ORACLE
FUSION MIDDLEWARE

An Oracle White Paper
July 2012

Load Balancing in Oracle Tuxedo CORBA Applications

ORACLE

Introduction

This paper is intended for the following audiences:

- Designers and programmers knowledgeable about Oracle Tuxedo CORBA and who want to write Oracle Tuxedo CORBA applications

- Designers and programmers knowledgeable about Oracle Tuxedo Application-to-Transaction Monitor Interface (ATMI)

Oracle Tuxedo CORBA is built on top of the Oracle Tuxedo ATMI infrastructure, which has a proven track record of supporting large-scale applications. Both Oracle Tuxedo CORBA and Oracle Tuxedo ATMI inherently provide transparent load balancing, although Oracle Tuxedo CORBA and Oracle Tuxedo ATMI achieve routing and load balancing differently due to the fundamental differences between objects and procedures. Except the difference, Oracle Tuxedo CORBA and ATMI applications share the same fundamental concepts and algorithm in load balancing at ATMI service layer. The other white paper “Load Balancing in Tuxedo ATMI Applications” can help you understand the load balancing under the object layer of Oracle Tuxedo CORBA applications.

The first part of this paper describes the fundamentals needed to understand load balancing in Oracle Tuxedo CORBA. It describes the factors affecting load balancing and how requests are routed through the Oracle Tuxedo infrastructure. The second part of this paper provides several design patterns for different styles of load balancing within Oracle Tuxedo CORBA applications.

Factors Affecting Load Balancing

Load balancing in Oracle Tuxedo CORBA takes place at several different levels:

- Between clients and the Oracle Tuxedo CORBA Internet Inter-ORB Protocol (IIOP) Listeners/Handlers (ISLs/ISHs)
- Between server groups within an Oracle Tuxedo CORBA domain
- Between servers within a server group

Several different factors affect load balancing at these levels within an Oracle Tuxedo CORBA application. In order of relative importance, they are:

- Object ID (OID) – The object identifier assigned to an object
- Activation policy – The activation policy model assigned to an object
- Factories and Factory-Based Routing (FBR) – How factories are distributed and whether factory-based routing is used
- UBB parameters – The settings of certain parameters within the configuration file for the Oracle Tuxedo CORBA application

To best understand how these factors affect load balancing, we will examine all these factors within the context of routing a request in Oracle Tuxedo CORBA.

Requesting Routing in Oracle Tuxedo CORBA

Understanding how Oracle Tuxedo CORBA requests are routed from the client to a server is fundamental to understanding load balancing in Oracle Tuxedo CORBA. The first stage of load balancing takes place between the client and the Oracle Tuxedo CORBA listener/handler (ISL/ISH).

Listener/Handler

An Oracle Tuxedo CORBA client creates a session with the Oracle Tuxedo CORBA domain using a bootstrap object. The construction of the bootstrap object creates a link into the domain. All communications to and from that client, and all servers within the domain, share this single link.

The listener/handler (ISL/ISH) is chosen based on the address parameter passed to the constructor. The address specifies the host and port of a specific listener, but is not limited to a single node (machine) in a multiple-machine Oracle Tuxedo CORBA domain. The address may specify a list of listener host:ports to choose from. The format of the list (comma-separated, parenthesized, or both) determines how an address is chosen from the list (in series, randomly, or a combination of these). When a host:port is selected, the client bootstrap object connects to the listener process. The listener process then reassigns the link to one of several handler processes.

The UBB configuration file for an Oracle Tuxedo CORBA application uses the following parameters to determine the load balancing within the listener/handler:

- MAXWSCLIENTS
- ISL CLOPT –m minh
- ISL CLOPT –M maxh
- ISL CLOPT –x number

MAXWSCLIENTS specifies the total number of remote clients that can access the domain through the listener/handlers. The command line options to ISL control how the ISL process does load balancing:

–m specifies the minimum number of handlers per listener, –M specifies the maximum number of handlers per listener, and –x specifies the number of clients that each handler (ISH) will handle.

When multiple ISLs are configured for a domain, each ISL corresponds to one of the host:ports specified in the address parameter to the bootstrap constructor. For more information on configuring listener/handlers, see Managing Remote Oracle Tuxedo CORBA Client Applications at

http://e-docs.Oracle.com/tuxedo/tux80/atmi/adiiop.htm.

The next two stages of routing, from handler to group and from group to server, are fundamentally different than the routing and balancing of Oracle Tuxedo ATMI. The difference in routing is a direct result of the difference between Oracle Tuxedo CORBA objects and Oracle Tuxedo ATMI services.

Stateful and Stateless Objects

An object's state is the collection of data and properties that fully describe the instance of the object at a particular moment in time. When an object is active, its state is typically maintained in memory in the server. An object's state may also be persistent, or stored in a non-volatile medium, such as a database. An object coordinates the relationship between its in-memory and persistent representations through a state management mechanism such as the Transaction Processing (TP) Framework provided by Oracle Tuxedo.

Stateful and stateless are two terms commonly used to describe objects. In the most literal interpretation, a stateful object is simply an object that has state. When the object is active, that state must be main- tained in memory. A stateless object has no state, so when the object is active, no state is maintained in memory. An Oracle Tuxedo ATMI service is an example of a stateless object.

Another common usage of stateful and stateless implies whether or not the object must remain active between method invocations. In this context, a stateful object must remain in memory between method invocations, while a stateless object may be swapped out of memory between method invocations. This usage of the terms stateful and stateless applies to Oracle Tuxedo CORBA, that is, Oracle Tuxedo CORBA makes a distinction between the life of an object's state and the life of an object's instance. The state management features of Oracle Tuxedo CORBA not only allow an object to have state, but also allow an object to be swapped in and out of memory.

