and block the main thread with the client condition variable. When the service info is available, the service thread signals the main thread to resume execution using the client condition variable. The main thread prints out the service info reply.

Line 130: set the shutdown flag, which causes the service thread to disable and remove the service.

Line 134: block execution of the main thread until the service thread has terminated.

```
110 sleep(60);
111   // get service info
112   service_info_reply = calloc(1, sizeof(esc_service_info_reply_t));
113   int succeed=0;
114   while(!succeed) {
115     pthread_mutex_lock(&service_stateP->service_mutex);
116     if (service_stateP->ready && service_stateP->client == NULL) {
117       service_stateP->client = client_stateP;
118       pthread_cond_signal(&service_stateP->service_cond);
119       succeed = 1;
120     }
121     pthread_mutex_unlock(&service_stateP->service_mutex);
122   }
123   fprintf(stderr, "Sent Request to Service:\n");
124   pthread_mutex_lock(&client_stateP->client_mutex);
125   pthread_cond_wait(&client_stateP->client_cond,
126  &client_stateP->client_mutex);
127   print_service_info_reply(service_info_reply);
128   pthread_mutex_unlock(&client_stateP->client_mutex);
129 sleep(60);
130   shutdown = 1;
131   pthread_mutex_lock(&service_stateP->service_mutex);
132   pthread_cond_signal(&service_stateP->service_cond);
133   pthread_mutex_unlock(&service_stateP->service_mutex);
134   pthread_join(service_thread, NULL);
```

Step 3: Create the service

In this sample, the service definition and creation is handled by the service thread. The service simply executes the sleep command for 60 milliseconds and then ends.

Lines 137-147: set the EGO_CONFDIR environment variable to the current working directory so that the Service Controller, which is just another EGO client, can find the EGO configuration. The configuration is stored in the ego.conf file.

Lines 148-151: define and initialize a security structure. The username and password variables have been initialized with "egoadmin" from sample # 2.

Lines 153-154: print out the service definition.

Lines 155-163: pass the security structure and the service definition to the esc_createservice() method. This API call creates a new service object. If startType is automatic in the service definition, the service is enabled automatically. In this case, the startType is set to manual so the service must be enabled by the esc_enableservice() method. This API call starts the service. Once the service is

started, the Service Controller allocates resources and starts service instances. A service can be started by esc_enableservice() or by starting a service that depends on it.

```
135 void *service_thread_fn(service_state_t *service_stateP)
136 {
137 int size=4096;
138   char *buf = calloc(size, sizeof(char));
139   sprintf(buf, "EGO_CONFDIR=");
140   char *cwd = getcwd((buf+12), size-12);
141   int errn =  errno;
142   if(cwd != NULL) {
143     putenv(buf);
144   } else {
145   fprintf(stderr, "Error getting CWD: %s\n",
146  strerror(errn));
147   }
148   esc_security_def_t security;
149   security.username = username;
150   security.password = password;
151   security.credential = NULL;
152   char *sname = "sample6_service";
153   char *xml = service_stateP->xml;
154   fprintf(stderr, "%s\n", xml);
155   if(esc_createservice(xml, &security)) {
156     fprintf(stderr, "Error creating service: %s\n",
157  esc_strerror(escerrno));
158     //goto bailout;
159   }
160 if(esc_enableservice(sname, &security)) {
161     fprintf(stderr, "Error enabling service: %s\n",
162  esc_strerror(escerrno));
163   }
```

Step 4: Query the service

Lines 165-175: lock the service_mutex object and wait until the service condition variable is set by the main thread, which is acting like a client requesting service information. (In order to wait on a condition variable, the associated mutex has to be locked first. Inside the wait function, the mutex will be automatically unlocked so anyone can then acquire it. On returning from wait, the mutex is automatically acquired by the service thread.) When the service thread resumes execution, pass the name of the service to the esc_queryservice()API method, which gets the service instance information from the EGO Service Controller.

Lines 176-177: lock the client_mutex object and print out the service information.

Line 178: since the service info has been retrieved, notify the client by setting the client condition variable. The main thread resumes execution.

Lines 179-181: unlock the client and service mutex objects and initialize the client structure.

```
164    while(!shutdown) {
165     pthread_mutex_lock(&service_stateP->service_mutex);
166     service_stateP->ready=1;
167    pthread_cond_wait(&service_stateP->service_cond,
168  &service_stateP->service_mutex);
169
170     fprintf(stderr, "Got Request:\n");
171     if (service_stateP->client == NULL) continue;
172     if(esc_queryservice(sname, service_info_reply)) {
173       fprintf(stderr, "Error geting service info: %s\n",
174  esc_strerror(escerrno));
175     }
176  pthread_mutex_lock(&service_stateP->client->client_mutex);
177       print_service_info_reply(service_info_reply);
178  pthread_cond_signal(&service_stateP->client->client_cond);
179 pthread_mutex_unlock(&service_stateP->client->client_mutex);
180       service_stateP->client = NULL;
181       pthread_mutex_unlock(&service_stateP->service_mutex);
```

Step 5: Disable and remove the service

Lines 182-190: disable the service by calling the esc_disableservice() API method. The Service Controller stops all service instances and de-allocates resources. Remove the service by calling the esc_removeservice() API method. The Service Controller destroys the service object and removes the service definition from the configuration.

```
182  if(esc_disableservice(sname, &security)) {
183     fprintf(stderr, "Error disabling service: %s\n",
184  esc_strerror(escerrno));
185   }
186
187   if(esc_removeservice(sname, &security)) {
188     fprintf(stderr, "Error removing service: %s\n",
189  esc_strerror(escerrno));
190   }
```

Run the client application

1. Select **Run > Run**.

   The Run dialog appears.
2. In the Configurations list, either select an EGO C Client Application or click **New** for a new configuration.

   For a new configuration, enter the configuration name.
3. Enter the project name and C/C++ Application name.
4. Click **Apply** and then **Run**.

Tutorial 7: Update a DNS Entry in the Service Director

This tutorial describes how to create a service and add a DNS entry for the service in the Platform EGO Service Director.

Using this tutorial, you will …

- Open a connection to Platform EGO
- Print out cluster information
- Check if there are any registered clients connected to Platform EGO
- Log on to Platform EGO
- Register the client with Platform EGO
- Print out allocation and container reply info from a previous connection
- Print out host group information
- Request resource allocation from Platform EGO and print the allocation ID
- Create, run, and query a service
- Query for the IP address of the host where the service is running
- Update the Service Director with a new entry for service instance location.

Underlying principles

Platform EGO features a plug-in model for the Service Director. There is a default plug-in that comes with Platform EGO, however different Service Director elements can be installed. This sample uses the default plug-in, which implements a DNS server for storing location information about the service instances. At runtime, the Service Director's shared library can be loaded, and appropriate methods (declared in the sdplugin.h file) can be looked up and invoked.

In order to communicate with a service on a host cluster, its location must be known. Due to the nature of distributed computing, the service can be running on any host. Normally, when a service instance switches into the run state, the Service Controller sends a notification to the Service Director that includes location information of the service instance. The Service Director then adds the location record to its DNS server. In this sample, we create a service called "Sample7_mysql_service". When this service is running, the Service Director automatically updates its records to reflect the new service instance. However, to demonstrate how to manually update the Service Director records, we also create an alias to this service instance and query the service to reveal its network address.

Step 1: Preprocessor directives and method declarations

The first step is to include a reference to the system and API header files, followed by the declaration of methods and variables that are implemented in the sample.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <pthread.h>
#include <dlfcn.h>
#include <netdb.h>

#include "vem.api.h"
#include "esc.api.h"
#include "samples.h"
#include "esd.h"
#include "esdplugin.h"

static int addResourceCB(vem_allocreply_t *areply);
static int reclaimForceCB(vem_allocreclaim_t *areclaim);
static int containerStateChgCB(vem_containerstatechg_t *cschange);
static int hostStateChangeCB(vem_hoststatechange_t *hschange);
static char *get_service_def_xml();

void printSDEntry(char *sname);
service_state_t *service_stateP;
esc_service_info_reply_t *service_info_reply;
```

Step 2: Implement the principal method

Lines 4-7: define and initialize a data structure that is used to request a connection with the EGO host cluster. The data structure contains a reference to a configuration file where the master host name and port numbers are stored.

Line 8: pass the data structure as an argument to the vem_open () method, which opens a connection to the master host. If the connection attempt is successful, a handle is returned; otherwise the method returns NULL. The handle acts as a communication channel to the master host and all subsequent communication occurs through this handle.

Lines 14-15: the vem_name_t structure (defined as clusterName) is initialized with NULL. This structure holds the cluster name, system name, and version. The vem_uname () method is passed the communication handle and, if successful, returns a valid vem_name_t structure ; otherwise the method returns NULL.

Line 23: the cluster info is printed out to the screen.

Lines 26-43: define the client info structure. Use vem_locate() to get all registered clients. Since NULL is provided as the client name, all registered clients will be located and the method returns the number of registered clients. Note that Platform

EGO is equipped with a number of default clients (services) such as the Service Controller, so as a minimum, the info relevant to these clients is printed out and the associated memory is released.

```
1   int
2   sample7()
3   {
4     vem_openreq_t orequest;
5     vem_handle_t *vhandle = NULL;
6     orequest.file = "ego.conf";
7     orequest.flags=0;
8     vhandle = vem_open(&orequest);
9     if (vhandle == NULL) {
10    // error opening
11     fprintf(stderr, "Error opening cluster: %s\n",  vem_strerror(vemerrno));
12     return -1;
13    }
14    vem_name_t *clusterName = NULL;
15    clusterName = vem_uname(vhandle);
16    if (clusterName == NULL) {
17    // error connecting
18     fprintf(stderr, "Error connecting to cluster: %s\n",
19   vem_strerror(vemerrno));
20     return -2;
21    }
22
23    fprintf(stdout, " Connected... %s %s %4.2f\n", clusterName->clustername,
24   clusterName->sysname, clusterName->version);
25
26    vem_clientinfo_t *clients;
27    int  rc = vem_locate(vhandle, NULL, &clients);
28    if (rc >=0) {
29      if (rc == 0) {
30       printf("No registered clients exist\n");
31      } else {
32      int i=0;
33      for (i=0; i<rc; i++) {
34         printf("%s %s %s\n", clients[i].name, clients[i].description,
35         clients[i].location);
36      }
37      // free
38      vem_clear_clientinfo(clients);
39      }
40    } else {
41    // error connecting
42     fprintf(stderr, "Error geting clients: %s\n",  vem_strerror(vemerrno));
43    }
```

Lines 44-46: authenticate the user to Platform EGO.

Lines 48-52: define and initialize a structure for callback methods. These callback methods are invoked by Platform EGO when resources are added or reclaimed, or when a change occurs to host status or a container. When Platform EGO wants to communicate about these events, it invokes these methods thereby calling back to the client.

Lines 54-63: define the vem_allocation_info_reply_t and vem_container_info_reply_t structures. If a client gets disconnected and then re-registers, its existing allocations and containers are returned to these structures. If the client had never registered before, the structures would be empty. Define and initialize a structure (rreq) that holds client info for registration purposes. (This includes assigning the client callback structure (cbf) to the callback member of the rreq structure.) Register with Platform EGO via the open connection using vem_register().

```
44 if (login(vhandle, username, password)<0) {
45    fprintf(stderr, "Error logon: %s\n",  vem_strerror(vemerrno));
46    goto leave;
47   }
48 vem_clientcallback_t cbf;
49   cbf.addResource = addResourceCB;
50   cbf.reclaimForce = reclaimForceCB;
51   cbf.containerStateChg = containerStateChgCB;
52   cbf.hostStateChange = hostStateChangeCB;
53
54   vem_allocation_info_reply_t aireply;
55   vem_container_info_reply_t  cireply;
56   vem_registerreq_t rreq;
57 rreq.name = "sample7_client";
58   rreq.description = "Sample7 Client";
59   rreq.flags = VEM_REGISTER_TTL;
60   rreq.ttl = 3;
61   rreq.cb = &cbf;
62
63   rc = vem_register(vhandle, &rreq, &aireply, &cireply);
64   if (rc < 0) {
65     fprintf(stderr, "Error registering: %s\n",  vem_strerror(vemerrno));
66    goto leave;
67   }
```

Lines 68-71: print out information related to the allocation requests and containers. Once the info is printed out, the memory for the allocations is freed.

Lines 73-84: the vem_gethostgroupinfo() method collects the information for the requested hostgroup. In this case, the requested hostgroup in the input argument is set to NULL, which means that information about all hostgroups is requested. If the method call is successful, hostgroup information is printed out to the screen.

```
68 print_vem_allocation_info_reply(&aireply);
69   print_vem_container_info_reply(&cireply);
70   // freeup any previous allocations
71   release_vem_allocation(vhandle, &aireply);
72
73   vem_hostgroupreq_t hgroupreq;
74   hgroupreq.grouplist = NULL;
75   vem_hostgroup_t *hgroup;
76
77   rc = vem_gethostgroupinfo(vhandle, &hgroupreq, &hgroup);
78   if (rc < 0) {
79     fprintf(stderr, "Error getting hostgroup: %s\n",
80  vem_strerror(vemerrno));
81   } else {
82   printf("%s %s %d %d\n", hgroup->groupName, hgroup->members, hgroup->free,
83  hgroup->allocated);
84   }
```

Lines 85-88: define and initialize the service_stateP and client_stateP structures. These structures contain the respective mutex objects and condition variables.

Lines 90-91: get the service definition and store it in the service state structure. Refer to Step 3: Create the service definition on page 106.

Lines 93-97: create and run the service_thread. The service thread is responsible for defining and creating the Platform EGO service.

```
85 service_stateP = calloc(1, sizeof(service_state_t));
86   initialize_service(service_stateP);
87   client_state_t *client_stateP = calloc(1, sizeof(client_state_t));
88   initialize_client(client_stateP);
89
90   char *xml = get_service_def_xml();
91   service_stateP->xml = xml;
92
93   pthread_t service_thread;
94   if (pthread_create(&service_thread, NULL, service_thread_fn,
95  service_stateP)) {
96     perror("Error creating service thread: ");
97   }
```

Lines 98-113: use vem_locate() to get all registered clients. Since NULL is provided as the client name, all registered clients will be located and the method returns the number of registered clients. If successful, print out the client info (name, description, and location) and free the associated memory.

Lines 120-123 lock the service_mutex object and wait until the service is running. Set the service condition variable to unblock the service thread, which causes the service thread to query the service.

Lines 129-132: lock the client_mutex object and block the main thread with the client condition variable. When the service info is available, the service thread signals the main thread to resume execution using the client condition variable. The main thread prints out the service info reply.

