Load Balancing

Previous

Next

Replication is an important IceGrid feature but, when combined with load balancing, replication becomes even more useful.

IceGrid nodes regularly report the system load of their hosts to the registry. The replica group's configuration determines whether the registry actually considers system load information while processing a locate request. Its configuration also specifies how many replicas to include in the registry's response.

IceGrid's load balancing capability assists the client in obtaining an initial set of endpoints for the purpose of establishing a connection. Once a client has established a connection, all subsequent requests on the proxy that initiated the connection are normally sent to the same server without further consultation with the registry. As a result, the registry's response to a locate request can only be viewed as a snapshot of the replicas at a particular moment. If system loads are important to the client, it must take steps to periodically contact the registry and update its endpoints.

On this page:

Replica Group Load Balancing

A replica group descriptor optionally contains a load balancing descriptor that determines how system loads are used in locate requests. The load balancing descriptor specifies the following information:

- Type
  Several load balancing types are supported.

- Sampling interval
  One of the load balancing types considers system load statistics, which are reported by each node at regular intervals. The replica group can specify a sampling interval of one, five, or fifteen minutes. Choosing a sampling interval requires balancing the need for up-to-date load information against the desire to minimize transient spikes. On Unix platforms, the node reports the system's load average for the selected interval, while on Windows the node reports the CPU utilization averaged over the interval.

- Number of replicas
  The replica group can instruct the registry to return the endpoints of one (the default) or more object adapters. If the specified number *N* is larger than one, the proxy returned in response to a locate request contains the endpoints of at most *N* object adapters. If *N* is 0, the proxy contains the endpoints of all the object adapters. The Ice run time in the client selects one of these endpoints at random when establishing a connection.

For example, the descriptor shown below uses adaptive load balancing to return the endpoints of the two least-loaded object adapters sampled with five-minute intervals:

XML

```xml
<replica-group id="ReplicatedAdapter">
    <load-balancing type="adaptive" load-sample="5" n-replicas="2"/>
</replica-group>
```

The type must be specified, but the remaining attributes are optional.

IceGrid ignores the object adapters of a disabled server when executing a locate request, meaning the client that initiated the locate request will not receive the endpoints for any of these object adapters.

You can optionally use custom load balancing strategies by installing replica group filters.

Back to Top ^

Load Balancing Types

A replica group can select one of the following load balancing types:

- Random
  Random load balancing selects the requested number of object adapters at random. The registry does not consider system load for a replica group with this type.
- Adaptive
  Adaptive load balancing uses system load information to choose the least-loaded object adapters over the requested sampling interval. This is the only load balancing type that uses sampling intervals.
- Round Robin
  Round robin load balancing returns the least recently used object adapters. The registry does not consider system load for a replica group with this type. Note that the round-robin information is not shared between registry replicas; each replica maintains its own notion of the "least recently used" object adapters.
- Ordered
  Ordered load balancing selects the requested number of object adapters by priority. A priority can be set for each object adapter member of the replica group. If you define several object adapters with the same priority, IceGrid will order these object adapters according to their order of appearance in the descriptor.

Choosing the proper type of load balancing is highly dependent on the needs of client applications. Achieving the desired load balancing and fail-over behavior may also require the cooperation of your clients. To that end, it is very important that you understand how and when the Ice run time uses a locato r to resolve indirect proxies.

Back to Top ^

Using Load Balancing in the Ripper Application

The only change we need to make to the ripper application is the addition of a load balancing descriptor:

XML

```
<icegrid>
    <application name="Ripper">
        <replica-group id="EncoderAdapters">
            <load-balancing type="adaptive"/>
            <object identity="EncoderFactory" type="::Ripper::MP3EncoderFactory"/>
        </replica-group>
        <server-template id="EncoderServerTemplate">
            <parameter name="index"/>
            <parameter name="exepath" default="/opt/ripper/bin/server"/>
            <server id="EncoderServer${index}" exe="${exepath}" activation="on-demand">
                <adapter name="EncoderAdapter" replica-group="EncoderAdapters"
                    endpoints="tcp"/>
            </server>
        </server-template>
        <node name="Node1">
            <server-instance template="EncoderServerTemplate" index="1"/>
        </node>
        <node name="Node2">
            <server-instance template="EncoderServerTemplate" index="2"/>
        </node>
    </application>
</icegrid>
```

Using adaptive load balancing, we have regained the functionality we forfeited when we introduced replica groups. Namely, we now select the object adapter on the least-loaded node, and no changes are necessary in the client.