Activation Policy

The activation policy assigned to the object defines when the object is available to be swapped out of memory, or deactivated. Oracle Tuxedo CORBA supports four different state management (or deactivation) models:

- Method – deactivated at the end of every method
- Transaction – deactivated at the end of a transaction
- Application controlled 1 – deactivated under the control of the Oracle Tuxedo CORBA application
- Process – deactivated when the server process is shutdown

The activation policy (could also be called the "deactivation policy") essentially controls the activation life of the object, or the lifetime of the object's state in memory, which has important ramifications to load balancing in an Oracle Tuxedo CORBA application.

[1] Unlike the other activation models, application controlled does not have a specific policy identifier value to be set in the Implementa- tion Configuration File (ICF) file. Instead, it is achieved using the "process" policy and the TP::deactivate_enable() method.

Objects Are Not Services

The most fundamental difference between objects and services is the presence of state in objects. How is this state identified and what are the implications of this difference?

An object's identity, which denotes a specific instance of an object, is typically associated with the state encapsulated by the object. An individual object instance maintains continuity among a series of requests, such that the state encapsulated by the object in subsequent requests reflects the changes made to the object in previous requests.

An application-assigned identifier, the Object ID (OID), is used to establish object identity in Oracle Tuxedo CORBA. Oracle Tuxedo CORBA makes the following guarantee about an object's identity: **Every request to the same object identity will be delivered to the same object instance.** The implication of this guarantee is that if the object is active in memory, the next request on that object must be sent to the exact process that has the object active in memory. If the object is not active, then the request can be sent to any server that offers the object's interface.

Oracle Tuxedo CORBA objects have state, and requests are pinned to a specific server for the activation life of the object. Oracle Tuxedo ATMI services do not have state, so requests may be routed to any available server. In other words, an Oracle Tuxedo ATMI service can move between servers on every invocation, whereas an object cannot move between servers during its activation life. The identity of an object instance and its role in determining an appropriate server are the major areas of change to the Oracle Tuxedo ATMI infra- structure for Oracle Tuxedo CORBA.

The IOR

In CORBA, an object is identified by an Interoperable Object Reference (IOR). The format and meaning of an IOR is described in the CORBA specification in such a way that an IOR created by any CORBA server can be used by any other CORBA client. An Oracle Tuxedo CORBA client is an example of "any CORBA client." (In fact, Oracle Tuxedo CORBA supports many different types of clients, including CORBA C++ clients, CORBA Java clients, and ActiveX clients.) When a client invokes a method on an object reference, the following information in the IOR is used to locate a server that can satisfy that request:

- Host:Port
- ObjectKey

  The CORBA specification defines how the ObjectKey is included in the IOR, but the content of the ObjectKey is vendor-specific. In other words, the meaning of an ObjectKey and how that meaning is used to locate a server is specific to a particular vendor's implementation. The host:port field will deliver the IOR to a CORBA system that is capable of recognizing the format of the ObjectKey and acting on it. In Oracle Tuxedo CORBA, the host:port directs a request to the listener (ISL) process associated with the host:port.

The pertinent information in the ObjectKey is:

- OID
- Group
- Interface

Figure 1. Content of An IOR Affecting Routing

The OID is the application-assigned identifier that was introduced in "Factors Affecting Load Balancing." The group is the server group (like an Oracle Tuxedo ATMI server group) in which the object can exist. The combination (tuple) of OID, interface, and group uniquely identifies an object in Oracle Tuxedo CORBA. The implication of this information is extremely important to load balancing. An object reference is tied to a specific server group.

Oracle Tuxedo CORBA has mechanisms to provide load balancing between server groups when creating an IOR, but once an IOR is assigned, load balancing will occur only between servers within that server group. To have load balancing distributed among server groups (and machines), cross-group considerations must be incorporated into the application design. (There must be a good reason to impose this restriction.) Let's examine why an IOR is tied to a server group.

Note: The IOR contains much more information than just these fields. For details, see the CORBA 2.2 specification, a copy of which is included on the BEA Tuxedo 8.0 Documentation CD.

Active Object Map

When an object is active, Oracle Tuxedo CORBA must keep track of the object's identity so that requests to that object are routed to the server process having that object in memory. A given system may have tens of thousands of objects active at any time, and objects are activated and deactivated at a relatively high rate. To achieve high levels of performance, it is imperative to have an efficient mechanism to determine if an object is currently active and to mark it as active. Oracle Tuxedo CORBA uses an Active Object Map (AOM) for this purpose.

The AOM contains the tuple of information (described in the previous section, "The IOR") that uniquely identifies an object. Because of the volatility of this information, keeping it on a cross-machine (Oracle Tuxedo CORBA domain) basis is not practical. Instead, the information is limited to a single machine, where it can be managed in a shared memory region. For the purposes of routing, the information is tied to an Oracle Tuxedo CORBA server group.

When a request for an object is queued to a server, the object is added to the AOM for that server group, and the object's OID is associated with the server receiving the request. Associating the object's

OID with the server receiving the request is important because doing so prevents requests for the same object appearing on different server queues. An implication of this association is that if two requests for the same object are received, they will both go to the same server queue, even if neither instance of the object is actually active.

When the object is deactivated (just after the return from the deactivate_object() routine), the object is removed from the AOM assuming that there is no other request for that object already queued. There is a finite period of time between when the response is returned to the client and when the object is removed from the AOM. If the next request for the object arrives before the object is actually removed from the AOM, that request is directed to the same server as the previous request. This can occur in the following scenarios:

- The time to deactivate the object is approximately equivalent to the time for the client to make a subsequent request. This scenario is more likely when using native clients (clients running on the same machine on which the server program is running) that have much less latency than remote (IIOP) clients.
- There are many clients trying to access the same object concurrently. This situation may be unavoidable in a given application, but is more likely the result of poor design regarding how OIDs are assigned by the application's factory.