```
98  rc = vem_locate(vhandle, NULL, &clients);
99   if (rc >=0) {
100    if (rc == 0) {
101     printf("No registered clients exist\n");
102    } else {
103    int i=0;
104    for (i=0; i<rc; i++) {
105       printf("%s %s %s\n", clients[i].name, clients[i].description,
106       clients[i].location);
107    }
108      vem_clear_clientinfo(clients);
109    }
110  } else {
111  // error connecting
112   fprintf(stderr, "Error geting clients: %s\n",  vem_strerror(vemerrno));
113  }
114  sleep(60);
115
116  // get service info
117  service_info_reply = calloc(1, sizeof(esc_service_info_reply_t));
118  int succeed=0;
119  while(!succeed) {
120    pthread_mutex_lock(&service_stateP->service_mutex);
121    if (service_stateP->ready && service_stateP->client == NULL) {
122      service_stateP->client = client_stateP;
123      pthread_cond_signal(&service_stateP->service_cond);
124      succeed = 1;
125    }
126    pthread_mutex_unlock(&service_stateP->service_mutex);
127  }
128  fprintf(stderr, "Sent Request to Service:\n");
129  pthread_mutex_lock(&client_stateP->client_mutex);
130  pthread_cond_wait(&client_stateP->client_cond,
131 &client_stateP->client_mutex);
132  print_service_info_reply(service_info_reply);
133  pthread_mutex_unlock(&client_stateP->client_mutex);
```

Lines 134-140: store the service info in the esc_service_info_t structure and retrieve the number of service instances. Assign the service instance info to the instance_info structure. Allocate and initialize an array in memory to hold the IP address associated with each service instance.

Lines 144- 149: for each service instance, retrieve the host name where the service instance is running and pass it to the gethostbyname() method. The method returns a pointer to a hostent structure (locationService), the members of which contain the fields of an entry in the network host database. The hostent structure is defined in the <netdb.h> header. Extract the host's IP address from the locationService structure and assign it to the ipAddresses array.

Lines 149-161: pass "mysql", an alias for "Sample7_mysql_service", to the printSDEntry() method. The method will query the service and print the IP address of the host that the service is running on. However, since there is currently no entry

in the Service Director records for a service called "mysql", nothing is printed. (The printSDEntry() method is used again later in the sample to print out the IP address after the Service Director records have been properly updated.)

```
134 esc_service_info_t *service_info = &service_info_reply->serviceV[0];
135  if (service_info == NULL) {
136     fprintf (stderr, "Could not get Service Info\n");
137     goto done;
138  }
139  int i, num_instances;
140  num_instances = service_info->instC;
141  int ipaddrC = num_instances;
142  char **ipAddresses = calloc(ipaddrC, sizeof(char*));
143
144  for(i=0; i < num_instances; i++) {
145  esc_service_instance_info_t *instance_info = &service_info->instV[i];
146    char *location = instance_info->host;
147    struct hostent * locationService = gethostbyname(location);
148    ipAddresses[i] = locationService->h_addr_list[0];
149  } char *sname = "mysql";
150  printSDEntry(sname);
```

```
151 void
152 printSDEntry(char *sname)
153 {
154   int i=0;
155   struct hostent * dnsService = gethostbyname(sname);
156   fprintf(stderr, "Existing Entry for %s\n", sname);
157   if (dnsService != NULL) {
158   for (i=0; i<dnsService->h_length; i++) {
159       fprintf(stderr, "%s\n", dnsService->h_addr_list[i]);
160   }
161   }
```

The following paragraphs demonstrate how to manually add a new entry in the Service Director that provides service instance location information.

Lines 162-167: define and initialize a structure (update) to hold service instance info that will be used to add a new record in the Service Director.

Lines 169-175: use dlopen() to open a shared library (esd_ego_default.so); this is the default Service Director plug-in. The dlopen() method returns a handle to the shared library. Define a structure to initialize all the plug-in variables. Pass the shared library handle and a function name (esd_plugin_initialize) to dlsym(). The method looks up the address of the function in the shared library handle obtained from dlopen.

Lines 179-189: set up the necessary plug-in variables and invoke the initialization method from the shared library.

Lines 190-191: pass the update structure to the update_service() method. The method adds a service location entry in the Service Directorconsisting of an alias (mysql) with the same IP address that was printed by the previous service query in the service thread; see Tutorial 6: Step 4: Query the service on page 96. The printSDEntry() method prints out the new service location entry.

```
162  si_updrec_t update;
163   sd_update_oper_t op = op_add;
164 update.name = sname;
165   update.ipaddrC = ipaddrC;
166   update.ipaddrV = ipAddresses;
167   update.oper = op;
168
169   void * esd_plugin = dlopen("esd_ego_default.so",  RTLD_LAZY);
170   if (esd_plugin == NULL) {
171      fprintf (stderr, "%s\n", dlerror());
172   }
173 esd_plugin_initialize_t esd_plugin_init;
174
175   void *fptr = dlsym(esd_plugin, "esd_plugin_initialize");
176   if (fptr == NULL) {
177      fprintf (stderr, "%s\n", dlerror());
178   }
179   * (void **) &esd_plugin_init =  fptr;
180
181   esd_utility_funcs_t utility_functions;
182   utility_functions.calloc = calloc;
183   utility_functions.malloc = malloc;
184   utility_functions.free = free;
185   utility_functions.realloc = realloc;
186   utility_functions.strdup =  strdup;
187 const char *param = "";
188   esd_plugin_t sd_plugin;
189 (*esd_plugin_init)(&utility_functions, param, &sd_plugin);
190  sd_plugin.update_service(&update);
191 printSDEntry(sname);
```

Step 3: Create the service definition

Create a service definition in XML format. Each service is described by an XML file that contains information about the service such as the type of resources required to run the service instances and how to start and monitor them.

```
char * get_service_def_xml()
{
char *xml =
"<?xml version=\"1.0\" encoding=\"UTF-8\"?>\
<sc:ServiceDefinition xmlns:sc=
\"http://www.platform.com/ego/2005/05/schema/sc\" xmlns:ego=
\"http://www.platform.com/ego/2005/05/schema\" xmlns:xsi=
\"http://www.w3.org/2001/XMLSchema-instance\" xmlns:xsd=
\"http://www.w3.org/2001/XMLSchema\" xsi:schemaLocation=
\"http://www.platform.com/ego/2005/05/schema/sc ../sc.xsd
 http://www.platform.com/ego/2005/05/schema ../ego.xsd\" ServiceName=
\"Sample7_mysql_service\">\
  <sc:Description>\"Sample 7 MySQL Service\"</sc:Description>\
  <sc:MinInstances>1</sc:MinInstances>\
  <sc:MaxInstances>1</sc:MaxInstances>\
  <sc:Priority>10</sc:Priority>\
  <sc:MaxInstancesPerSlot>1</sc:MaxInstancesPerSlot>\
  <sc:ControlPolicy>\
    <sc:StartType>MANUAL</sc:StartType>\
    <sc:MaxRestarts>10</sc:MaxRestarts>\
    <sc:HostFailoverInterval>60s</sc:HostFailoverInterval>\
  </sc:ControlPolicy>\
  <ego:AllocationSpecification>\
    <ego:ConsumerID>/SampleApplications/EclipseSamples</ego:ConsumerID>\
    <!-- The ResourceType specifies a ""compute element"" identified by the
 URI used below -->\
    <ego:ResourceSpecification ResourceType=
\"http://www.platform.com/ego/2005/05/schema/ce\">\
      <ego:ResourceGroupName>ComputeHosts</ego:ResourceGroupName>\
      <ego:ResourceRequirement>LINUX86</ego:ResourceRequirement>\
    </ego:ResourceSpecification>\
  </ego:AllocationSpecification>\
  <sc:ContainerDescription>\
    <ego:Attribute name=\"hostType\" type=\"xsd:string
\">LINUX86</ego:Attribute>\
    <ego:ContainerSpecification>\
      <ego:Command>bin/mysql --user mysql</ego:Command>\
      <ego:RunAsOSUser>root</ego:RunAsOSUser>\
      <ego:WorkingDirectory>/usr/local/mysql</ego:WorkingDirectory>\
      <ego:Umask>0777</ego:Umask>\
    </ego:ContainerSpecification>\
  </sc:ContainerDescription>\
</sc:ServiceDefinition>\";
return xml;
}
```

Step 4: Create the service

In this sample, service creation is handled by the service thread. The service simply starts the mySQL program.

Lines 194-204: set the EGO_CONFDIR environment variable to the current working directory so that the Service Controller, which is just another Platform EGO client, can find the Platform EGO configuration. The configuration is stored in the ego.conf file.

Lines 205-208: define and initialize a security structure. The username and password variables have been initialized with "egoadmin" from sample # 2.

Line 210: create a service definition in XML format. Each service is described by an XML file that contains information about the service such as the type of resources required to run the service instances and how to start and monitor them.

Lines 212-219: pass the security structure and the service definition to the esc_createservice() method. This API call creates a new service object. If startType is automatic in the service definition, the service is enabled automatically. In this case, the startType is set to manual so the service must be enabled by the esc_enableservice() method. This API call starts the service. Once the service is started, the Service Controller allocates resources and starts service instances. A service can be started by esc_enableservice() or by starting a service that depends on it.

```
192 void *service_thread_fn(service_state_t *service_stateP)
193 {
194 int size=4096;
195   char *buf = calloc(size, sizeof(char));
196   sprintf(buf, "EGO_CONFDIR=");
197   char *cwd = getcwd((buf+12), size-12);
198   int errn =  errno;
199   if(cwd != NULL) {
200     putenv(buf);
201   } else {
202   fprintf(stderr, "Error getting CWD: %s\n",
203  strerror(errn));
204   }
205   esc_security_def_t security;
206   security.username = username;
207   security.password = password;
208   security.credential = NULL;
209   char *sname = "sample6_service";
210   char *xml = get_service_def_xml();
211
212   if(esc_createservice(xml, &security)) {
213     fprintf(stderr, "Error creating service: %s\n",
214  esc_strerror(escerrno));
215     //goto bailout;
216   }
217 if(esc_enableservice(sname, &security)) {
218     fprintf(stderr, "Error enabling service: %s\n",
219  esc_strerror(escerrno));
220   }
```

Step 5: Client callback methods

These callback methods are invoked by Platform EGO when resources are added or reclaimed, or when a change occurs to host status or a container. When Platform EGO wants to communicate about these events, it invokes these methods thereby calling back to the client.

Lines 221-227: this method is called by Platform EGO when resources have been added to an allocation in order to tell the client which resources have been provided for its use. This method prints out the allocation and consumer IDs, the number of hosts allocated, host names and number of slots, and host attributes.

Lines 229-235: this method is called by Platform EGO when resources need to be reclaimed. Resources may be reclaimed either for policy reasons, or because a resource has been found to be down or unavailable. The method prints out the host info including host name and slots for each host being reclaimed.

Lines 237-243: this method is called by Platform EGO in order to communicate status changes in containers to the clients that started them. The method prints out the container ID and its associated state.

Lines 245-251: this method is called by Platform EGO when a host changes state. The method prints out the host name and its new host state.

```
221 int
222 addResourceCB(vem_allocreply_t *areply)
223 {
224 printf("addResource Call Back\n");
225     print_vem_allocreply(areply);
226     return 0;
227 }
228
229 int
230 reclaimForceCB(vem_allocreclaim_t *areclaim)
231 {
232 printf("reclaimForce Call Back\n");
233 print_vem_allocreclaim(areclaim);
234 return 0;
235 }
236
237 int
238 containerStateChgCB(vem_containerstatechg_t *cschange)
239 {
240 printf("containerStateChg Call Back\n");
241 printf("%s %d\n", cschange->containerId, cschange->newState);
242     return 0;
243 }
244
245 int
246 hostStateChangeCB(vem_hoststatechange_t *hschange)
247 {
248 printf("hostStateChange Call Back\n");
249 printf("%s %d\n", hschange->name, hschange->newState);
250     return 0;
251 }
```

Run the client application

1. Select **Run > Run**.

   The Run dialog appears.
2. In the Configurations list, either select an EGO C Client Application or click **New** for a new configuration.

   For a new configuration, enter the configuration name.
3. Enter the project name and C/C++ Application name.
4. Click **Apply** and then **Run**.

CHAPTER

4

Getting Started with the Web Service Client: A Collection of Tutorials

Before you begin the tutorials

One fact about Web Service clients is that they can be written in almost any programming language. In this series of tutorials, the sample code has been developed in the Java programming language.

Before starting these tutorials, ensure that the EGO Web Service API plug-in is installed in Eclipse and the Platform EGO runtime has been installed, configured, and running on a host cluster. The sample programs use a URL or IP address and port number to communicate with a Web Service gateway in the master host. The gateway is installed as part of Platform EGO. It comes with a configuration file called wsg.conf where the port number for the gateway can be defined. By default the port number is 9090. The default location for the wsg.conf file is /opt/ego/kernel/conf/. The gateway URL can then be formed by using the hostname where the gateway is running and the port number, i.e., http://host:port. This URL is passed as an input argument to the client samples.

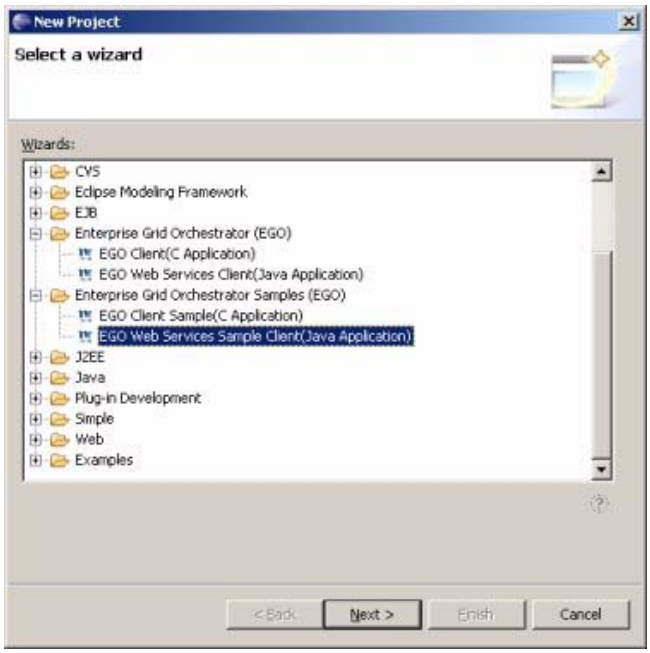
Also, try to connect to the Web Service gateway using a browser by entering the URL or IP address and port number of the gateway. If you cannot connect through the browser, you will not be able to run the samples.

Contents

Locate the code samples

1. Launch Eclipse.
2. Select **File > New > Project**.
3. In the New Project dialog, expand **Enterprise Grid Orchestrator Samples (EGO)** and select **EGO Web Services Sample Client (Java Application)**. Click **Next**.
4. In the EGO Sample Project dialog, enter a project name. To use default project settings, click **Finish**, or, to adjust the settings, click **Next** . Note that in order to run the SDK samples, the compliance level must be set to 5.0 since the samples use Java 2 Standard Edition (J2SE) 5.0 features. Click **Finish** when settings are complete.
5. In the Package Explorer view, expand the project to see the list of code samples. Note: If the Package Explorer is not visible, select **Window > Show View > Package Explorer.**
6. Double-click the sample file.

   The sample code appears in the main view.



Tutorial 1: Request Information About Hosts in a Cluster

This tutorial describes the minimum amount of code required to create an unregistered EGO client that connects to a host cluster.

Using this tutorial, you will …

- Open a connection to the EGO Web Service endpoint
- Retrieve and print out cluster info
- Retrieve and print out resource info

Step 1: Import class references

Import the necessary classes and interfaces that are required by the client to invoke the Web Service.

Step 2: Retrieve cluster information

Check if the client can connect to the MonitoringPortType endpoint in the Web Service. If successful, the next step is to create an XML document for requesting cluster information.