Back to Top ^

Interacting with Object Replicas

In some applications you may have a need for interacting directly with the replicas of an object. You might be tempted to call `ice_getEndpoints` on the proxy of a replicated object in an effort to obtain the endpoints of all replicas, but that is not the correct solution because the proxy is indirect and therefore contains no endpoints. The proper approach is to query well-known objects using the `findAllReplicas` operation.

Back to Top ^

Custom Load Balancing Strategies

The IceGrid registry allows you to plug in custom load balancing implementations that the registry invokes to filter its query results. Two kinds of filters are supported:

- Replica group filter
  The registry invokes a replica group filter each time a client requests the endpoints of a replica group or object adapter, as well as for calls to `findAllReplicas`. The registry passes information about the query that the filter can use in its implementation, including the list of object adapters participating in the replica group whose nodes are active at the time of the request. The object adapter list is initially ordered using the load balancing type configured for the replica group; the filter can modify this list however it chooses.

- Type filter
  The registry invokes a type filter for each query that a client issues to find a well-known object by type using the operations `findObjectByType`, `findAllObjectsByType`, and `findObjectByTypeOnLeastLoadedNode`. Included in the information passed to the filter is a list of proxies for the matching well-known objects; the filter implementation decides which of these proxies are returned to the client.

In the sections below we describe how to implement these filters.

Back to Top ^

Overview of Custom Load Balancing

Filters are installed into the IceGrid registry using the standard Ice plug-in facility. The registry is implemented in C++, using the Ice C++98 mapping, therefore the filters must be implemented in C++ with the C++98 mapping.

The Registry Plug-in Facade Object

During initialization, your plug-in will obtain a reference to a facade object with which it can register one or more filters. A filter typically retains a reference to this facade object because it offers a number of useful methods that the filter might need during its implementation. The `RegistryPluginFacade` class provides the following methods:

C++98

```
namespace IceGrid
{
    class RegistryPluginFacade : virtual public Ice::LocalObject
    {
    public:
        void addReplicaGroupFilter(const std::string& id, const IceGrid::ReplicaGroupFilterPtr& filter);
        bool removeReplicaGroupFilter(const std::string& id, const IceGrid::ReplicaGroupFilterPtr& filter);

        void addTypeFilter(const std::string& id, const IceGrid::TypeFilterPtr& filter);
        bool removeTypeFilter(const std::string& id, const IceGrid::TypeFilterPtr& filter);

        IceGrid::ApplicationInfo getApplicationInfo(const std::string& name) const;

        IceGrid::ServerInfo getServerInfo(const std::string& serverId) const;

        std::string getAdapterServer(const std::string& adapterId) const;
        std::string getAdapterApplication(const std::string& adapterId) const;
        std::string getAdapterNode(const std::string& adapterId) const;
        IceGrid::AdapterInfoSeq getAdapterInfo(const std::string& adapterId) const;
        std::string getPropertyForAdapter(const std::string& adapterId, const std::string& name) const;

        IceGrid::ObjectInfo getObjectInfo(const Ice::Identity& id) const;

        IceGrid::NodeInfo getNodeInfo(const std::string& name) const;
        IceGrid::LoadInfo getNodeLoad(const std::string& name) const;
    }
    typedef ... RegistryPluginFacadePtr;

    RegistryPluginFacadePtr getRegistryPluginFacade();
}
```

There are methods for adding and removing replica group and type filters, along with a number of methods for obtaining information about the deployment. As you can see, a great deal of information is available to a filter implementation for use in making its decisions. The data structures returned by these methods correspond directly to their XML descriptors; you can also review the Slice definitions of the IceGrid data types for more information.

The methods are described below:

- `void addReplicaGroupFilter(const std::string& id, const IceGrid::ReplicaGroupFilterPtr& filter)`
  Adds a replica group filter with the given identifier. The identifier must match the `filter` attribute of a `replica-group` descriptor. The registry maintains a list of filters for each identifier. If you register more than one filter with the same identifier, the registry invokes each one in turn, in the order of registration. To add a replica group filter for dynamically registered replica groups, you should use the empty string for the identifier.

- `bool removeReplicaGroupFilter(const std::string& id, const IceGrid::ReplicaGroupFilterPtr& filter)`
  Removes an existing replica group filter that matches the given identifier and smart pointer. Returns true if a match was found, false otherwise.