The section entitled "The ObjectID" discusses the assignment of OIDs in detail.

How Requests are Routed

Request routing takes place in three parts, based on information in the IOR and information maintained by Oracle Tuxedo CORBA. The numbers in the following figure illustrate the routing.

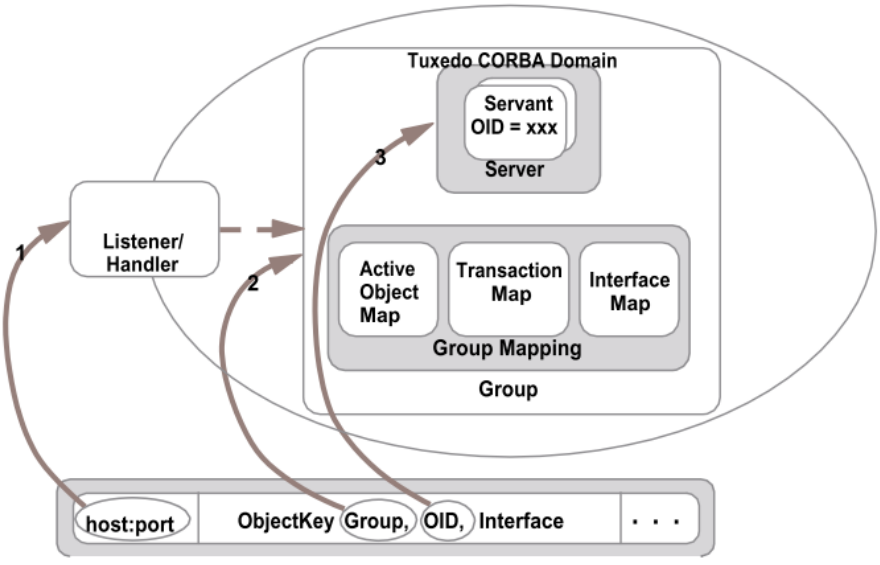

Figure 2. Request Routing in Oracle Tuxedo CORBA

- Host:port – The host:port information in the IOR directs the request to a specific IIOP Listener/Handler (ISL/ISH)
- Server group – The group information in the ObjectKey of the IOR directs the request from the listener/handler (ISL/ISH) to a specific server group.
- Let's examine this step, which we will call server load balancing, in detail. Steps 1 and 2 have routed the request to the appropriate server group. There are three scenarios within the server group for choosing a server process.
  - Is the object active? – The AOM is checked to see if the object is active in this group. Recall that an IOR is tied to a specific server group, so the object must be in this AOM. If the object is active, the AOM will hold an entry for the object that points to the server process having that object active in memory.
  - Is the object in a transaction? – If the object is not already active, the next check is to see if the object is part of a transaction. Oracle Tuxedo CORBA applies a concept called transaction affinity to the selection of the server to minimize the transaction commit tree. Because the transaction affinity algorithm is part of a software patent applied for by Oracle, it is not discussed here.
  - Choose the best available server – The object is not active or is in an existing transaction so Oracle Tuxedo CORBA is free to choose the best server from all available servers in the server group offering this interface. This information is maintained in the interface map. This choice is made based on determining the server queue with the smallest quantity of work and placing the request on that queue.

    In all cases, the request is placed on the queue for a specific server process.

  Notice that directing a request to a specific server process would not work for Multiple Servers, Single Queue (MSSQ) sets. The characteristic that makes MSSQ sets attractive for stateless services also makes them unusable for stateful objects. With an MSSQ set, several server processes pull requests from the queue independently. All servers are equal, so which server process handles a request does not matter. Furthermore, there is no way to direct a request to a specific server process. Because a request for an active object must always be sent to the specific server process having that object's state in memory, MSSQ sets are not available for Oracle Tuxedo CORBA servers.

Load-Balancing Parameters

In Oracle Tuxedo CORBA, as in Oracle Tuxedo ATMI, several configuration parameters influence the selection of a server queue within a server group. The following UBB configuration parameters affect load balancing in Oracle Tuxedo applications:

- LOAD
- NETLOAD
- LDBAL
- SCANUNIT and SANITYSCAN

LOAD is a number between 1 and 100 representing the relative load that the CORBA interface is expected to impose on the Oracle Tuxedo system. The numbering scheme is relative to the LOAD numbers assigned to other CORBA interfaces used by this application. The default is 50.

The NETLOAD parameter affects the load-balancing behavior of an Oracle Tuxedo system when a service is available on both local and remote machines. In Oracle Tuxedo CORBA, server load balancing occurs only within a server group, so this parameter does not affect load balancing within Oracle Tuxedo CORBA server groups.

The LDBAL parameter in the RESOURCES section together with MODEL setting determines how the load balancing algorithm works. Please refer to the other white paper "Load Balancing in Tuxedo ATMI Applications" for detail.

The SCANUNIT and SANITYSCAN numbers multiplied together determine the number of seconds between recalculating the work-queued value in the BB for each queue. There are two different modes for load- balancing calculation based on the MODEL setting in the RESOURCES section of the UBB configuration file. If the setting is SHM, the real-time algorithm is used, if the setting is MP, the periodic algorithm is used.

Obviously, the most interesting parameter for Oracle Tuxedo CORBA is LOAD. From the previous discussion, we know that the LOAD parameter setting will only affect load balancing within a server group, and only when the object is not currently active. Oracle Tuxedo CORBA will use LOAD to choose the least busy server among all the servers offering the requested interface. Each server has a request queue. The total work load, or work queued of a queue, is used to determine the least busy server. The work-queued value is the sum of the load for each request on the queue, which is the LOAD factor of the interface of each request on the queue. (The work-queued values of the queues are stored in the BB.) The work-queued value of a queue is increased when a new request is added to the queue, and decreased when the server completes the request.