```
public void clusterInfo() {
if (monitor == null) {
System.err.println("Could not find MonitorPort...");
return;
}
ClusterInfoRequestDocument requestDoc = ClusterInfoRequestDocument.Factory
.newInstance();
ClusterInfoRequest request = requestDoc.addNewClusterInfoRequest();
ClusterInfoResponseDocument responseDoc;
ClusterInfoResponse response;

// no security check needed for cluster info
SecurityDocument sdoc1 = SecurityDocument.Factory.newInstance();
SecurityHeaderType sec1 = sdoc1.addNewSecurity();
SecurityDocument sdoc2 = SecurityDocument.Factory.newInstance();
SecurityHeaderType sec2 = sdoc2.addNewSecurity();
SecurityDocument sdoc3 = SecurityDocument.Factory.newInstance();
SecurityHeaderType sec3 = sdoc3.addNewSecurity();
SecurityDocument sdoc4 = SecurityDocument.Factory.newInstance();
SecurityHeaderType sec4 = sdoc4.addNewSecurity();

try {
responseDoc = monitor.ClusterInfo(requestDoc, sdoc1, sdoc2, sdoc3,
sdoc4);
response = responseDoc.getClusterInfoResponse();
print(response);
} catch (RemoteException rex) {
rex.printStackTrace();
return;
}
}
```

The ClusterInfoRequestDocument and ClusterInfoResponseDocument classes represent XML and the ClusterInfoRequest and ClusterInfoResponse classes represent the data.

Since client registration is not required for querying host information, security is not an issue. The Web Service, however, expects security documents so they are simply instantiated without any security information.

An object of type ClusterInfoRequestDocument is passed to the local proxy method, ClusterInfo(). The Web Service returns the cluster information and the result is printed out to the console using the overloaded print() method.

Step 3: Retrieve resource information

The code required to retrieve resource information is similar to retrieving cluster information. Check if the client can connect to the MonitoringPortType endpoint in the Web Service. If successful, create a ResourceInfoRequestDocument object

and a ResourceInfoRequest object to hold the request data. Since we want to retrieve the information from all resources, the ResourceInfoRequest object is empty.

Create the security documents but leave them empty since security information is not required for resource queries.

Next, we pass the ResourceInfoRequestDocument and the security documents to the local proxy method, ResourceInfo(). The result of the operation is returned to the ResourceInfoResponseDocument object. The data is extracted from the document into the ResourceInfoResponse object and then printed out to the console using the overloaded print() method.

```
public Resource [] resourceInfo(String[] resourceNames) {
if (monitor == null) {
System.err.println("Could not find MonitorPort...");
return null;
}

try {
ResourceInfoRequestDocument rreqDoc = ResourceInfoRequestDocument.Factory
.newInstance();
ResourceInfoRequest rreq = rreqDoc.addNewResourceInfoRequest();
rreq.setResourceNameArray(resourceNames);

ResourceInfoResponseDocument rresDoc = monitor.ResourceInfo(
rreqDoc);
ResourceInfoResponse rres = rresDoc.getResourceInfoResponse();
print(rres);
Resource [] resources = rres.getResourceArray();
return resources;
catch (RemoteException rex) {
rex.printStackTrace();
return null;
}
}
```

Step 4: Print the resource information

```
public void print(ClusterInfoResponse response) {
ClusterInfo [] cinfos = response.getClusterInfoArray();
for(int i=0; i<cinfos.length; i++) {
ClusterInfo cinfo = cinfos[i];
    System.out.printf("%-12s\t\t%-12s\t\t%-12s%n",
    cinfo.getClusterName(), cinfo.getVersion());
}
}
public void print(ResourceInfoResponse response) {
Resource[] resources = response.getResourceArray();
if(resources != null) {
  // print header
  System.out.printf("%-12s\t\t", "Attribute");
  for(int i=0; i<resources.length; i++) {
  System.out.printf("%-12s\t\t", resources[i].getResourceName());
  }
  System.out.println();
  System.out.printf("%-12s\t\t", "Type");
  for(int i=0; i<resources.length; i++) {
  System.out.printf("%-12s\t\t", resources[i].getResourceType());
  }
  System.out.println();

  System.out.printf("%-12s\t\t", "State");
  for(int i=0; i<resources.length; i++) {
  System.out.printf("%-12s\t\t", resources[i].getResourceState());
  }
  System.out.println();
}
// assumes all resources have same attributes
        Attribute [] attributes = resources[0].getAttributeArray();
   int numAttrs = attributes.length;
for(int j=0; j<numAttrs; j++) {
  System.out.printf("%-12s\t\t", attributes[j].getName());
  for (int i = 0; i < resources.length; i++) {
Resource r = resources[i];
System.out.printf("%-12s\t\t", r.getAttributeArray(j).getStringValue());
  }
  System.out.println();
}
}
```

```
public void print(Resource res) {
String name = res.getResourceName();
String type = res.getResourceType();
Enum state = res.getResourceState();
System.out.printf("%-12s%n%-12s%n%-12s%n",
name, type, state.toString());
Attribute[] attributes = res.getAttributeArray();

for (int i = 0; i < attributes.length; i++) {
Attribute attr = attributes[i];
print(attr);
}
}

public void print(Attribute a) {
String name = a.getName();
String value = a.getStringValue();
System.out.printf("%-12s\t\t%-12s%n", name, value);
}
public void print(ClientInfo cinfo) {
System.out.printf("%-12s\t%-12s\t%-12s%n",
cinfo.getClientName(), cinfo.getClientDescription(),
cinfo.getClientLocation());
}
```

Step 5: Call the sample program

In this step, we simply create an EGOClient object and call its serviceInfo() method from the main method.

```
...
public static void main(String[] args) throws Exception {
if(args==null || args.length <1) {
throw new Exception("Incorrect Arguments");
}
EGOclient client = new EGOclient(args[0]);
client.serviceInfo();
}
```

Run the client application

1. Select **Run > Run**.

   The Run dialog appears.
2. In the Configurations list, either select a Java Application or click **New** for a new configuration.

   For a new configuration, enter the configuration name.
3. Enter the project name and Main class.
4. Click the Arguments tab and enter the URL of the web service gateway.
5. Click **Apply** and then **Run**.

Sample output

Attribute	egonode-a	egonode-b	egonode-c	egonode-d
Type	host	host	host	host
State	ok	ok	ok	ok
status	0	0	0	0
type	LINUX86	LINUX86	LINUX86	LINUX86
model	PC300	PC300	PC300	PC300
ncpu	1	1	1	1
cpuf	8.300000	8.300000	8.300000	8.300000
mem	122.000000	176.000000	170.000000	172.000000
swp	544.500000	473.750000	645.000000	467.750000
maxmem	250	250	250	250
maxswp	556	556	729	564
tmp	9984.000000	10032.000000	15968.000000	10112.000000
ut	0.003643	0.001510	0.001562	0.001888
it	31712.000000	35616.000000	35616.000000	35616.000000
io	47.531250	17.093750	13.750000	14.453125
pg	3.150391	1.052734	0.914551	0.962402
r1m	0.000000	0.000000	0.000000	0.000000
r15s	1.767578	0.180664	1.674805	0.871094
r15m	0.000000	0.000000	0.000000	0.000000
ls	1.000000	1.000000	1.000000	1.000000
slot	2	2	2	2
freeslot	2	2	2	2
RESOURCES	(linux)	(linux)	(linux)	(linux)

Tutorial 2: Register, Locate, and Unregister a Client

This tutorial describes how to register and unregister the client with the EGO Web Service.

Using this tutorial, you will …

- Open a connection to the EGO Web Service endpoint
- Retrieve and print out cluster info
- Retrieve and print out resource info
- Register the client with Platform EGO and print out the registration response
- Locate the client and print out the client info
- Unregister the client

Step 1: Import class references

Import the necessary classes and interfaces that are required by the client to invoke the Web Service.

Step 2: Retrieve cluster and Resource information

The Sample 2 EGOclient class is a subclass of the Sample 1 EGOclient class. Refer to Tutorial 1: Step 2: Retrieve cluster information on page 115 and Step 3: Retrieve resource information on page 115.

Step 3: Register the client

Registration is required in order for clients to be able to request allocations and start activities. The client must also be registered in order to receive notifications for events related to the client's allocations, assigned resources, and activities.

Check if the client can connect to the RegistrationPortType endpoint. If successful, create the RegisterRequestDocument (requestDoc) and RegisterRequest (request) objects. The RegisterRequest object has three member variables: the clientID or name, which must be unique for each client connected to the cluster; a client description; and a URI for the notification endpoint. This endpoint implements the EGO notification WSDL interface. Platform EGO will then send notifications to the client using this endpoint.

Create the security document. The EGO WSDLs and Web Service gateway support Web Service Security (WSSE specification). This means that different types of security information can be passed in the header of SOAP messages sent by the clients. The samples use the simplest form, i.e., username and password authentication. The wrappers generated for Java have signatures that provide for

multiple types of security information to be included. In this case, we are using just one security document, which is passed to the register method along with requestDoc and the logonDoc security document.

The response to a registration request has the following sub-elements:

ClientName - the client name that has been assigned to this registration (this will be the same as the optional requested ClientName if the registration is accepted).

AllocationInfo - zero or more elements describing the allocations that have been made previously by this client.

ActivityInfo - zero or more elements describing the activities that have be created previously by this client.

Print out the client name, and the allocation and activity info from the registration response.

```
public void register(String clientId, String clientDescription, String uri)
throws Exception, RemoteException {
this.clientId = clientId;

//the default constructor would use the right endpoint
if (regPort == null) {
try {
  regPort = new RegistrationPortTypeStub(null, targetURL);
} catch (Exception e) {
e.printStackTrace();
  System.err.println("Could not find RegistrationPort...");
  return;
}
}
// create the RegisterRequest Document
RegisterRequestDocument requestDoc = RegisterRequestDocument.Factory
.newInstance();
RegisterRequest request = requestDoc.addNewRegisterRequest();
request.setClientName(clientId);
request.setClientDescription(clientDescription);

URL ntfURL = new URL(uri);
ntfPortNumber = ntfURL.getPort();

        //TODO Hack until we get Addressing/SoapHeaders to work
String uriHack = uri;
request.setNotificationEndpoint(uriHack);
        request.setOptionArray(new String[]{});
// create the RegisterResponse Document
RegisterResponseDocument responseDoc;
// security documents
SecurityDocument sdoc1 = SecurityDocument.Factory.newInstance();
UsernameTokenType usernameTType = getUsernameTokenType(username, password);
        SecurityHeaderType sech1 =
 SecurityHeaderType.Factory.parse(usernameTType.getDomNode());
        sdoc1.setSecurity(sech1);
    sdoc1.dump();
logonDoc = sdoc1;
sdoc1.dump();
requestDoc.dump();
responseDoc = regPort.Register(requestDoc, logonDoc);
RegisterResponse response = responseDoc.getRegisterResponse();
print(response);

handleRecovery(response);
}
```

Step 4: Locate the client

The Locate operation is used to retrieve information about clients registered with Platform EGO.

Check if the client can connect to the RegistrationPortType endpoint. If successful, create the LocateRequestDocument (lreqDoc) and LocateRequest (lreq) objects. Set the client ID for the LocateRequest object. If no client ID is given, then all registered clients will be returned.

Create the security documents. The wrappers generated for Java have signatures that provide for multiple types of security information to be included. As in the case of client registration, we are using just one security document. Since the Locate method signature has more than one document defined, we pass two empty security documents (sdoc2 and sdoc3) along with lreqDoc and the logonDoc security document.

The response to a locate client request contains zero or more elements of client info.

Print out the client info from the locate client response.