- `void addTypeFilter(const std::string& id, const IceGrid::TypeFilterPtr& filter)`
  Adds a type filter with the given identifier. The registry maintains a list of filters for each identifier. If you register more than one filter with the same identifier, the registry invokes each one in turn, in the order of registration.

- `bool removeTypeFilter(const std::string& id, const IceGrid::TypeFilterPtr& filter)`
  Removes an existing type filter that matches the given identifier and smart pointer. Returns true if a match was found, false otherwise.

- `IceGrid::ApplicationInfo getApplicationInfo(const std::string& name) const`
  Returns the descriptor for the application with the given name. Raises `IceGrid::ApplicationNotExistException` if no match is found.

- `IceGrid::ServerInfo getServerInfo(const std::string& serverId) const`
  Returns the descriptor for the server with the given identifier. Raises `IceGrid::ServerNotExistException` if no match is found.

- `std::string getAdapterServer(const std::string& adapterId) const`
  Returns the identifier of the server hosting the object adapter with the given adapter identifier. Returns an empty string if no match is found.

- `std::string getAdapterApplication(const std::string& adapterId) const`
  Returns the name of the application containing the given object adapter identifier. Returns an empty string if no match is found.

- `std::string getAdapterNode(const std::string& adapterId) const`
  Returns the name of the node containing the given object adapter identifier. Returns an empty string if no match is found.

- `IceGrid::AdapterInfoSeq getAdapterInfo(const std::string& adapterId) const`
  If `adapterId` is a replica group identifier, this method returns a sequence of adapter descriptors for all of the replicas. Otherwise, this method returns a sequence containing one descriptor for the given object adapter. Raises `IceGrid::AdapterNotExistException` if no match is found.

- `std::string getPropertyForAdapter(const std::string& adapterId, const std::string& name) const`
  Obtains the value of a server configuration property with the key `name` for the server hosting the object adapter identified by `adapterId`. Returns an empty string if no match is found.

- `IceGrid::ObjectInfo getObjectInfo(const Ice::Identity& id) const`
  Returns information about an object with the given identity. Raises `IceGrid::ObjectNotRegisteredException` if no match is found.

- `IceGrid::NodeInfo getNodeInfo(const std::string& name) const`
  Returns the descriptor for the node with the given name. Raises `IceGrid::NodeNotExistException` if no match is found.

- `IceGrid::LoadInfo getNodeLoad(const std::string& name) const`
  Returns load information for the node with the given name. Raises `IceGrid::NodeNotExistException` if no match is found.

Methods that require access to deployment information raise `IceGrid::RegistryUnreachableException` if called before the registry has fully initialized itself.

Back to Top ^

Implementing a Registry Plug-in

The API for creating an Ice plug-in using C++ requires a factory function with external linkage, along with a class that implements the `Ice::Plugin` local Slice interface. We can use the code from the example in `demo/IceGrid/customLoadBalancing` to illustrate these points. First, the factory function instantiates and returns the plug-in:

C++98

```
extern "C"
{
    ICE_DECLSPEC_EXPORT Ice::Plugin*
    createRegistryPlugin(const Ice::CommunicatorPtr& communicator, const string&, const Ice::StringSeq&)
    {
        return new RegistryPluginI(communicator);
    }
}
```

The plug-in class is straightforward:

C++98

```
class RegistryPluginI : public Ice::Plugin
{
public:
    RegistryPluginI(const Ice::CommunicatorPtr& communicator)
        : _communicator(communicator)
    {
    }

    virtual void initialize()
    {
        IceGrid::RegistryPluginFacadePtr facade = IceGrid::getRegistryPluginFacade();
        if(facade)
        {
            facade->addReplicaGroupFilter("filterByCurrency", new ReplicaGroupFilterI(facade));
        }
    }

    virtual void destroy()
    {
    }

private:
    const Ice::CommunicatorPtr _communicator;
};
```

The `initialize` method calls `getRegistryPluginFacade` to obtain a smart pointer for the registry's facade object. The plug-in uses this object to install a replica group filter.

We describe filter implementations in more detail below.

Back to Top ^

Installing a Registry Plug-in

Continuing with our example from `demo/IceGrid/customLoadBalancing`, we use the following property to install our plug-in in the IceGrid registry:

```
Ice.Plugin.RegistryPlugin=RegistryPlugin:createRegistryPlugin
```

The `Ice.Plugin` property must be defined in the registry's configuration file.