Oracle Tuxedo CORBA will place a new request on the queue having the smallest load. If there is a tie between one or more queues, then the first server in the list will be chosen. This choice optimizes the chance that the server's working set is already in the cache.

The situation where there are no requests on any queues is also a tie, so when no requests are active, the first server in the list will always be chosen. For this reason, in an unloaded environment, you will not see an even distribution of work among all the servers, but rather you will see one or two servers doing most of the work. Even though the configuration might have several servers for the same interface, the first one will always be chosen if none are busy. Load will not be distributed to the second server until a request arrives while the first server is busy. Work will not be distributed to the third server until a request arrives while both the first and second servers are busy, and so on.[2]

[2] The choice of the first server works well for BEA Tuxedo ATMI services, and for method-bound BEA Tuxedo COR BA objects. However, for stateful objects, a side effect is that objects may tend to cluster on the first server in a lightly loaded environment. In the future, a different algorithm may be considered for transaction - or process-bound objects.

Determining Interface Load Based on Operations

Objects are a combination of state and interfaces. Interfaces are composed of one or more operations that typically act on the object's state. Not all operations on an interface are equal, that is, not all operations represent an equal load. The load of an operation is based on the frequency of the operation and the cost (time and resources) of executing the operation. It is typical to have within the same interface one or two operations that are frequently performed and several operations that are infrequently performed. The load of an operation is approximately the cost of the operation multiplied times the frequency percentage.

The LOAD factor defined in the UBB configuration file and used in queue load calculation applies to the entire interface. Thus, the interface LOAD factor should represent an average of the loads of all the operations of the interface. We can calculate the load of the interface by summing together the load multiplied times the usage for each operation as:

```
LOAD(interface) = ∑ (LOAD(operationN) * percentage(operationN))
```

Grouping of Interfaces within Servers

A server is capable of providing one or more interfaces. However, for each server, there is only one request queue. Thus, requests for different interfaces will be mixed together in the same queue. If some interfaces take a long time to execute, they could block smaller operations from executing quickly, which may not be a problem. However, you should give some thought to how interfaces are grouped together in servers, keeping in mind that all requests to a server will be funneled through the same queue.

Example

We will use the following scenario to illustrate how the LOAD determines which server will be selected. Our simple example will use two identical servers, one with a work request queue named Q1, and the other with a work request queue named Q2. Each server can satisfy requests for three interfaces, IA, IB and IC. IA is defined by the following IDL:

```
interface IA {
void operation1 (in param1, out param2);
void operation2 (in param1, in param2);
void operation3 (out param1, out param2);
};
```

In interface IA, operation1 is called 70% of the time, operation2 is called 20% of the time, and operation3 is called 10% of the time. Operation1 has a relative load of 30, operation2 has a load of 40, and operation3 has a load of 80. We calculate the load of the interface by summing together the load multiplied times the usage for each operation, so for interface IA the load is: 37 = (30 * .70) + (40 * .20) + (80 * .10). For interface IB the load is 50, and for interface IC the load is 60. Assume the following queue configurations:

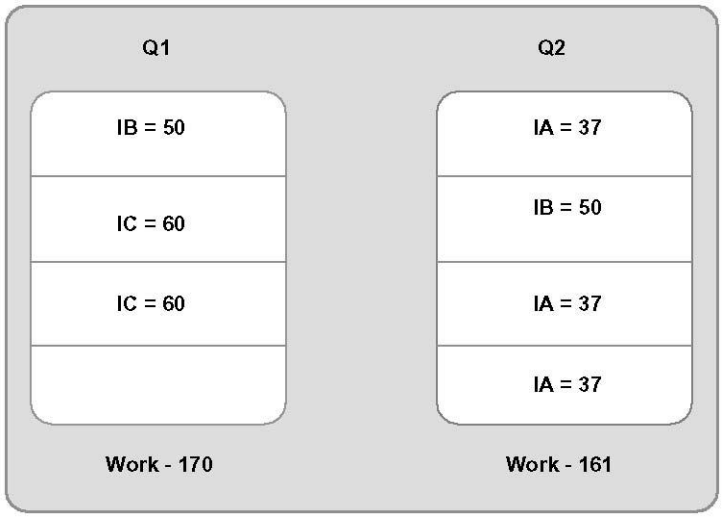

Figure 3. Calculating Work Queued.

The work-queued value is calculated as: Work Queued = S (loadN). For Q1, we calculate the work-queued value as 170 = (50 + 60 + 60); for Q2, we calculate the work-queued value as 161 = (37 +50 + 37 + 37). The next request for either server will be queued on Q2 because it has the lowest work-queued load, even though there are more requests on Q2.

Factory and Factory Finder

Factories and the Factory Finder play an important role in load balancing, and are key to distributing the load between server groups. The role of factories in Oracle Tuxedo CORBA is to create object references, or IORs, on behalf of CORBA clients, that direct the client requests to Oracle Tuxedo CORBA objects. There are no requirements about the uniqueness of object references. In other words, the factory is free to give the same object reference to every client, or to create a unique object reference for each client, or anything in between. Recall that the key components of the IOR for Oracle Tuxedo CORBA routing are the OID, group, and interface. The factory is asked to create an object reference for a specific interface; the other parameters, OID and group, influence the load balancing among different servers that offer the same interface.

The ObjectID

The factory has direct control over the assignment of the OID. The OID assigned to an object must accomplish two goals:

- Identify the state of the object
- Determine the uniqueness of the object

Identifying the state of an object must satisfy application requirements without regard to Oracle Tuxedo CORBA infrastructure. A typical scenario is to use a key to a record in a database as the OID. When the application servant is called by the activate object() routine, the application servant is passed the OID, which is used to retrieve the object's state from the database record identified by the OID. Because the object is tied to a specific database record, only one instance of the object exists within the application.