```
public void locateClients(String clientId) {
if(this.clientId != clientId) {
System.err.println("This clientId was not registered through this Object...:"
 + clientId);
}
if (regPort == null) {
System.err.println("No RegistrationPort exists...");
return;
}
try {
LocateRequestDocument lreqDoc = LocateRequestDocument.Factory
.newInstance();
LocateRequest lreq = lreqDoc.addNewLocateRequest();
//TODO does empty string mean allClients?
lreq.setClientName(clientId);
SecurityDocument sdoc2 = SecurityDocument.Factory.newInstance();
SecurityHeaderType sec2 = sdoc2.addNewSecurity();
SecurityDocument sdoc3 = SecurityDocument.Factory.newInstance();
SecurityHeaderType sec3 = sdoc3.addNewSecurity();

LocateResponseDocument lresDoc = regPort.Locate(lreqDoc,
logonDoc, sdoc2, sdoc3);
LocateResponse lres = lresDoc.getLocateResponse();
print(lres);
} catch (RemoteException rex) {
rex.printStackTrace();
return;
}
}
```

Step 5: Unregister the client

The Unregister operation is used to terminate an existing EGO registration, which has the effect of terminating all activities started by this client, and releasing all current allocations.

Check if the client can connect to the RegistrationPortType endpoint.

Create the UnregisterRequestDocument (requestDoc) and UnregisterRequest (request) objects. Set the client ID for the UnregisterRequest object.

Create the UnregisterResponseDocument (responseDoc) object. Call the Unregister method and pass the requestDoc and logonDoc objects as input arguments.

```
public void unregister(String clientId) {
if(this.clientId != clientId) {
System.err.println("This clientId was not registered through this Object...:"
 + clientId);
}
if (regPort == null) {
System.err.println("No RegistrationPort exists...");
return;
}

try {
// create the RegisterRequest Document
UnregisterRequestDocument requestDoc = UnregisterRequestDocument.Factory
.newInstance();
UnregisterRequest request = requestDoc.addNewUnregisterRequest();
request.setClientName(clientId);

// create the RegisterResponse Document
UnregisterResponseDocument responseDoc = UnregisterResponseDocument.Factory
.newInstance();

responseDoc = regPort.Unregister(requestDoc, logonDoc);
UnregisterResponse response = responseDoc.getUnregisterResponse();
response.dump();

} catch (RemoteException rex) {
rex.printStackTrace();
}
}
```

Step 6: Print out the information

The following methods iterate through the various arrays to collect and format the client, activity, and allocation information before printing it out.

```
public void print(RegisterResponse response) {
String name = response.getClientName();
AllocationInfo [] alocinfos   = response.getAllocationInfoArray();
ActivityInfo   [] actinfos = response.getActivityInfoArray();

System.out.println("RegisterResponse: " + name);
print(alocinfos); System.out.println();
print(actinfos);  System.out.println();
}

public void print(LocateResponse lres) {
ClientInfo[] cinfos = lres.getClientInfoArray();
for (int i = 0; i < cinfos.length; i++) {
ClientInfo cinfo = cinfos[i];
print(cinfo);
}
}

public void print(AllocationInfo [] alocinfos)
{
for(int i=0; i<alocinfos.length; i++) {
AllocationInfo ainfo = alocinfos[i];
String id = ainfo.getAllocationID();
String consumer = ainfo.getConsumerName();
AllocationStatus.Enum status = ainfo.getAllocationStatus();
AllocationSpecification aspec = ainfo.getAllocationSpecification();
Resource [] ress = ainfo.getResourceArray();
System.out.printf("%-12s\t%-12s\t%-12s\t", id, consumer, status);
//TODO print aloc spec
for(int j=0; j<ress.length; j++) {
System.out.printf("%-12s\t", ress[j].getResourceName());
}
System.out.println();
}
}
```

```
public void print(ActivityInfo [] actInfos)
{
for (int i=0; i<actInfos.length; i++) {
ActivityInfo ainfo = actInfos[i];
            String id = ainfo.getActivityID();
ActivityState.Enum state = ainfo.getActivityState();
int exitCode = ainfo.getExitStatus().intValue();
String [] exitReason = ainfo.getExitReasonArray();
Calendar start = ainfo.getStartTime();
Calendar end = ainfo.getEndTime();
ActivityResourceUsage rusage = ainfo.getActivityResourceUsage();
ainfo.getResourceNameArray();
System.out.printf("%-12s\t%-12s\t", id, state.toString());
if(state == ActivityState.FINISH) {
System.out.printf("%-12d\t", exitCode);
for(int j=0; j<exitReason.length; j++) {
  System.out.printf("%-12s\t", exitReason[j]);
}
} else {
System.out.printf("%-12d\t%-12d\t%-12d", start, rusage.getUtime(),
rusage.getStime());
}
System.out.println();
}
}
```

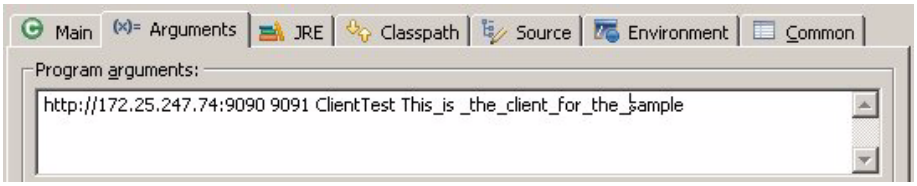
Run the client application

1. Select **Run > Run**.

   The Run dialog appears.
2. In the Configurations list, either select a Java Application or click **New** for a new configuration.

   For a new configuration, enter the configuration name.
3. Enter the project name and Main class.
4. Click the Arguments tab and enter the following arguments in the given order:
   1. URL of the web service gateway
   2. Port number (string) for the notification interface
   3. Client ID (string)
   4. Client description (string).

NOTE: Arguments must be separated by a space.



5. Click **Apply** and then **Run**.

Sample output

Attribute	egonode-a	egonode-b	egonode-c	egonode-d
Type	host	host	host	host
State	ok	ok	ok	ok
status	0	0	0	0
type	LINUX86	LINUX86	LINUX86	LINUX86
model	PC300	PC300	PC300	PC300
ncpu	1	1	1	1
cpuf	8.300000	8.300000	8.300000	8.300000
mem	122.000000	176.000000	170.000000	172.000000
swp	544.500000	473.750000	645.000000	467.750000
maxmem	250	250	250	250
maxswp	556	556	729	564
tmp	9984.000000	10032.000000	15968.000000	10112.000000
ut	0.003643	0.001510	0.001562	0.001888
it	31712.000000	35616.000000	35616.000000	35616.000000
io	47.531250	17.093750	13.750000	14.453125
pg	3.150391	1.052734	0.914551	0.962402
r1m	0.000000	0.000000	0.000000	0.000000
r15s	1.767578	0.180664	1.674805	0.871094
r15m	0.000000	0.000000	0.000000	0.000000
ls	1.000000	1.000000	1.000000	1.000000
slot	2	2	2	2
freeslot	2	2	2	2
RESOURCES	(linux)	(linux)	(linux)	(linux)

```
  ROOT (USER) *:R:<cur>[0] (DocumentXobj)
    ELEM Security@http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd (USER) (El
      ELEM wsse:UsernameToken@http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xs
        ATTR xmlns:wsse@http://www.w3.org/2000/xmlns/ Value( "http://docs.oasis-open.org/wss/2004/..." ) (Attr
        ATTR xmlns:wsu@http://www.w3.org/2000/xmlns/ Value( "http://docs.oasis-open.org/wss/2004/..." ) (AttrX
        ATTR wsu:Id@http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd Value(
        ELEM wsse:Username@http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd V
        ELEM wsse:Password@http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd (
          ATTR Type Value( "http://docs.oasis-open.org/wss/2004/..." ) After( "egoadmin" ) (AttrXobj)

  ROOT (USER) *:R:<cur>[0] (DocumentXobj)
    ELEM Security@http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd (USER) (El
      ELEM wsse:UsernameToken@http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xs
        ATTR xmlns:wsse@http://www.w3.org/2000/xmlns/ Value( "http://docs.oasis-open.org/wss/2004/..." ) (Attr
        ATTR xmlns:wsu@http://www.w3.org/2000/xmlns/ Value( "http://docs.oasis-open.org/wss/2004/..." ) (AttrX
        ATTR wsu:Id@http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd Value(
        ELEM wsse:Username@http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd V
        ELEM wsse:Password@http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd (
          ATTR Type Value( "http://docs.oasis-open.org/wss/2004/..." ) After( "egoadmin" ) (AttrXobj)

  ROOT (USER) *:R:<cur>[0] (DocumentXobj)
    ELEM RegisterRequest@http://www.platform.com/ego/2005/05/wsdl/registration (USER) (ElementXobj)
      ELEM ClientName@http://www.platform.com/ego/2005/05/schema Value( "client_name" ) (USER) (ElementXobj)
      ELEM ClientDescription@http://www.platform.com/ego/2005/05/schema Value( "client_description" ) (USER) (
      ELEM NotificationEndpoint@http://www.platform.com/ego/2005/05/schema Value( "http://dchabot:9091/axis/se

RegisterResponse: client_name


client_name     client_description      39260@172.25.244.143
  ROOT (USER) <mark>[0] (DocumentXobj)
    ELEM reg:UnregisterResponse@http://www.platform.com/ego/2005/05/wsdl/registration (USER) *:R:<cur>[0] (Ele
```

Tutorial 3: Request a Resource Allocation in a Cluster

This tutorial describes how to create a registered EGO client that requests a resource allocation in a cluster and starts an activity on it.

Using this tutorial, you will …

- Open a connection to the EGO Web Service endpoint
- Retrieve and print out cluster info
- Retrieve and print out resource info
- Register the client with Platform EGO and print out the registration response
- Request a resource allocation
- Check for notification of resource allocation
- Create an activity that will run on the requested resource
- Locate the client and print out the client info
- Unregister the client

Step 1: Import class references

Import the necessary classes and interfaces that are required by the client to invoke the Web Service.

Step 2: Retrieve cluster and Resource information

Refer to Tutorial1: Step 2: Retrieve cluster information on page 115 and Step 3: Retrieve resource information on page 115.

Step 3: Register the client

Refer to Tutorial 2: Step 3: Register the client on page 120.

Step 4: Make a resource allocation request

The RequestAllocation operation is used by an EGO client to make a request for resources.

Call the allocate method and pass the number of resources requested as the input argument. In this case, the number of resources is 1.

Check if the client can connect to the AllocationPortTypeStub endpoint. If successful, create a resource specification (resourceSpecifications). The resource specification provides a means for specifying a request for consumable resources and any constraints on resource selection.

Create an allocation specification (alocSpec). This allocation specification describes a request to Platform EGO for an allocation of resources. The only required variables for the allocation specification are the consumer name to charge this allocation to, and the resource specification (resourceSpecifications) of what is being requested.

Create an allocation request object (alocReq) and link it with the client name and allocation specification (alocSpec).

Create the security document. The wrappers generated for Java have signatures that provide for multiple types of security information to be included. As in the case of client registration, we are using just one security document (logonDoc). Pass it along with along with alocReqDoc to the RequestAllocation method.

Create an allocated resource object (alocResources). The AllocatedResources class models the resources that the application has requested and tracks them as it obtains and uses them. The constructor takes the allocation ID and the number of resources requested.

The allocation ID is then added to a list of allocation IDs. The allocidToResourcesMap object is used to map allocation ID to the AllocatedResources class that tracks it. So given an allocation ID, one can find out the details about it from the AllocatedResources instance. The alocIds.add and allocidToResourceMap.put methods update the list of IDs and the map, respectively.

```
public void allocate(int numberOfResources)
{
if(allocPort == null) {
  try {
    allocPort = new AllocationPortTypeStub(null, targetURL);
  } catch(Exception e) {
e.printStackTrace();
return;
  }}
resourceSpecifications = createResourceSpecification(numberOfResources);
AllocationSpecificationDocument alocSpecDoc =
 AllocationSpecificationDocument.Factory.newInstance();
AllocationSpecification alocSpec =
 alocSpecDoc.addNewAllocationSpecification();
alocSpec.setAllocationName("Sample3Allocation");
alocSpec.setConsumerName("SampleApplications/EclipseSamples");
alocSpec.setResourceSpecificationArray(resourceSpecifications);
RequestAllocationRequestDocument alocReqDoc =
 RequestAllocationRequestDocument.Factory.newInstance();
RequestAllocationRequest alocReq =
 alocReqDoc.addNewRequestAllocationRequest();
alocReq.setClientName(clientId);
alocReq.setAllocationSpecification(alocSpec);
RequestAllocationResponseDocument alocResDoc;
RequestAllocationResponse alocRes;
try {
alocResDoc = allocPort.RequestAllocation(alocReqDoc, logonDoc);
          alocRes = alocResDoc.getRequestAllocationResponse();
  AllocatedResources alocResources = new
 AllocatedResources(alocRes.getAllocationID(),
numberOfResources);
          alocIds.add(alocRes.getAllocationID());
          allocidToResourcesMap.put(alocRes.getAllocationID(), alocResources);
}catch (RemoteException rex) {
System.err.println("Failed to Allocate");
rex.printStackTrace();
return;
}}
```

```
protected ResourceSpecification[] createResourceSpecification(int numberOfResources) {
ResourceSpecification [] resSpecs = new ResourceSpecification [1];
   ResourceSpecification resSpec =
 ResourceSpecification.Factory.newInstance();
   resSpec.setMaxResources(new
 BigInteger(Integer.toString(numberOfResources)));
   resSpec.setMinResources(new BigInteger("1"));
   resSpec.setResourceGroupName("ComputeHosts");
   resSpec.setResourceRequirement("LINUX86"); //  NTX86
   resSpecs[0] = resSpec;
return resSpecs;
}
```

Step 5: Check the allocation status

This sample implements a notification service that runs as a separate thread in the background and listens continuously for allocation notification events. When the checkAddResourceNotification() method is called, the main thread "waits" on the Notification.notification object, blocking the thread. When a notification arrives, the thread that handles the notification, "notifies" the Notification.notification object. This wakes up the main thread that was blocked. A while loop is used so that if the thread is interrupted, the execution goes back to waiting. The synchronized keyword ensures that the code block is executed by only one thread at a time.

Once the resource is allocated, print out the resource request details and add them to the AllocatedResources instance.