✓ Make sure to configure all of the replicas to load the same registry plug-ins, otherwise a client could get different behavior depending on which replica it's currently using.

Back to Top ^

Filter Implementation Techniques

A filter may require client-specific information in order to assemble its list of results. We recommend using request contexts for this purpose. Briefly, a request context is a dictionary of key/value string pairs that a client can configure and send along as "out of band" metadata accompanying a request. Ice provides several ways for a client to establish a request context:

- Implicit - provides a default request context for every request on all proxies
- Per-proxy - configures a default request context for every request on a particular proxy
- Explicit - specifies a request context at the time of invocation, overriding any default context

Our goal here is to transfer information from a client to a filter. Consequently, we don't recommend using implicit contexts because the context will add overhead to *every* invocation the client makes, not just the ones that involve the filter. To decide whether to use per-proxy or explicit contexts, we first must understand the circumstances in which each kind of filter receives a request context:

- Replica group filter

  Most invocations involving a replica group filter occur when the Ice run time in a client issues requests on a registry. This internal activity is triggered by the client's invocation on a proxy, but Ice doesn't use the client's proxy or the request context that the client may have configured for its proxy or passed explicitly to the invocation. Rather, the Ice run time uses the locator that the client configured for its proxy or communicator. As a result, the request contexts passed to a replica group filter are those configured for the *locator* proxy. (The only exception is when a client invokes `findAllReplicas` directly on the registry via its `Query` interface; refer to the discussion of type filters below for more details.)

  There are several ways you can configure a request context for the locator proxy. If a client configures its locator proxy statically using properties, the simplest solution is to add `Context` properties to the client's configuration, such as:

  ```
  Ice.Default.Locator=...
  Ice.Default.Locator.Context.someKey=someValue
  ```

  If you need to define the context at run time, you can obtain the locator proxy by calling `getDefaultLocator` on the communicator, create a new locator proxy with the desired context by calling `ice_context`, and then replace the locator proxy by calling `setDefaultLocator`.

  Both of these approaches are examples of per-proxy request contexts.

- Type filter

  Invocations involving type filters occur when a client invokes directly on the registry using its `Query` interface. To use per-proxy request contexts, configure the `Query` proxy as necessary. Explicit request contexts can also be used when querying the registry.

So far we've discussed how client-specific information can be passed to a filter, but what if the filter needs to obtain more information about the object adapters (in the case of a replica group filter) or objects (in the case of a type filter) in order to perform its duties? This is where the registry's facade object comes in handy, as with it the filter can retrieve information about the deployment. For example, a replica group filter can use server-specific properties as a form of metadata. The registry supplies the filter with a list of object adapter identifiers; each object adapter is hosted by a server, and the filter can look up property values for that server using the facade. The sample filter in `demo/IceGrid/customLoadBalancing` uses this technique, and we describe it in more detail below.

Back to Top ^

Implementing a Custom Replica Group Filter

A replica group filter must define a subclass of `ReplicaGroupFilter`:

C++98

```
namespace IceGrid
{
    class ReplicaGroupFilter : virtual public Ice::LocalObject
    {
    public:
        virtual Ice::StringSeq filter(const std::string& id,
                                      const Ice::StringSeq& adapters,
                                      const Ice::ConnectionPtr& connection,
                                      const Ice::Context& context);
    };
}
```

The registry passes the following arguments to the `filter` method:

- `id`
  The replica group identifier involved in this request.
- `adapters`
  A sequence of object adapter identifiers denoting the object adapters participating in the replica group whose nodes are active at the time of the request. The object adapter list is initially ordered using the load balancing type configured for the replica group.
- `connection`
  The incoming connection from the client to the registry.
- `context`
  The request context that accompanied the request.

The implementation returns a sequence containing zero or more object adapter identifiers. The registry may truncate this list if the filter supplies more object adapters than the `n-replicas` value configured for the replica group's load balancing policy, but the registry will not change the ordering.

The `filter` implementation must not block.

The C++ example in `demo/cpp98/IceGrid/customLoadBalancing` uses a replica group filter to select only those object adapters that support the currency requested by the client. The replica group's descriptor specifies the filter:

```
<replica-group id="ReplicatedPricingAdapter" filter="filterByCurrency">
  <load-balancing type="random"/>
  <object identity="pricing" type="::Demo::PricingEngine"/>
</replica-group>
```

Notice that the filter identifier `filterByCurrency` matches that used when the plug-in registered the filter.