Another scenario is a process object, such as described by the Oracle Tuxedo CORBA Process-Entity design pattern; for example, the Teller object in Bankapp, or the Registrar in the Oracle Tuxedo CORBA University sample application. In this scenario, the process object is not tied to any specific state of its own; instead, it acts on entities that are identified in each request to the process object, and that are associated with their own state (we will make this assumption for simplicity, although more complex scenarios will have state associated with the process objects). The application needs to scale to thousands of simultaneous clients, and thus needs to have many (hundreds or thousands) of process objects active simultaneously. The important difference between this scenario and the "typical" scenario described previously is not the state associated with the object, but rather the number of different instances of the object that can exist simultaneously.

Controlling the number of concurrent instances of an object is the second goal of the OID, and relies on the characteristic of the Oracle Tuxedo CORBA infrastructure for (1) the guarantee of uniqueness and (2) the distribution among instances. Let's review the Oracle Tuxedo CORBA guarantee that is important here: Every request to the same object identity will be delivered to the same object instance. Also, recall that in Oracle Tuxedo CORBA the object identity is determined by the IOR, which is created by the factory.

Assigning the OID

If an application wants only one instance of an object, the application should assign a single, unique OID to that object, and use the same OID for every object reference handed out by the factory. Oracle Tuxedo CORBA guarantees that all requests to that object reference will go to the same instance. Similarly, if an application needs many instances of an object, the application must assign a different OID for each instance required.

In the scenario where there are an unlimited number of process objects having no state, the application could create a unique ID for the OID for each object reference created by the factory. A variation on this behavior is where there are a limited number of objects that can be used simultaneously. For this scenario, the application could create one OID for each instance of the object desired. The factory would then cycle through these OIDs when handing out object references. In the case where the object has state but the application also needs multiple instances of an object, a composite OID, as shown in the following figure, can be used. Obviously, there are many other designs that would achieve similar results.

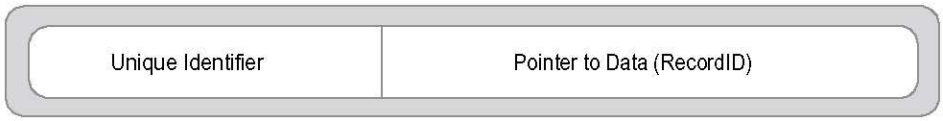

Figure 4. Components of a Compound OID.

System Configuration Implications

There are several other factors to consider when designing how OIDs will be assigned. For example, let's examine the scenario of the compound OID. In this scenario, there will be a separate instance of the object for each OID assigned. However, each object instance will use the same state, that is, multiple objects will be accessing the same data simultaneously. The Oracle Tuxedo CORBA application must provide the concurrency control for this data (potentially using the features of a database) because Oracle Tuxedo CORBA does not automatically provide any type of concurrency control for the application. Where the data is mostly read-only, this scenario works very well.

There are many different possible configurations of factories in an Oracle Tuxedo CORBA application, such as:

- Factory per server
- Factory per group
- Factory per domain

The difference between these configurations is the scope of the object references created and the interfaces for which the factory can create objects. The simplest configuration is to have one factory per server. That factory can create object references for all of the interfaces provided by that server. Typical in Oracle Tuxedo CORBA is to have multiple copies of the same server running at the same time, which implies that multiple copies of the same factory are running simultaneously. An application must keep this fact in mind when assigning OIDs.

To emphasize this latter point, consider the scenario of many concurrent process objects running simultaneously. In this scenario, we want our OIDs to be relatively unique. A simple algorithm to accomplish this uniqueness would be to use a long word for the OID and to increment the long word counter each time a new object reference is assigned. As long as a client generally uses an object for less time than the time required to wrap the long word counter, this algorithm appears to work. What happens when 10 servers, and thus 10 factories, are running simultaneously? Because each server is running the same executable, each factory is using the same algorithm to assign IDs. As a result, potentially 10 object references can exist with the same ID at the same time. A better algorithm would be to use a generated unique number for the OID, which would guarantee that each object reference referred to a different instance of an object, regardless of which factory created the reference.[3]

The previous example demonstrates the interdependency between multiple factories. The algorithm that assigns OIDs must take into account the fact that multiple factories will be running the same algorithm simultaneously. In some applications, the factories will need to access data to assign IDs; the application must coordinate the access to data from the different factories. One way to simplify this coordination and to reduce data contention is to have factory servers that are responsible for creating objects references for other servers, either within the same group (factory-per-group configuration) or within the domain (factory-per-domain configuration). To minimize contention and avoid a single point of failure, you must have more than one of these factory servers, so the application must still deal with data concurrency control between the factory servers, but the problem is greatly reduced with only a few factories.

You need to consider one other area of coordination. In Oracle Tuxedo CORBA, factories create object references and thus assign the OID. Servants implement the business objects and use the OID to save and restore an object's state. In some cases, the factory and the servant are not in the same process, or are not even in the same server group. The factories and the servants must have the same understanding of application OIDs. This understanding includes the relationship of the OID to the object state and the concurrency requirements of multiple object instances having the same state.

[3] As an example, a unique identifier algorithm is supplied by the DCE uuidgen algorithm and is available on most UN IX and all Windows platforms. This algorithm is available through an API call or through a standalone uuidgen utility program invocable from a command shell. The BEA Tuxedo COR BA TP Framework will also be adding a convenience routine for generating unique identifiers. A simple technique for almost uniqueness is to use a process ID (PI D).