```
public int checkAddResourceNotification()
{
// wait until we hear about allocation
boolean done = false;
AddResourceRequest arReq = null;
synchronized(Notification.notification) {
            while(!done){
  try {
               Notification.notification.wait();
          arReq = Notification.notification.getAddResourceRequest();
          done = match(arReq); // true
  } catch (InterruptedException ie) {
ie.printStackTrace();
  }
            }
}

// received soap call at notification port
print(arReq);

//TODO Handle multiple AllocationIds
// For multiple resources requested in a single allocation
// the addResource may come individually at different times
// Append to the already recieved ones.
// incrementally use the ones as we obtain them.
        AllocatedResources allocatedResources =
 allocidToResourcesMap.get(arReq.getAllocationID());
        allocatedResources.add(arReq.getResourceArray());

// sendResponse();
        // handled by the NotificationReceiver

        return arReq.getResourceArray().length;
}
```

Step 6: Create and start an activity on a resource

The StartActivity operation is used by an EGO client to request the execution of an activity on a resource. Usually resources are allocated to the client via an EGO allocation request prior to starting an activity . Thus the AllocationID within the request message is the one returned from a previous RequestAllocation call.

Call the createActivity method and pass the allocation ID and the number of activities as input arguments. In this case, the number of activities is 1.

Check if the client can connect to the ActivityPortTypeStub endpoint. If successful, create an activity specification (actSpec). The activity specification describes the execution parameters for an activity, such as the command. In this case, the sleep command is used to simulate activity on the resource.

Create an activity start request object (sactReq). The sactReq object links the activity specification with the allocation ID and client ID.

Call the StartActivity() method with the start activity request document (sactReqDoc) and the logon document (logonDoc) as input arguments. The response message for StartActivity (sactRes) contains the ActivityID assigned by Platform EGO to this activity. The ActivityID can then be used in other operations to manage and query the state of the activity.

Create an activity resources object (activityResources). The ActivityResources class models the resources that the activity is consuming and tracks them as they are used. The constructor takes the activity ID, allocation ID, and resource name array.

The activityidToResourcesMap object is used to map activity ID to the ActivityResources class that tracks it. So given an activity ID, you can find out the details about it from the ActivityResources instance. The activityidToResourceMap.put method updates the list of IDs and the map.

Program execution returns to the main method and waits until the activity, which is sleep for 120 seconds, is finished so that various notifications about the activity can be displayed

```
public void createActivity(String allocationId, int numActivities)
{
this.numActivities = numActivities;

if (activityPort == null) {
try {

  activityPort = new ActivityPortTypeStub(null, targetURL);
} catch(Exception e) {
e.printStackTrace();
return;
}
}

AllocatedResources allocatedResources =
 allocidToResourcesMap.get(allocationId);
String [] resourceNames = allocatedResources.consume();

EnvironmentVariable [] env = new EnvironmentVariable [] {};
Rlimit [] rlimits = new Rlimit[] {};
ActivitySpecificationDocument actSpecDoc =
 ActivitySpecificationDocument.Factory.newInstance();
ActivitySpecification actSpec = actSpecDoc.addNewActivitySpecification();
actSpec.setActivityName("Sample3Job");
actSpec.setCommand("/bin/sleep 120");
actSpec.setEnvironmentVariableArray(env);actSpec.setExecutionUser("lsfadmin");
actSpec.setWorkingDirectory("/tmp");actSpec.setUmask("0777");
actSpec.setRlimitArray(rlimits);

StartActivityRequestDocument sactReqDoc =
 StartActivityRequestDocument.Factory.newInstance();
StartActivityRequest sactReq = sactReqDoc.addNewStartActivityRequest();
sactReq.setActivitySpecification(actSpec);
sactReq.setAllocationID(allocationId);
sactReq.setClientName(clientId);

sactReq.setResourceNameArray(resourceNames);
sactReq.setOptionArray(null);
SecurityDocument sdoc1 = SecurityDocument.Factory.newInstance();
SecurityHeaderType sec1 = sdoc1.addNewSecurity();

StartActivityResponseDocument sactResDoc;
StartActivityResponse sactRes;
```

```
for (int i=0; i<numActivities; i++) {
  try {
    sactResDoc = activityPort.StartActivity(sactReqDoc, logonDoc /* sdoc1 */);
    sactRes = sactResDoc.getStartActivityResponse();

            //TODO check if activity started correctly
    activities.add(sactRes.getActivityID());

    // add to the map
    ActivityResources activityResources = new ActivityResources(
    sactRes.getActivityID(), allocationId, resourceNames);
    activityidToResourcesMap.put(sactRes.getActivityID(), activityResources);

    // monitorActivity(activities);
  } catch(RemoteException rex) {
rex.printStackTrace();
//TODO Release Resource
releaseResources(allocationId);
return;
  }
}
}
```

Step 7: Locate the client

Refer to Tutorial 2: Step 4: Locate the client on page 122.

Step 8: Unregister the client

Refer to Tutorial 2: Step 5: Unregister the client on page 123.

Run the client application

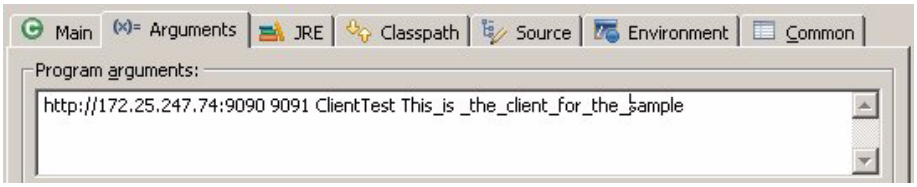
1. Select **Run > Run**.

   The Run dialog appears.
2. In the Configurations list, either select a Java Application or click **New** for a new configuration.

   For a new configuration, enter the configuration name.
3. Enter the project name and Main class.

4 Click the Arguments tab and enter the following arguments in the given order:
   1 URL of the web service gateway
   2 Port number (string) for the notification interface
   3 Client ID (string)
   4 Client description (string).

NOTE: Arguments must be separated by a space.



5 Click **Apply** and then **Run**.

Sample output

```
  ROOT (USER) *:R:<cur>[0] (DocumentXobj)
    ELEM Security@http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd (USER) (El
      ELEM wsse:UsernameToken@http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xs
        ATTR xmlns:wsse@http://www.w3.org/2000/xmlns/ Value( "http://docs.oasis-open.org/wss/2004/..." ) (Attr
        ATTR xmlns:wsu@http://www.w3.org/2000/xmlns/ Value( "http://docs.oasis-open.org/wss/2004/..." ) (AttrX
        ATTR wsu:Id@http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd Value(
        ELEM wsse:Username@http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd V
        ELEM wsse:Password@http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd (
          ATTR Type Value( "http://docs.oasis-open.org/wss/2004/..." ) After( "egoadmin" ) (AttrXobj)

  ROOT (USER) *:R:<cur>[0] (DocumentXobj)
    ELEM Security@http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd (USER) (El
      ELEM wsse:UsernameToken@http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xs
        ATTR xmlns:wsse@http://www.w3.org/2000/xmlns/ Value( "http://docs.oasis-open.org/wss/2004/..." ) (Attr
        ATTR xmlns:wsu@http://www.w3.org/2000/xmlns/ Value( "http://docs.oasis-open.org/wss/2004/..." ) (AttrX
        ATTR wsu:Id@http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd Value(
        ELEM wsse:Username@http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd V
        ELEM wsse:Password@http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd (
          ATTR Type Value( "http://docs.oasis-open.org/wss/2004/..." ) After( "egoadmin" ) (AttrXobj)

  ROOT (USER) *:R:<cur>[0] (DocumentXobj)
    ELEM RegisterRequest@http://www.platform.com/ego/2005/05/wsdl/registration (USER) (ElementXobj)
      ELEM ClientName@http://www.platform.com/ego/2005/05/schema Value( "client_name" ) (USER) (ElementXobj)
      ELEM ClientDescription@http://www.platform.com/ego/2005/05/schema Value( "client_description" ) (USER) (
      ELEM NotificationEndpoint@http://www.platform.com/ego/2005/05/schema Value( "http://dchabot:9091/axis/se

RegisterResponse: client_name


client_name     client_description      39260@172.25.244.143
  ROOT (USER) <mark>[0] (DocumentXobj)
    ELEM reg:UnregisterResponse@http://www.platform.com/ego/2005/05/wsdl/registration (USER) *:R:<cur>[0] (Ele


client_name                                     9                       archtsmst
13              start           9               client_name
13              run             9               client_name
13              finish          9               client_name
client_name     client_description      39290@172.25.244.143
  ROOT (USER) <mark>[0] (DocumentXobj)
    ELEM reg:UnregisterResponse@http://www.platform.com/ego/2005/05/wsdl/registration (USER) *:R:<c
```

Tutorial 4: Monitor an Activity on a Resource

This tutorial describes how to create a registered EGO client that monitors an activity on a resource.

Using this tutorial, you will …

- Open a connection to the EGO Web Service endpoint
- Retrieve and print out cluster info
- Retrieve and print out resource info
- Register the client with Platform EGO and print out the registration response
- Request a resource allocation
- Check for notification of resource allocation
- Create an activity that will run on the requested resource
- Calculate the activity load on the resource and print it out
- Monitor the activity
- Locate the client and print out the client info
- Unregister the client

Step 1: Import class references

Import the necessary classes and interfaces that are required by the client to invoke the Web Service.

Step 2: Retrieve cluster and Resource information

Refer to Tutorial 1: Step 2: Retrieve cluster information on page 115 and Step 3: Retrieve resource information on page 115.

Step 3: Register the client

Refer to Tutorial 2: Step 3: Register the client on page 120.

Step 4: Make a resource allocation request

Refer to Tutorial 3: Step 4: Make a resource allocation request on page 128.

Step 5: Check for notification of resource allocation

Refer to Tutorial 3: Step 5: Check the allocation status on page 131.

Step 6: Create an activity that will run on a requested resource

Refer to Tutorial 3: Step 6: Create and start an activity on a resource on page 131.

Step 7: Calculate the activity load on the resource

In order to determine if a resource is too busy to receive jobs, a load index value is calculated and compared to a corresponding load threshold parameter. The following code retrieves and processes the load index value for r1m, the 1-minute CPU run queue length.

The main method contains a While loop that retrieves the average resource load every 30 seconds. This update interval is implemented by a sleep method prior to collecting the information. This cycle is repeated four times and then the loop exits.

Pass the activity ID to the getActivityResourceLoad() method. This method collects the resource names associated with the activity ID. Pass the resource names array to the calculateActivityLoad() method, which cycles through the array and retrieves the load attribute index associated with each resource. The getLoadAttributeIndex() method cycles through the list of resource attributes looking for the r1m load index. (The r1m load index represents the average number of processes ready to use the CPU during a one-minute interval.) This method returns the attribute array index corresponding to load index r1m. The value for r1m is retrieved and converted to a double data type. This value is stored in a

variable (total), which is used as an accumulator. The total sum is then divided by the number of resources to yield the average load index value of r1m. The activity ID and the average load index value are printed out.

```
public double getActivityResourceLoad(String activityId)
{
double load = 0;

// for all the resources in this activity compute the average load
ActivityResources activityResources =
 activityidToResourcesMap.get(activityId);
        String [] resourceNames = activityResources.getResources();
Resource [] resources = this.resourceInfo(resourceNames);
load = calculateActivityLoad(resources);
    return load;
}
public double calculateActivityLoad(Resource [] resources)
{
double total=0;
for(int i=0; i<resources.length; i++) {
Resource resource = resources[i];
    int index = getLoadAttributeIndex(resource);
    total += getDoubleValue(resource.getAttributeArray(index));
}
return total/resources.length;
}
public int getLoadAttributeIndex(Resource res)
{
int rv = -1;
Attribute [] attrs = res.getAttributeArray();
for(int i=0; i<attrs.length; i++) {
Attribute attr = attrs[i];
if(attr.getName().equals("r1m")) {
return i;
}
}
return rv;
}
```

Step 8: Monitor the activity

Activity information describes the state of an activity within Platform EGO. There is both EGO meta data described (activity ID, activity state, allocation ID, consumer name, and activity specification) as well as run time state (start time, end time, resource name, exit status, exit reason, resource usage, and activity resource usage).

Pass the activity ID to the monitorActivity() method. Check if the activity ID is valid and that the client can connect to the MonitoringPortType endpoint. If successful, call the getActivityInfos() method and pass the activity ID as an argument.

Use the getActivityInfos() method to create an activity information request object (aiReq). Call the ActivityInfo() method with the activity info request document (aiReqDoc) and the logon document (logonDoc) as input arguments. The response

message for ActivityInfo (aiRes) contains the activity state information for the associated activity ID, as described above. Print out the activity and resource load information, and return control to the monitorActivity() method.

Create a For loop that checks the state information for each activity. Once an activity has been accepted and assigned an activity ID, it can be in one of the following states: null, start, run, suspend, finish, and unknown. If the activity state is "run", the loop will be repeated after one minute. Print out the activity information.