In this example, the client uses a request context to indicate the desired currency. The context is configured on the locator proxy in the client's configuration file:

```
Ice.Default.Locator=...
Ice.Default.Locator.Context.currency=USD
```

Here we use the `Context` proxy property to statically assign a request context to the locator proxy, which means every invocation on the locator proxy includes the key/value pair `currency/USD`.

In addition to configuring the replica group filter, the deployment descriptor plays another important role here by defining server-specific properties that the filter uses in its implementation:

```
<server-template id="PricingServer">
  <parameter name="index"/>
  <parameter name="currencies"/>
  <server id="PricingServer-${index}" exe="./server" activation="on-demand">
    <adapter name="Pricing" endpoints="tcp -h localhost" replica-group="ReplicatedPricingAdapter"/>
    <property name="Identity" value="pricing"/>
    <property name="Currencies" value="${currencies}"/>
    ...
  </server>
```

```
</server-template>


<node name="node1">
  <server-instance template="PricingServer" index="1" currencies="EUR, GBP, JPY"/>
  <server-instance template="PricingServer" index="2" currencies="USD, GBP, AUD"/>
  <server-instance template="PricingServer" index="3" currencies="EUR, USD, INR"/>
  <server-instance template="PricingServer" index="4" currencies="JPY, GBP, AUD"/>
  <server-instance template="PricingServer" index="5" currencies="GBP, AUD"/>
  <server-instance template="PricingServer" index="6" currencies="EUR, USD"/>
</node>
```

As a convenience, the descriptor file uses a server template to define several servers, with each server supporting a specific set of currencies. The template adds to the property set of each server a property named `Currencies` representing the currencies that the server supports.

Finally, you can see how all of this ties together in the filter implementation:

C++98

```
Ice::StringSeq
ReplicaGroupFilterI::filter(const string& replicaGroupId,
                            const Ice::StringSeq& adapters,
                            const Ice::ConnectionPtr& connection,
                            const Ice::Context& ctx)
{
    Ice::Context::const_iterator p = ctx.find("currency");
    if(p == ctx.end())
    {
        return adapters;
    }

    string currency = p->second;

    Ice::StringSeq filteredAdapters;
    for(Ice::StringSeq::const_iterator p = adapters.begin(); p != adapters.end(); ++p)
    {
        if(_facade->getPropertyForAdapter(*p, "Currencies").find(currency) != string::npos)
        {
            filteredAdapters.push_back(*p);
        }
    }
    return filteredAdapters;

}
```

The code first checks the request context for a value associated with the key `currency` and returns the adapter list unmodified if no entry is found. Next, the method builds a new list of adapters by iterating over the adapter list in its existing order and calling `getPropertyForAdapter` on the facade for each adapter. This method uses the registry's deployment information to find the server hosting the given adapter and then searches the server's configuration properties for one matching the given property name. If the `Currencies` property contains the client's specified currency, the adapter is added to the list that is eventually returned by the filter.

As this example demonstrates, request contexts are a convenient way to supply a filter with client-specific information, and server properties can serve as a simple database when a filter needs to tailor its results based on server attributes.

Back to Top ^

Implementing a Custom Type Filter

A replica group filter must define a subclass of `TypeFilter`:

C++98

```
namespace IceGrid
{
    class TypeFilter : virtual public Ice::LocalObject
    {
    public:
        virtual Ice::ObjectProxySeq filter(const std::string& type,
                                           const Ice::ObjectProxySeq& proxies,
                                           const Ice::ConnectionPtr& connection,
                                           const Ice::Context& context);
    };
}
```

The registry passes the following arguments to the `filter` method:

- `type`
  The type identifier involved in this request.
- `proxies`
  A sequence of proxies denoting the objects that matched the type.
- `connection`
  The incoming connection from the client to the registry.
- `context`
  The request context that accompanied the request.

The implementation returns a sequence containing zero or more proxies.

The `filter` implementation must not block.

Refer to the previous section for more information on implementing a filter.

Back to Top ^

See Also

- Object Adapter Replication
- Connection Establishment
- Replica-Group Descriptor Element
- Load-Balancing Descriptor Element
- Well-Known Objects
- IceGrid Troubleshooting

Previous

Next