Assignment of Group

The remaining part of the ObjectKey that determines an object identity is the server group. The application does not have direct control over the assignment of the group but can have indirect influence. A factory creates an object reference by calling the TP::create_object_reference() method, passing in the interface, the OID, and the criteria. The TP framework calls an internal infrastructure notifier component to create the object reference. By default, that is, with no criteria specified, the ObjectKey inherits the server group of the factory that creates it. With criteria specified (also called factory-based routing), the infrastructure uses the routing table to determine the group.

Recall that the uniqueness of an object is determined by the OID and the group (and the interface). Therefore, an object of interface A, with OID x in group 1, is not the same object as interface A, with OID x in group 2. An application must be aware of this fact when distributed across groups. However, distribution of groups is generally a deployment configuration decision. An application should typically allow for being spread among groups. If an interface cannot be distributed among groups, the application administrator must know this restriction.

Factory Finder Load Balancing

When a server is initialized, it creates object references for its factories and registers them with the factory finder by calling the TP::register_factory() method. The server creates the factory object reference just like a factory does, by calling TP::create_object_reference(). Thus, the server is responsible for specifying the OID of the factory, or in other words, the uniqueness of the factory object reference. Typically, the server will not specify criteria when creating a factory object reference, so the factory will have the same group as the server.

The Factory OID

When a client requests a factory from the factory finder, the factory finder looks for all the factories that match the specified name and then selects one of them using a round-robin algorithm. The factory finder allows, in fact expects, many factories with the same name.

Let's revisit the example where 10 copies of the same server are running, and thus 10 copies of the same factory are registered with the factory finder. The first request for a factory is given the first IOR on the list. The next request is given the second IOR, and so on, in a round-robin fashion. What effect does the OID of the factory IOR have on load balancing? An implementation might simply assign a constant for the OID. Since each instance of the server is just a copy of the same executable, each will use the same algorithm to create the factory IOR. Even though 10 factories are registered with the factory finder, each IOR has the same interface, group, and OID, so there is in fact only one IOR. Every client request will be returned the same IOR from the factory finder. Although 10 servers are available to service requests to the factory, since there is only one object, only one server will ever be used at a time, which is probably not the intended result.

If, instead, the server used a unique ID for the OID when creating the factory object reference, then every factory IOR would be different. So with 10 servers, there would be 10 different IORs (with 100 servers, there would be 100 different IORs). Now, each client request would get one of 10 different IORs, selected in a round-robin fashion. However, notice what we have actually done here. Each factory object reference is for a specific interface and a specific OID. Each factory is capable of providing the same interface, which is ultimately for creating some application object. If the OID is not active, we use the load balancing described in "Load-Balancing Parameters" to choose an available server. So the first client request will go to the first factory, regardless of which server actually created the IOR. The second client request will go to the best available server. If the first factory is not busy, the second client request will also go to the first factory. We have not tied any specific client request to

any specific factory. Instead, we have made it possible for up to 10 client requests to be processed simultaneously.

An application might need objects that model the login session of a client, where each object must maintain the client session context. Activating and deactivating the state may be a costly operation. In this case, the application needs to distribute many persistent, application-controlled deactivation objects among the available servers. The last discussion assumed that the factory had an activation policy of method-bound. We could, however, define the activation policy of the factory to be process-bound and force the factory to become active in the server at the time that the factory was created. These modifications to the previous scenario would force client requests to factories to round robin among each of the 10 servers. If we also pinned the application object to the server at the time that the object was created by the factory, the application objects would also be distributed among the 10 servers. The cost of these changes would be the resource costs of keeping the factory object in the memory of each server, when probably only one or two factory objects would ever be needed at the same time. A different way to achieve this result would be to establish 10 server groups each with a separate factory.

Groups

Many applications will need to distribute servers across multiple machines for increased throughput. In Oracle Tuxedo CORBA, this means distributing the servers across different groups. Let's modify the previous 10-server example to see how it works across groups (remember that each server has both a factory and an object implementation in it).

Imagine that we have 2 groups, each with 5 servers for a total of 10 servers. The first group of servers (group 1) will come up and create object references for their factories. The server uses a unique number for the OID, and the IOR inherits the group from the server. The server then registers the factory and IOR with the factory finder. The same sequence happens with the second group of servers. Now, 10 factories are registered in the factory finder. All ten factories have the same interface, all have a different OID: 5 factories are for group 1, and 5 factories are for group 2. As clients request factories from the factory finder, references will be given out in a round-robin fashion, so half the clients will get IORs for group 1, and half will get IORs for group 2. Remember the three stages of request routing that take place in Oracle Tuxedo CORBA? First, the host:port directs the request to a listener/handler (ISL/ISH). Next, the group is used to go to the specific server group. Finally, the interface and OID are used to find the server. Therefore, half of the requests will go to group 1 and get load balanced among the servers there, and half the requests will go to group 2.

Perhaps the machine for group 1 is twice as powerful as group 2. We could improve the load balancing by changing the number of servers we start in each group. If group 1 has 10 servers, then 10 factories will be registered with an IOR for group 1. Group 2 still has 5 servers, so 5 factories will be in the factory finder for group 2, for a total of 15 factories. Again, the factory finder will return factory IORs in a round-robin manner, so twice as many clients will get IORs for group 1 as for group 2.

This behavior is Oracle Tuxedo CORBA's automatic (default) method of load balancing among groups, which works well for many applications. Of course, default mechanisms are never sufficient for every application, so an infrastructure must also provide a mechanism for applications to control their own destiny. If an application's objects are tied to some sort of data, the different machines need to coordinate access to the database, which could be a bottleneck in some cases. One approach is to partition the data so that half of the data is serviced by group 1, and half by group 2. Now, the application needs a way to assign a specific group to a specific IOR.