```
public void monitorActivity(String[] activityIds)
{
if (activityIds == null) {
return;
}

// use ActivityInfo from the MonitoringPort
if (monitor == null) {
  try {
monitor = new MonitoringPortTypeStub(null, targetURL);
   } catch (Exception ex) {
ex.printStackTrace();
return;
  }
}

boolean done = false;
while(!done) {
// as long as the activity is running
     ActivityInfo [] infos = getActivityInfos(activityIds);
  done = true;
  for (int i=0; i<infos.length; i++) {
  ActivityInfo info = infos[i];
  ActivityState.Enum state = info.getActivityState();
  if(state == ActivityState.RUN) {
  done = false;
  }
  print(info);
  }

  // wait for a minute
  try {
    Thread.sleep(60000);
  } catch(InterruptedException ie) {
  ie.printStackTrace();
  }
}
}
```

```
public ActivityInfo [] getActivityInfos(String [] activityIds)
{
if (activityIds == null) {
return null;
}
// use ActivityInfo from the MonitoringPort
if (monitor == null) {
try {
monitor = new MonitoringPortTypeStub(null, targetURL);
} catch (Exception ex) {
ex.printStackTrace();
return null;
}
}
// as long as the activity is running
ActivityInfoRequestDocument aiReqDoc = ActivityInfoRequestDocument.Factory
.newInstance();
ActivityInfoRequest aiReq = aiReqDoc.addNewActivityInfoRequest();
aiReq.setActivityIDArray(activityIds);
aiReq.setOptionArray(null);

ActivityInfoResponseDocument aiResDoc;
ActivityInfoResponse aiRes;
try {
aiResDoc = monitor.ActivityInfo(aiReqDoc, logonDoc);
aiRes = aiResDoc.getActivityInfoResponse();
print(aiRes.getActivityInfoArray());
} catch (RemoteException rex) {
rex.printStackTrace();
return null;
}

ActivityInfo[] infos = aiRes.getActivityInfoArray();
return infos;
}
```

Step 9: Locate the client

Refer to Tutorial 2: Step 4: Locate the client on page 122.

Step 10: Unregister the client

Refer to Tutorial 2: Step 5: Unregister the client on page 123.

Run the client application

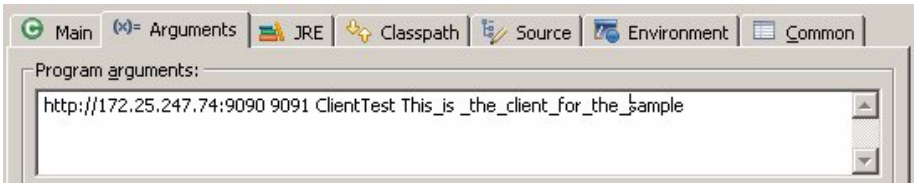
1. Select **Run > Run**.

   The Run dialog appears.
2. In the Configurations list, either select a Java Application or click **New** for a new configuration.

   For a new configuration, enter the configuration name.
3. Enter the project name and Main class.

4 Click the Arguments tab and enter the following arguments in the given order:
   1 URL of the web service gateway
   2 Port number (string) for the notification interface
   3 Client ID (string)
   4 Client description (string).

NOTE: Arguments must be separated by a space.



5 Click **Apply** and then **Run**.

Sample output

```
client_name                                    9                    archtsmst
13              start           9              client_name
13              run             9              client_name
13              finish          9              client_name
client_name     client_description      39290@172.25.244.143
  ROOT (USER) <mark>[0] (DocumentXobj)
    ELEM reg:UnregisterResponse@http://www.platform.com/ego/2005/05/wsdl/registration (USER) *:R:<c
```

```
Attribute               archtsmst
Type                    host
State                   ok
status                  0
type                    LINUX86
model                   PC450
ncpu                    1
cpuf                    13.200000
mem                     228.000000
swp                     711.656250
maxmem                  377
maxswp                  823
tmp                     13205.007812
ut                      0.002510
it                      2.716667
io                      946.916870
pg                      63.515724
r1m                     0.010000
r15s                    0.107358
r15m                    0.000000
ls                      1.000000
slot                    3
freeslot                2
RESOURCES               (linux)
Resource Load: 14                       0.01
```

Tutorial 5: Modify Resources Based on Load Information

This tutorial describes how to create a registered EGO client that modifies resources in accordance with changes in resource loading. A minimum of two resources is required.

Using this tutorial, you will …

- Open a connection to the EGO Web Service endpoint
- Retrieve and print out cluster info
- Check that the cluster has enough resources
- Register the client with Platform EGO and print out the registration response
- Request a resource allocation
- Check for notification of resource allocation
- Create an activity that will run on the requested resource
- Check the resource loading
- Modify the resources
- Locate the client and print out the client info
- Unregister the client

Step 1: Import class references

Import the necessary classes and interfaces that are required by the client to invoke the Web Service.

Step 2: Retrieve cluster information

Refer to Tutorial 1: Step 2: Retrieve cluster information on page 115.

Step 3: Check that the cluster has enough resources

Pass null as an argument to the resourceInfo() method to retrieve all resources in the cluster. For this sample, we need at least two resources.

```
Resource [] resources = client.resourceInfo(null);
int num = 0;
int needed = 2;
if(resources != null) {
  num = resources.length;
}
if (num < needed) {
  System.err.println("Cluster Requirement: Should be run on a cluster with at
 least " +

needed + " resources, it has only "+num);
notification.stop();
return;
}
```

Step 4: Register the client

Refer to Tutorial 2: Step 3: Register the client on page 120.

Step 5: Make a resource allocation request

Refer to Tutorial 3: Step 4: Make a resource allocation request on page 128. In this sample, we make an allocation request for one resource.

Step 6: Check for notification of resource allocation

Refer to Tutorial 3: Step 5: Check the allocation status on page 131.

Step 7: Create an activity that will run on a requested resource

Refer to Tutorial 3: Step 6: Create and start an activity on a resource on page 131. In this sample, we create 10 activities on a single resource.

Step 8: Check the resource loading

Wait 40 seconds and pass the activity ID to the getActivityResourceLoad() method. This method collects the resource names associated with the activity ID. Pass the resource names array to the calculateAcvtivityLoad() method, which cycles through the array and retrieves the load attribute index associated with each resource. The getLoadAttributeIndex() method cycles through the list of resource attributes looking for the r1m load index. (The r1m load index represents the average number of processes ready to use the CPU during a one-minute interval.) This method returns the attribute array index corresponding to load index r1m. The value for r1m is retrieved and converted to a double data type. This value is

stored in a variable (total), which is used as an accumulator. The total sum is then divided by the number of resources to yield the average load index value of r1m. The activity ID and the average load index value are printed out.

Step 9: Modify the resources

Pass the activity ID, delta, loadThreshold, and hysteresis variables as arguments to the modifyActivity() method. The delta value represents the number of resources to be added or released based on the load. However if the load is within the hysteresis range, no change is made; this is to prevent oscillatory behaviour.

Call the getActivityResourceLoad() method to determine the average load on resources; see description for the getActivityResourceLoad() method in Step 8: Check the resource loading on page 144. Retrieve the allocation ID from the ActivityResources object (activityResources).

Check if the average load exceeds the load threshold plus the hysteresis value. If the average load exceeds the threshold, call the modifyResources() method to add another resource to the activity allocation; if it doesn't, call the modifyResources() method to drop a resource from the activity allocation.

```
public void modifyActivity(String activityId, int delta, double loadThreshold, double
hysterisys)
{
// for any resource that this activity is running on
        // find load, how do you define activity load? on a specefic resource
double load = getActivityResourceLoad(activityId);

        ActivityResources activityResources =
 activityidToResourcesMap.get(activityId);
        String allocationId = activityResources.getAllocationId();

// if greater than loadThreshold + hysterisys/2, release specific resources,
 blklist?
        if(load > (loadThreshold + hysterisys/2)) {
        // load on the resources is high, need to add resources to activity
 allocation
        modifyResources(allocationId, delta, true);
        }

// if less    than loadThreshold - hysterisis/2, add resources
        if(load < (loadThreshold + hysterisys/2)) {
        // resources are lightly loaded, can drop resources from allocation
        modifyResources(allocationId, delta, false);
        }
}
```

In the modifyResources() method, check if the client can connect to the AllocationPortType endpoint. If successful, create a resource specification (resSpec). If a resource needs to be added to the allocation, set the resSpec object so that the minimum and maximum resources is increased by one; otherwise, decrease the minimum and maximum resources by one.

Create an allocation specification (alocSpec). This allocation specification describes a request to Platform EGO for an allocation of resources. The only required variables for the allocation specification are the consumer name to charge this allocation to, and the resource specification (resSpec) of what is being requested.

Create a modify allocation request object (malocReq) and link it with the client name, allocation ID, and allocation specification (alocSpec). Create the security document and the modify allocation response (malocRes).

Call the ModifyAllocation() method with the modify allocation request document (malocReqDoc) and the logon document (logonDoc) as input arguments. This method is used to modify the parameters of an existing allocation. If an error occurs, release the resources associated with the allocation ID by calling the releaseResources() method.

```
public void modifyResources(String allocationId, int delta, boolean add)
{
if (allocPort == null) {
try {
allocPort = new AllocationPortTypeStub();
} catch (Exception e) {
e.printStackTrace();
return;
}
}
// release resources
// another possibility is to release specific resources that are loaded
ResourceSpecification[] resSpecs = new ResourceSpecification[1];
ResourceSpecification resSpec = ResourceSpecification.Factory.newInstance();
if (!add) {
resSpec.setMaxResources(new BigInteger(Integer.toString(-delta)));
resSpec.setMinResources(new BigInteger(Integer.toString(-delta)));
} else {
resSpec.setMaxResources(new BigInteger(Integer.toString(delta)));
resSpec.setMinResources(new BigInteger(Integer.toString(delta)));
}
resSpec.setResourceGroupName("ComputeHosts");
resSpec.setResourceRequirement("LINUX86"); // NTX86
// TODO "EGO_ALLOC_EXCLUSIVE"
resSpecs[0] = resSpec;
AllocationSpecificationDocument alocSpecDoc =
 AllocationSpecificationDocument.Factory.newInstance();
AllocationSpecification alocSpec =
 alocSpecDoc.addNewAllocationSpecification();
alocSpec.setAllocationName("Sample3Allocation");
alocSpec.setConsumerName("/SampleApplications/EclipseSamples");
alocSpec.setResourceSpecificationArray(resSpecs);
// alocSpec.setOptionArray(new String[]{"EGO_ALLOC_EXCLUSIVE"}); //TODO
ModifyAllocationRequestDocument malocReqDoc =
 ModifyAllocationRequestDocument.Factory.newInstance();
ModifyAllocationRequest malocReq =
 malocReqDoc.addNewModifyAllocationRequest();

malocReq.setClientName(clientId);
malocReq.setAllocationID(allocationId);
malocReq.setAllocationSpecification(alocSpec);
malocReq.setOptionArray(new String[] { "EGO_REALLOC_DELTA" });
ModifyAllocationResponseDocument malocResDoc;
ModifyAllocationResponse malocRes;
```

```
boolean release = false;
try {
malocResDoc = allocPort.ModifyAllocation(malocReqDoc, logonDoc);
malocRes = malocResDoc.getModifyAllocationResponse();
// TODO what is the response?
} catch (RemoteException rex) {
// this could be because we cannot shrink below already allocated
// release the resource explicitly in that case or
releaseResources(allocationId, delta);
release = true;

            // another alternative is to cancel allocation
// return;
}
int changes = 0;
if (add) {
  while (changes < delta){
// TODO start a new thread to wait for Resource notification
changes += checkAddResourceNotification();
createActivity(alocIds.get(0), changes);
  }
}

return;
}
```

Create an AllocatedResources object (allocatedResources) and associate it with the allocation ID.

Create a release resource request object (rrreq) and link it with the client name, allocation ID, and the resource entries for all resources. The resource entry consists of one or more elements describing which resources to remove from the allocation. They must be part of the existing allocation to be released.

Create the security documents and the release resource response (rrres)

Call the ReleaseResource() method with the release resource response document (rrreqDoc), the logon document (logonDoc), and two security documents as input arguments. This method is used to release resources from an existing allocation. In this sample, one resource is released since this was the value of delta, which represents the number of resources to add or release.

```
public void releaseResources(String allocationId, int delta)
{
AllocatedResources allocatedResources =
 allocidToResourcesMap.get(allocationId);
Resource [] resources = allocatedResources.getResourcesToRelease(delta);

if (resources == null || resources.length ==0) {
return;
}

ReleaseResourceRequestDocument rrreqDoc =
 ReleaseResourceRequestDocument.Factory.newInstance();
ReleaseResourceRequest rrreq = rrreqDoc.addNewReleaseResourceRequest();
rrreq.setAllocationID(allocationId);
rrreq.setClientName(clientId);
ResourceEntry [] resourceEntries = new ResourceEntry[resources.length];
for(int i=0; i<resources.length; i++){
ResourceEntry entry = ResourceEntry.Factory.newInstance();
entry.setResourceName(resources[i].getResourceName());
entry.setResourceType(resources[i].getResourceType());
resourceEntries[i] = entry;
}
rrreq.setResourceEntryArray(resourceEntries);
rrreq.setOptionArray(new String[]{"EGO_RELEASE_AUTOADJ"});
if(allocPort == null) {
System.err.println("Could not release resource");
return;
}
SecurityDocument sdoc2 = SecurityDocument.Factory.newInstance();
SecurityHeaderType sec2 = sdoc2.addNewSecurity();
SecurityDocument sdoc3 = SecurityDocument.Factory.newInstance();
SecurityHeaderType sec3 = sdoc3.addNewSecurity();

ReleaseResourceResponseDocument rrresDoc;
ReleaseResourceResponse rrres;
try {
  rrresDoc = allocPort.ReleaseResource(rrreqDoc, logonDoc, sdoc2, sdoc3);
  rrres = rrresDoc.getReleaseResourceResponse();
  //TODO what does the response contain?
} catch (RemoteException rex) {
rex.printStackTrace();
return;
}
}
```

Step 10: Locate the client

Refer to Tutorial 2: Step 4: Locate the client on page 122.

Step 11: Unregister the client

Refer to Tutorial 2: Step 5: Unregister the client on page 123.

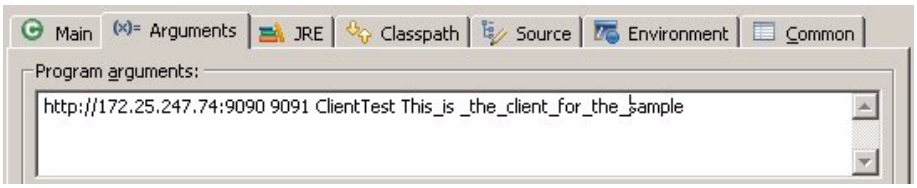
Run the client application

1. Select **Run > Run**.

   The Run dialog appears.
2. In the Configurations list, either select a Java Application or click **New** for a new configuration.

   For a new configuration, enter the configuration name.
3. Enter the project name and Main class.
4. Click the Arguments tab and enter the following arguments in the given order:
   1. URL of the web service gateway
   2. Port number (string) for the notification interface
   3. Client ID (string)
   4. Client description (string).

NOTE: Arguments must be separated by a space.



5. Click **Apply** and then **Run**.

Sample output

```
client_name                                    9                    archtsmst
13              start           9              client_name
13              run             9              client_name
13              finish          9              client_name
client_name     client_description      39290@172.25.244.143
  ROOT (USER) <mark>[0] (DocumentXobj)
    ELEM reg:UnregisterResponse@http://www.platform.com/ego/2005/05/wsdl/registration (USER) *:R:<c
```

```
ActivityId      state           utime           stime           archtsmst
14              finish          null            null            null            Attribute
Type                    host
State                   ok
status                  0
type                    LINUX86
model                   PC450
ncpu                    1
cpuf                    13.200000
mem                     228.000000
swp                     711.656250
maxmem                  377
maxswp                  823
tmp                     13205.007812
ut                      0.004512
it                      8.766666
io                      576.129272
pg                      38.424328
r1m                     0.010000
r15s                    0.293385
r15m                    0.000000
ls                      1.000000
slot                    3
freeslot                3
RESOURCES               (linux)
r1m                     0.010000


client_name                                    11                   archtsmst
15              start           11             client_name
15              run             11             client_name
16              start           11             client_name
```

```
mem                     225.000000
swp                     711.656250
maxmem                  377
maxswp                  823
tmp                     13205.007812
ut                      0.006621
it                      9.783334
io                      1855.542480
pg                      120.819931
r1m                     0.000000
r15s                    0.170442
r15m                    0.000000
ls                      1.000000
slot                    3
freeslot                3
RESOURCES               (linux)
Resource Load: 15                       0.00
24              finish          11              client_name
23              finish          11              client_name
22              finish          11              client_name
21              finish          11              client_name
20              finish          11              client_name
19              finish          11              client_name
18              finish          11              client_name
17              finish          11              client_name
16              finish          11              client_name
15              finish          11              client_name
client_name     client_description      39353@172.25.244.143
  ROOT (USER) <mark>[0] (DocumentXobj)
    ELEM reg:UnregisterResponse@http://www.platform.com/ego/2005/05/wsdl/registration (USER) *:R:<c
```

Tutorial 6: Create an EGO Service

This tutorial describes how to create and run an EGO service.

Using this tutorial, you will …

- Open a connection to the EGO Web Service endpoint
- Retrieve and print out resource info
- Register the client with Platform EGO and print out the registration response
- Locate all clients and print out the client info
- Query all EGO services
- Create and run an EGO service
- Subscribe to notification of service state changes
- Check for service state and instance state changes
- Stop the EGO service
- Unsubscribe to service notifications

Underlying principles

The sample uses a separate thread to interact with a service. It creates a service and subscribes to service event notifications, which are sent by the Service Controller whenever there is a change in service state. The thread waits until a notification arrives. It also queries the service whenever these notifications occur.

Step 1: Import class references

Import the necessary classes and interfaces that are required by the client to invoke the Web Service.

Step 2: Retrieve resource information

Refer to Tutorial 1: Step 3: Retrieve resource information on page 115.

Step 3: Register the client

The register operation takes three input parameters: client ID (processedArgs[2]), description (processedArgs[3]), and the URL for service notification (processedArgs[4]); this URL, derived from the ProcessArgs() method in sample 2, is an endpoint that implements the EGO notification WSDL interface. Platform EGO will then send service notifications to the client using this endpoint. Refer to Tutorial 2: Step 3: Register the client on page 120 for general information about the registration process.

Step 4: Locate all clients

Refer to Tutorial 2: Step 4: Locate the client on page 122

Step 5: Query all EGO services

Create the QueryServiceRequestDocument (qreqDoc) and QueryServiceRequest (qReq) objects. The QueryServiceRequest object has one member variable, which is the service name. In this case, we set the service name to null in order to get information about all services running on Platform EGO.

The EGO WSDLs and Web Service gateway support Web Service Security (WSSE specification). This means that different types of security information can be passed in the header of SOAP messages sent by the clients. The samples use the simplest form, i.e., username and password authentication. The wrappers generated for Java have signatures that provide for multiple types of security information to be included. In this case, we are using just one security document (logonDoc), which we pass along with qreqDoc to the QueryService method.

The response to a service query request consists of a structure that includes the number of services, service names, descriptions, states, and host names amongst others.

```
public void queryService(String sname)
{
QueryServiceRequestDocument qreqDoc =
 QueryServiceRequestDocument.Factory.newInstance();
QueryServiceRequest qReq = qreqDoc.addNewQueryServiceRequest();
qReq.setServiceName(sname);
QueryServiceResponseDocument qresDoc;
QueryServiceResponse qres;

try {
  qresDoc = servicePort.QueryService(qreqDoc, logonDoc);
  qres = qresDoc.getQueryServiceResponse();
  print(qres);
} catch(RemoteException rex) {
rex.printStackTrace();
}
}
```

Step 6: Create a service definition

The first step in creating an EGO service is to specify a service definition. Each service is described by a service definition that consists of an XML file containing information about the service such as the type of resources required to run the service instances and how to start and monitor them. The serviceDefinition() method creates an instance of the service definition object. Set all the member variables that make up a service definition. (Service name and description are global string variables that have already been initialized.)

Integral parts of the service definition are the activity and allocation specifications. Define and initialize an activity specification including the setting of its resource limits to default values. The activity specification essentially defines a job that the

user wants to be executed. The actSpec.setCommand method specifies the actual binary that should be executed. In the sample, we want the program "sleep" to be executed. The sleep command takes the number of seconds to sleep (for unix) as an input argument.

Create an allocation specification. The allocation specification describes a request to Platform EGO for an allocation of resources. The only required variables for the allocation specification are the consumer name to charge this allocation to, and the resource specification of what is being requested.

```
ServiceDefinition serviceDefinition()
{
ServiceDefinition sdef = ServiceDefinition.Factory.newInstance();
sdef.setDescription(serviceDescription);
sdef.setMaxInstances(2);
sdef.setMinInstances(1);
sdef.setServiceName(serviceName);
sdef.setActivityDescriptionArray(new
 ActivityDescription[]{activityDescription()});
sdef.setAllocationSpecification(allocationSpecification());
sdef.setControlPolicy(controlPolicy());
return sdef;
}

private ActivityDescription activityDescription()
{
EnvironmentVariable [] env = new EnvironmentVariable [] {};
Rlimit [] rlimits = new Rlimit[] {};

ActivityDescription actd = ActivityDescription.Factory.newInstance();
ActivitySpecification actSpec = actd.addNewActivitySpecification();
actSpec.setActivityName("Sample6ServiceActivity");
actSpec.setCommand("/bin/sleep 120");
actSpec.setEnvironmentVariableArray(env);
actSpec.setExecutionUser("lsfadmin");
actSpec.setWorkingDirectory("/tmp");
actSpec.setUmask("0777");
actSpec.setRlimitArray(rlimits);
        return actd;
}
private AllocationSpecification allocationSpecification()
{
AllocationSpecification alocSpec =
 AllocationSpecification.Factory.newInstance();
alocSpec.setAllocationName("Sample6ServiceAllocation");
alocSpec.setConsumerName("SampleApplications/EclipseSamples");
ResourceSpecification [] resourceSpecs;
resourceSpecs = createResourceSpecification(1);
alocSpec.setResourceSpecificationArray(resourceSpecs);
//alocSpec.setOptionArray(new String[]{"EGO_ALLOC_EXCLUSIVE"});
return  alocSpec;
}
```

Step 7: Create a Service Controller Client object

Create a ServiceControllerClient object that implements the Runnable interface. The ServiceControllerClient class implements a run() method that enables you to subscribe to service notifications, as well as create and query a service. When this class is instantiated and the run() method is called, a new thread will be spawned. The stop() method will disable and remove the service, and unsubscribe the client to notifications.

The following steps describe the run() and stop() methods of the ServiceControllerClient class in more detail.

```
public class ServiceControllerClient implements Runnable {
private EGOclient egoClient;
private boolean finish;
public ServiceControllerClient(EGOclient egoClient)
{
this.egoClient = egoClient;
}

public void run()
{
String id = egoClient.subscribeNotification(egoClient.notificationEndPoint);
egoClient.createService(egoClient.serviceDefinition());

while(!finish) {
try {
              Notification.notification.wait();
         ServiceStateChange stateChange =
 Notification.notification.getServiceStateChange();
         ServiceInstanceStateChange instanceStateChange =
 Notification.notification.getServiceInstanceStateChange();
} catch(InterruptedException ie) {
  ie.printStackTrace();
}
egoClient.queryService(egoClient.serviceName);
}
}

public void stop()
{
egoClient.controlService(egoClient.serviceName,
 ServiceControlOperation.DISABLE);
egoClient.removeService(egoClient.serviceName);
egoClient.unsubscribeNotification(egoClient.subscriptionID);

finish = true;
this.notify();
}
}
```

Step 8: Subscribe to notifications

In order to receive notifications about service state changes, it is necessary to subscribe to the notification service.

The procedure for issuing a notification subscription request is similar to other requests previously described. Create a subscription request document (nrDoc) and a subscription request object (nr). Set the notification endpoint to tell the Service Controller where to send the notification messages. The notification endpoint is derived from the ProcessArgs() method in Sample 2.

Create a subscription response document (nresDoc), a security document, and a subscription response object (nres). Pass the request and logon documents to the ServiceNotificationSubscribe operation. When the operation is invoked, it returns a subscription ID, which is printed out.

```
public String subscribeNotification(String endpoint)
{
String id = null;
ServiceNotificationSubscribeRequestDocument nrDoc =
 ServiceNotificationSubscribeRequestDocument.Factory.newInstance();
ServiceNotificationSubscribeRequest nr =
 nrDoc.addNewServiceNotificationSubscribeRequest();
nr.setNotificationEndpoint(endpoint);
ServiceNotificationSubscribeResponseDocument nresDoc;
ServiceNotificationSubscribeResponse nres;
try {
  nresDoc = servicePort.ServiceNotificationSubscribe(nrDoc, logonDoc);
  nres = nresDoc.getServiceNotificationSubscribeResponse();
  print(nres);
  id = nres.getSubscriptionID();
} catch (RemoteException rex) {
rex.printStackTrace();
}
subscriptionID = id;
return id;
}
```

Step 9: Create and start an EGO service

Pass the service definition object to the createService() method. The createService() method creates a new service object and starts the service based on the service definition provided. Once the service is started, the Service Controller allocates resources and starts service instances.

Create the CreateServiceRequestDocument (sreqDoc) and CreateServiceRequest (sReq) objects. The CreateServiceRequest object has one member variable, which is the service name. In this case, we set the service name to null in order to get information about all services running on Platform EGO.

Pass sreqDoc and the logonDoc security document to the CreateService method and print out the response.

```
public void createService(ServiceDefinition serviceDef)
{
CreateServiceRequestDocument sreqDoc =
 CreateServiceRequestDocument.Factory.newInstance();
CreateServiceRequest sreq = sreqDoc.addNewCreateServiceRequest();
sreq.setServiceDefinition(serviceDef);


CreateServiceResponseDocument sresDoc;
CreateServiceResponse sres;

        try {
  sresDoc = servicePort.CreateService(sreqDoc, logonDoc);
  sres = sresDoc.getCreateServiceResponse();
  print(sres);
        } catch(RemoteException rex) {
        rex.printStackTrace();
        }
}
```

Step 10: Check for service state changes

Once the service has been created and is running, the Service Controller Client thread will pause execution while it waits for service state change notifications from the Service Controller. If either a service state change or a service instance state change occurs, thread execution resumes and retrieves the notification message.

When the state change information is retrieved, the service is queried. Refer to Step 5: Query all EGO services on page 153.

```
try {
              Notification.notification.wait();
         ServiceStateChange stateChange =
 Notification.notification.getServiceStateChange();
         ServiceInstanceStateChange instanceStateChange =
 Notification.notification.getServiceInstanceStateChange();
} catch(InterruptedException ie) {
  ie.printStackTrace();
}
```

Step 11: Stop an EGO service

Create the respective documents and objects for the ControlService request and response. Set the service name and control operation for the control service request object. In this case, set the control operation to disable the service. Create the security documents and pass the documents with the logon document and ControlService request document to the ControlService operation. The Service Controller stops all service instances and de-allocates resources.

Remove the service by specifying the service name while invoking the removeService operation. The Service Controller destroys the service object and removes the service definition from the configuration.

```
public void controlService(String sname, ServiceControlOperation.Enum op)
{
ControlServiceRequestDocument reqDoc =
 ControlServiceRequestDocument.Factory.newInstance();
ControlServiceRequest req = reqDoc.addNewControlServiceRequest();
req.setServiceName(sname);
req.setServiceControlOperation(op);

ControlServiceResponseDocument resDoc;
ControlServiceResponse res;

SecurityDocument sdoc2 = SecurityDocument.Factory.newInstance();
SecurityHeaderType sec2 = sdoc2.addNewSecurity();
SecurityDocument sdoc3 = SecurityDocument.Factory.newInstance();
SecurityHeaderType sec3 = sdoc3.addNewSecurity();
SecurityDocument sdoc4 = SecurityDocument.Factory.newInstance();
SecurityHeaderType sec4 = sdoc4.addNewSecurity();
SecurityDocument sdoc5 = SecurityDocument.Factory.newInstance();
SecurityHeaderType sec5 = sdoc5.addNewSecurity();
SecurityDocument sdoc6 = SecurityDocument.Factory.newInstance();
SecurityHeaderType sec6 = sdoc6.addNewSecurity();
SecurityDocument sdoc7 = SecurityDocument.Factory.newInstance();
SecurityHeaderType sec7 = sdoc7.addNewSecurity();
SecurityDocument sdoc8 = SecurityDocument.Factory.newInstance();
SecurityHeaderType sec8 = sdoc8.addNewSecurity();

try {
resDoc = servicePort.ControlService(reqDoc, logonDoc, sdoc2, sdoc3, sdoc4,
 sdoc5, sdoc6, sdoc7, sdoc8);
res = resDoc.getControlServiceResponse();
print(res);
} catch (RemoteException rex) {
rex.printStackTrace();
}
}
```

Step 12: Unsubscribe to service notifications

Create the respective documents and objects for the ServiceNotificationUnsubscribe request and response. Use the subscription ID previously obtained during the subscription process to unsubscribe to service notifications.

Create the security documents and pass the documents with the logon document and ServiceNotificationUnsubscribe request document to the ServiceNotificationUnsubscribe operation. The Service Controller will no longer communicate service state change notifications to the notification endpoint.