Note that factories are just a special case of an application object. The main difference is that factories are registered with the factory finder, which uses a round-robin algorithm to distribute references to them. However, the principles just discussed relating OIDs to the number of concurrent instances that can run at the same time also apply to application objects.

Factory-Based Routing

Factory-based routing provides applications with the ability to specify what group is associated with an IOR. The factory creates object references using the TP::create_object_reference() method. The last parameter to this method is the criteria. The criteria are specified in the form of a CORBA NVList, or list of name-value pairs. When the infrastructure is creating the object reference, it compares the values in the criteria list against ranges of values in the routing table. When a match is found, the group specified in the table for that value is assigned to the IOR. The IOR does not inherit the group of the factory that created it. The list of names, values, and groups that make up the routing table is defined in the UBB configuration file, similar to how data-dependent routing is defined in Oracle Tuxedo ATMI.

The factory can determine the criteria in a number of different ways. The criteria may be based solely on information available to the factory during the request, such as the time of day. The criteria may be passed in as part of the request from the client to create the application object. Or, the factory may take the input from the client and process it in some way to determine the criteria. In any case, the factory only knows what criteria to pass on to the infrastructure. How the criteria is evaluated and what group is chosen is determined administratively by the values in the routing table, which means that an administrator can change the routing by reconfiguring the table, without changing any interfaces or application code.

In the last example, we wanted to divide a database between two servers so that requests for half of the data went to one server, and requests for the other half went to another server. Assume that the application object is for a teller and that the database is partitioned based on the branch location for the teller. The client (the teller, not the customer) would get a teller factory from the factory finder and ask it for a teller object, passing in the branch as an input parameter. The factory would put the branch into an NVList and pass it as the criteria parameter to the TP::create_object_reference() method. The infrastructure would compare the branch in the NVList to the routing table and assign the correct group to the IOR. At a later date, the administrator could move the branch data from one partition to another, or add another partition to the database. Simply changing the routing table would cause new teller objects to be created for the new application configuration; this outcome would be accomplished without any need to change the application code.

Sometimes an application factory will take input from the client and process it to derive the criteria, which may involve retrieving information from a database or other external processing. In this case, the factory- per-domain configuration may be appropriate, so that the factories that do this processing are limited to specific (possibly dedicated) servers. Of course, to avoid a single point of failure, you should always have more than one such dedicated factory server. Factory-based routing also provides a mechanism for an application to provide its own load balancing. For example, the factory could monitor application specific information and use it to make routing decisions for assigning groups to IORs.

An application design that uses factory-based routing must take into consideration the fact that there will be servers in multiple groups when assigning OIDs.

Load-Balancing Configuration Patterns

This section describes some common patterns of application configurations for load balancing. Each pattern is described in the following format.

- Description
- Motivation
- Applicability
- Settings
  - OID assignment
  - Factory OID assignment
  - Activation policy
  - Factory activation policy
  - Group/factory configuration
- Other considerations
- Variations

Some common variations are described for each pattern. However, many other patterns and variations are possible by altering the basic settings (parameters) of each pattern.

Singleton Object

- **Description**: The singleton object pattern supports applications that have one or more objects for which there is only a single instance of the object. This pattern is also the easiest to implement and can be used when first developing a prototype.
- **Motivation**: This pattern provides load balancing for the initial creation of the singleton object. All future invocations to the object will go to the same server. A singleton object may be indicated when the cost to save and restore the state of the singleton object is expensive and/or when the concurrency control on the object's data prohibits sharing.
- **Applicability**: This pattern is applicable when there are a small number of objects that need to be accessed frequently within an application.
- **Setting**:
  - **OID = Constant** – Assign a constant value for the OID. It may correspond to a key for a database record. Every object reference returned for this object must have the same OID.
  - **FactoryOID = Constant** – Assign a constant value for the OID of the factory. This will limit the number of concurrent factories to one, although there may be more than one server capable of providing the interface.
  - **ActivationPolicy = Process** – Because there is only ever one instance of the object, it is not necessary to continually save and restore the state of the object between invocations. In addition, the cost of saving/restoring state may be expensive for a singleton object. If the cost of saving/ restoring the object is not expensive, other policies may be considered.
  - **Factory Activation Policy = Method** – Since there is only one object, and it is only created once, the factory object should not be consuming resources.
  - **Configuration = Single Group** – The singleton object is limited to operating in a single group. There is no requirement for having the available servers span groups.
- **Other Considerations**: Because this pattern suggests the use of process-bound objects (activation policy), each singleton object will consume a certain amount of resources. If this resource is a lot, it can limit the amount of resources available for other objects. One approach is to dedicate a server to the singleton object (or a small set of objects) and process other objects that have to scale to a large number on other servers. Another approach is to use the method-level activation policy.

Variation 1 - Small number of objects

This pattern works when there is not a singleton object, but rather a small number of objects. In that case, the OID can be limited to a small number of constants, rather than a single constant. The factory should hand the small number of OIDs out in a round-robin fashion. If there is one factory per server, and each factory creates a single OID, the round-robin algorithm of the factory finder will distribute client requests among the small number of objects. If the objects point to the same data, the application must implement the concurrency control.

Variation 2 – Pinned object

A variation on the singleton object is when the singleton object must execute within (be pinned to) a specific server. To accomplish this, the server initialize method should do the following: Create and register the factory, get a reference to the application object from the factory, and invoke a method on the applica- tion object. In Oracle Tuxedo CORBA, an object will always be created in the same server in which the request was initiated, if possible. The application object will be pinned to run in the server that created it, assuming that the server implements the object's interface. Alternately, the factory could invoke a method on the object before handing it out, which would pin the object to the same server that has the factory. To be pinned to a specific server, the object needs to have a process-level activation policy.