```
public void unsubscribeNotification(String id)
{
ServiceNotificationUnsubscribeRequestDocument nrDoc =
 ServiceNotificationUnsubscribeRequestDocument.Factory.newInstance();
ServiceNotificationUnsubscribeRequest nr =
 nrDoc.addNewServiceNotificationUnsubscribeRequest();
nr.setSubscriptionID(id);
ServiceNotificationUnsubscribeResponseDocument nresDoc;
ServiceNotificationUnsubscribeResponse nres;
SecurityDocument sdoc2 = SecurityDocument.Factory.newInstance();
SecurityHeaderType sec2 = sdoc2.addNewSecurity();
SecurityDocument sdoc3 = SecurityDocument.Factory.newInstance();
SecurityHeaderType sec3 = sdoc3.addNewSecurity();
SecurityDocument sdoc4 = SecurityDocument.Factory.newInstance();
SecurityHeaderType sec4 = sdoc4.addNewSecurity();
try {
  nresDoc = servicePort.ServiceNotificationUnsubscribe(nrDoc, logonDoc, sdoc2,
 sdoc3, sdoc4);
  nres = nresDoc.getServiceNotificationUnsubscribeResponse();
  print(nres);
} catch (RemoteException rex) {
rex.printStackTrace();
}
}
```

Run the client application

1. Select **Run > Run**.

   The Run dialog appears.

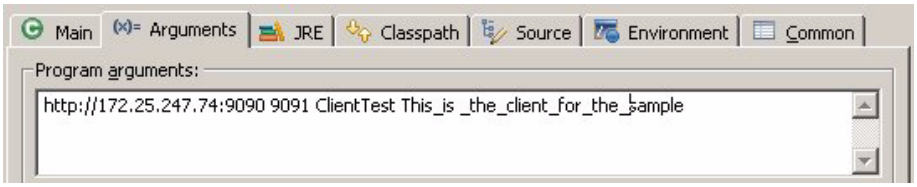
2. In the Configurations list, either select a Java Application or click **New** for a new configuration.

   For a new configuration, enter the configuration name.

3. Enter the project name and Main class.

4 Click the Arguments tab and enter the following arguments in the given order:
   1 URL of the web service gateway
   2 Port number (string) for the notification interface
   3 Client ID (string)
   4 Client description (string).

NOTE: Arguments must be separated by a space.



5 Click **Apply** and then **Run**.

Sample output

```
RegisterResponse: ClientTest


Attribute               egonode-a               nvgesx11                egonode-b
Type                    host                    host                    host
State                   ok                      unavail                 ok
status                  0                                               0
type                    LINUX86                                         LINUX86
model                   PC300                                           PC300
ncpus                   1                                               1
cpuf                    8.300000                                        8.300000
mem                     148.000000                                      162.000000
swp                     544.500000                                      544.500000
maxmem                  250                                             250
maxswp                  556                                             556
tmp                     9936.000000                                     9976.000000
ut                      0.011559                                        0.001019
it                      1322.000000                                     1322.000000
io                      58.218750                                       21.140625
pg                      3.261719                                        1.190430
r1m                     0.000000                                        0.000000
r15s                    1.202148                                        0.999512
r15m                    0.000000                                        0.000000
ls                      1                                               1
slot                    6                                               6
freeslot                2                                               6
RESOURCES               (linux)                                         (linux)
ndisks                  1                                               1
maxtmp                  12191                                           12191
GUIURL_archtsmst        http://archtsmst:8080/platform  53106@172.25.247.177
ClientTest      This_is_the_client_for_the_sample       53756@172.25.247.177
```

```
VMO_Manager     VMO_Manager     46150@172.25.247.177
SSM_/Application/SymSOATest                     40823@172.25.247.173
WSG             Web Service Gateway     46021@172.25.247.177
SD_SSM          archtsmst:46001 46005@172.25.247.177
SD_ADMIN        archtsmst:7874  46004@172.25.247.177
SD_SDK          archtsmst:7873  46003@172.25.247.177
SD_SDK_/Application/SymSOATest                  46002@172.25.247.177
RS_DEPLOY       archtsmst:7875  45984@172.25.247.177
EGO_SERVICE_CONTROLLER  url=archtsmst:7872      45971@172.25.247.177
VMOManager      defined
WEBGUI          started
sunsps          defined
SymphonyRS      started
SymphonySD      started
moviestore      defined
vnc             defined
proftp          defined
sddnsserver     started
wsgserver       started
Service Notification Subscription Id http://userlab03.lsf.platform.com:9091/axis/services/
```

Tutorial 7: Create an EGO Service and Query the Domain Name Server

This tutorial describes how to create and run an EGO service, and query the Domain Name Server (DNS) for host information. The DNS is a standard naming service under the control of the Service Director.

Using this tutorial, you will …

- Open a connection to the EGO Web Service endpoint
- Retrieve and print out resource info
- Register the client with Platform EGO and print out the registration response
- Locate all clients and print out the client info
- Query all EGO services
- Create a service definition
- Create and start an EGO service
- Query the DNS
- Stop the EGO service

Underlying principles

In order to communicate with a service on a host cluster, its location must be known. Due to the nature of distributed computing, the service can be running on any host. Normally, when a service instance switches into the run state, the Service Controller sends a notification to the Service Director that includes location information of the service instance. The Service Director then adds the location record to its DNS server. In this sample, we create a service called "sample6Service". When this service is running, the Service Director automatically updates its DNS records to reflect the new service instance.

The service is created from a new thread (see Sample 6). Once the service is running, we query the Service Controller for service instance information. Then we query the Service Controller, via the Service Director, for the IP address of the host that the service in running on.

Step 1: Import class references

Import the necessary classes and interfaces that are required by the client to invoke the Web Service.

Step 2: Register the client

Refer to Tutorial 2: Step 3: Register the client on page 120.

Step 3: Retrieve resource information

Refer to Tutorial 1: Step 3: Retrieve resource information on page 115.

Step 4: Locate all clients

Refer to Tutorial 2: Step 4: Locate the client on page 122.

Step 5: Query all services

Refer to Tutorial 6: Step 5: Query all EGO services on page 153.

Step 6: Create a service definition

Refer to Tutorial 6: Step 6: Create a service definition on page 153.

Step 7: Create a Service Controller Client object

Create a ServiceControllerClient object that implements the Runnable interface. This object will interact with the Service Controller as a client. The ServiceControllerClient class contains a run() method that enables you to create and query a service; refer to Tutorial 6: Step 7: Create a Service Controller Client object on page 155 for the sample code. Define a new thread (scThread) and pass the ServiceControllerClient object to it. When the start() method is called, a new thread will be spawned. The stop() method will disable and remove the service.

Block the main thread for 120 seconds while the service is created and started. As the service starts, the Service Controller notifies the client of service state changes.

Send a query to the Service Controller to retrieve service info. The response to a service query request with a null input argument consists of a structure that includes the total number of services, service names, descriptions, states, and host names amongst others.

```
ServiceControllerClient scClient = new ServiceControllerClient(client);
Thread scThread = null;
try {
scThread = new Thread(scClient);
    scThread.start();
} catch(Exception e) {}

Thread.sleep(120 *1000);

try {
  client.queryService(null);
} catch(Exception e) {}
```

Step 8: Create and start an EGO service

Refer to Tutorial 6: Step 9: Create and start an EGO service on page 156.

Step 9: Query the DNS

With the service running, we pass the service name to the getByName() method, which is a member of the InetAddress class. This method returns an InetAddress object containing the IP address and name of the host where the service instance is running. The address is returned by the Service Director's DNS.

```
InetAddress address = client.queryDNS(client.serviceName);
System.err.println("Service:"  + client.serviceName + "Host:" + address.getHostName() +
            "Address:" + address.getHostAddress());

public InetAddress queryDNS(String name)
{
try {
  InetAddress address = InetAddress.getByName(name);
  return address;
} catch (java.net.UnknownHostException uhe) {
uhe.printStackTrace();
}
return null;
}
```

Step 11: Stop an EGO service

Refer to Tutorial 6: Step 11: Stop an EGO service on page 157 for more information about the stop() method of the ServiceControllerClient class.

Block the main thread for 120 seconds while the service is being stopped. Query the DNS again. There should no longer be a record in the DNS for the service instance. The current thread is then blocked until the ServiceControlClient is notified, which signals the conclusion of the stop() method.

```
try {
  scClient.stop();
    } catch(Exception e) {}

Thread.sleep(120 *1000);

// The address must be removed
address = client.queryDNS(client.serviceName);
System.err.println("Service:"  + client.serviceName + "Host:" + address.getHostName() +
            "Address:" + address.getHostAddress());

    synchronized(scClient) {scClient.wait();}
```

Run the client application

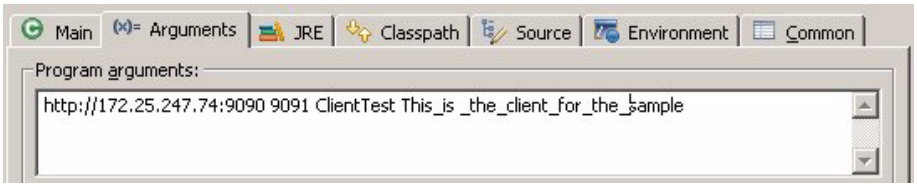
1. Select **Run > Run**.

   The Run dialog appears.
2. In the Configurations list, either select a Java Application or click **New** for a new configuration.

   For a new configuration, enter the configuration name.
3. Enter the project name and Main class.

4 Click the Arguments tab and enter the following arguments in the given order:
   1 URL of the web service gateway
   2 Port number (string) for the notification interface
   3 Client ID (string)
   4 Client description (string).

NOTE: Arguments must be separated by a space.



5 Click **Apply** and then **Run**.

CHAPTER

5

Troubleshooting

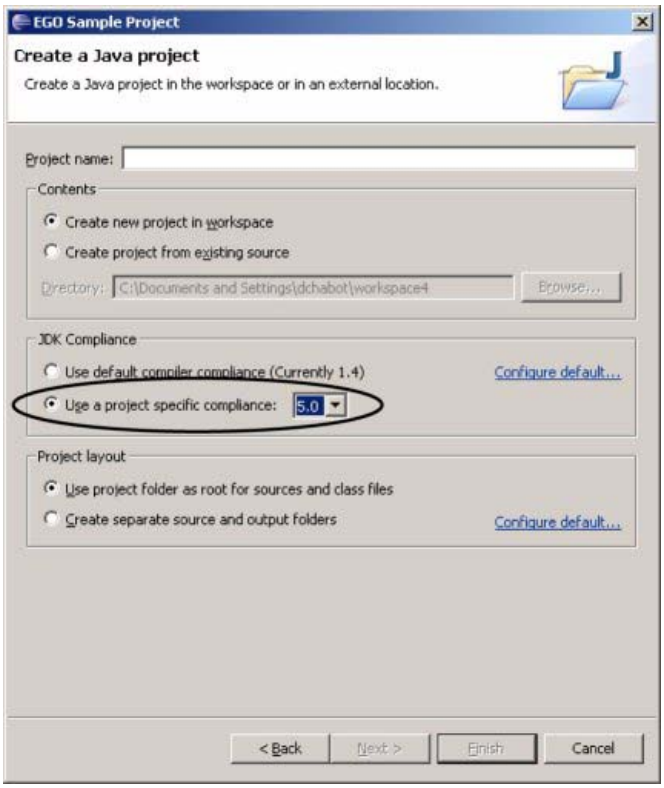
This chapter provides some quick checks for isolating problems when trying to run the samples.

Contents

Compiler errors

Wrong compliance level

Error occurs when trying to build the Java project. To run the SDK samples, the compliance level must be set to 5.0 since the samples use Java 2 Standard Edition (J2SE) 5.0 features. To change the compliance level, create a new project and set the compliance level in the **EGO Project** dialog.



Incorrect number of arguments

Each Java sample requires a specific number of arguments to be passed to its main method. Click the Arguments tab and enter the arguments. For example, Sample 5 requires the following arguments in the given order:

1. URL of the web service gateway
2. Port number (string) for the notification interface
3. Client ID (string)
4. Client description (string).

Connection errors

NOTE: Users should be familiar with the EGO runtime setup before trying to run the client samples.

Incorrect username or password

Check the EGO runtime setup. The samples use "Admin" as the username and password when logging on to EGO.

Platform EGO not running

Check that the Platform EGO runtime has been installed, configured, and running on a host cluster. For C clients, check that the EGO daemon port numbers in the ego.conf file match the port numbers configured for the EGO master host. The C samples reads the information in the ego.conf file when opening a connection to Platform EGO.

Incorrect URL

For Java clients, try to connect to the Web Service Gateway using a browser by entering the URL or IP address and port number of the gateway. If you cannot connect through the browser, verify the URL of the gateway and check that it is running. The gateway is installed as part of Platform EGO. It comes with a configuration file called wsg.conf where the port number for the gateway can be defined. By default the port number is 9090. The default location for the wsg.conf file is /opt/ego/kernel/conf/. The gateway URL can then be formed by using the host name where the gateway is running and the port number, i.e., http://host:port.

If you can connect to the gateway from a browser but cannot connect to it using the samples, verify that the URL of the gateway matches the URL that is passed as an argument to the client sample. This URL consists of an IP address or host name and a port number.

DNS cannot resolve host name

Verify that the TCP/IP configuration of the client host is properly set up. Check that it does not have an external DNS server, such as a DNS server from an Internet service provider (ISP). If the client is configured to use an external DNS server, it may be unable to resolve internal names. This can also cause problems with conflicting internal and external namespaces. Likewise, the Web Service gateway should be properly configured and have an appropriate DNS.

Notification problems

Platform EGO communicates notifications such as allocation request responses and state changes to the client using a customer designated port number on the client host. This port number or endpoint implements the EGO notification WSDL interface. Platform EGO will send notifications to the client using this endpoint. If the port is busy, i.e., already in use by another application, Platform EGO will be unable to send the notifications to the client. Verify that the selected port number is available.

Index