Process-Entity Objects

- **Description**: The process-entity pattern complements the Oracle Tuxedo CORBA Process-Entity design pattern and TP Framework.
- **Motivation**: This pattern provides load balancing between every method invocation from every client. This type of balancing supports large-scale parallelism in processing, that is, tens of thousands of clients simultaneously using hundreds or thousands of "process" objects.
- **Applicability**: This pattern supports a large number of concurrent users accessing a large number of "stateless" process objects. This works especially well when the process objects access mostly read-only information.
- **Setting**:
  - **OID = Compound** – A compound OID is used. An important aspect of the OID assignment is that it must be unique. The compound nature of the OID allows the application to assure uniqueness and still tie the object reference to specific data.
  - **FactoryOID = Unique** – A unique OID is required for every factory.
  - **ActivationPolicy = Method** – Method-level activation policy allows objects to be swapped in and out of memory for maximum concurrency.
  - **Factory ActivationPolicy = Method** – Method-level activation policy allows factories to be swapped out of memory for minimum resource utilization and maximum concurrency.
  - **Configuration = Multiple Groups, n Factories per group.** In this configuration, there should be n factories per group, where n is in relation to the number of servers in each group (in relation to the processing power of group). The easiest way to do this is to have a factory-per-server configuration, but other configurations are possible.
- **Other Considerations:** Because the state of the process object is saved and restored on ever method invocation, care must be taken to implement an efficient mechanism for this. For example, only state that has actually been changed should be saved on deactivation.

Variation 1 – Session Oriented Process Objects

The OID of the process object can be tied directly to the entity state. This is a scenario where a client gets an object reference for an application process object (such as customer) where the OID defines the client's session context (such as customer ID). The application object still acts as a mediator for the entity object and still has the method-level activation policy. The difference is that the process object is tied to a specific entity and does not operate on many different entities.

Variation 2 – Parallel Objects

Support for parallel objects became available with Oracle Tuxedo CORBA 8.0 as a performance enhancement. This feature enables the designation of all business objects in a particular application as stateless objects. The effect is that, unlike stateful business objects, which can only run on one server in a single domain, stateless business objects can run on all servers in a single domain. The benefits of parallel objects are:

- Parallel objects can run on multiple servers in the same domain at the same time. Utilization of all servers to service concurrent multiple requests improves performance.
- When the Oracle Tuxedo system services requests to parallel business objects, it always looks for an available server to the local machine first. If all servers on the local machine are busy, Oracle Tuxedo chooses the best server among all the candidate servers including those on remote machines based on their work load. Thus, if there are multiple servers on the local machine, network traffic is reduced and performance is improved.

Session Objects

- **Description**: A session object is an object that is tied to some specific state, and is used by a client for a series of interactions. This pattern is similar to the EJB Stateful Session Object. However, unlike EJB, the state and identity of Oracle Tuxedo CORBA session objects can exist beyond the life of the session and survive server or system failures.
- **Motivation**: This pattern provides load balancing among these objects for the creation of the session. All future interactions are then tied to that specific object for the duration of the session. This pattern supports many concurrent conversational interactions with clients where each "session" is tied to some specific data.
- **Applicability**: This pattern is applicable to applications where clients have a series of interactions with the server object, where state must be preserved between interactions, and where saving/restoring state on each method is expensive or impractical.

- **Setting**:
  - **OID = Based on data** – The OID is related directly to the data that is associated with the object. There is one object for each data entity. The application may choose to give out multiple object references with the same OID, but only 1 client will be able to hold a "session" at a time.
  - **FactoryOID = Unique** – The OID for the factory should be unique, so that clients will get distributed to multiple factories simultaneously.
  - **ActivationPolicy = Transaction** – The activation policy should reflect the life of the session. Either demarcated transactionally by using the transaction policy, or under application control using application control deactivation.
  - **Factory ActivationPolicy = Method** – Method-level activation policy allows factories to be swapped out of memory for minimum resource utilization and maximum concurrency.
  - **Configuration = Spans groups, FBR** – This pattern can span groups to provide parallelism in processing and potentially application partitioning. Factory-based routing should be used to assign client requests to appropriate server groups.
- **Other Considerations:** Once a client gets a reference to an object, that object will be dedicated to that client until the session is ended. Parallelism can be achieved by handing out many similar objects. Where the objects share the same state, the application may use a compound OID and provide the concurrency control.

Conclusion

Designing large-scale distributed object systems is difficult. The Oracle Tuxedo CORBA framework helps simplify the task by providing a scalable infrastructure for state management, but you, as an application developer, must still make basic architectural decisions for each application. A one-size-fits-all policy is not an effective mechanism for building large-scale, mission-critical applications. Instead, you must decide how to achieve the correct amount of load balancing for each of the different objects within an application. Your application must consider how to assign OIDs for both the application objects and factories. You must understand the relationship between an object's activation policy and how requests to that object will be routed. You must understand how the factory finder and factory-based routing affect distribution among groups. Finally, you must understand how the LOAD parameter in the UBB configuration file affects server choice and how to achieve a realistic value of load for all the operations in a given interface. Some decisions, such as the assignment of the OID, are solely the responsibility of the applications. In other cases, Oracle Tuxedo CORBA provides useful default implementations for load balancing, and provides the flexibility to tune your application to achieve optimum load balancing.

This paper was written to explain the interdependencies that you must consider to achieve the level of load balancing appropriate for your application. "Load-Balancing Configuration Patterns" presents some patterns that take all these criteria into consideration. Those patterns should serve as a starting point for your application design.

Load Balancing in Oracle Tuxedo CORBA Applications
July 2012

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200

oracle.com

Oracle is committed to developing practices and products that help protect the environment

Hardware and Software, Engineered to Work Together