Coherence*Extend

Objectives

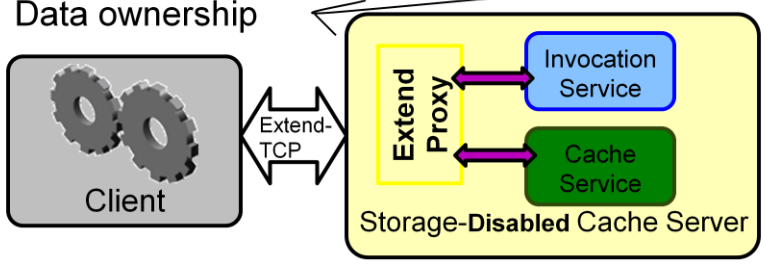
After completing this lesson, you should be able to describe, configure, and deploy a Coherence*Extend.

13 - 2

What Is Coherence*Extend?

- Coherence*Extend:
  - Extends the reach of the core Coherence TCMP cluster to a wider range of consumers
  - Allows applications to access Coherence clusters without being a member of the cluster
- With Coherence*Extend, there is no:
  - In-process transport
  - Cluster and service membership
  - Data ownership

Local caches are not considered to have data ownership.



13 - 3

Coherence*Extend extends the reach of Coherence to applications that are not well-suited to run within the Coherence cluster. It provides a rich client API to the grid without any of the overhead associated with being a cluster member. Coherence*Extend clients do not need to concern themselves with partitioning, data ownership, or cluster membership. Typical uses of Coherence*Extend include:

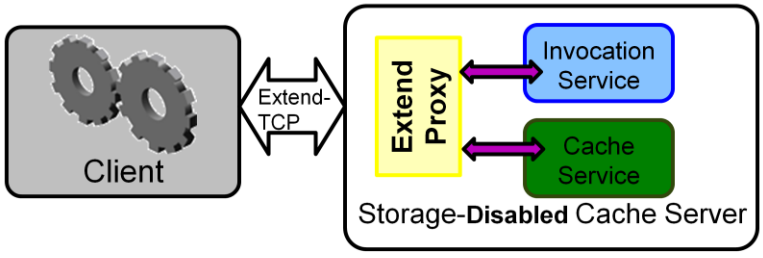
- Providing desktop applications with access to Coherence caches (including support for near cache and continuous query)
- Coherence cluster “bridges” that link together multiple Coherence clusters that are connected via a high-latency, unreliable WAN

Coherence*Extend Clients

There are three types of clients for Coherence*Extend, allowing cluster access from other languages over TCP/IP:

- Java
  - Included with main Coherence product
  - Uses same Java API as clustered clients
- C++
- .Net

Separate download and similar API. No Java required.

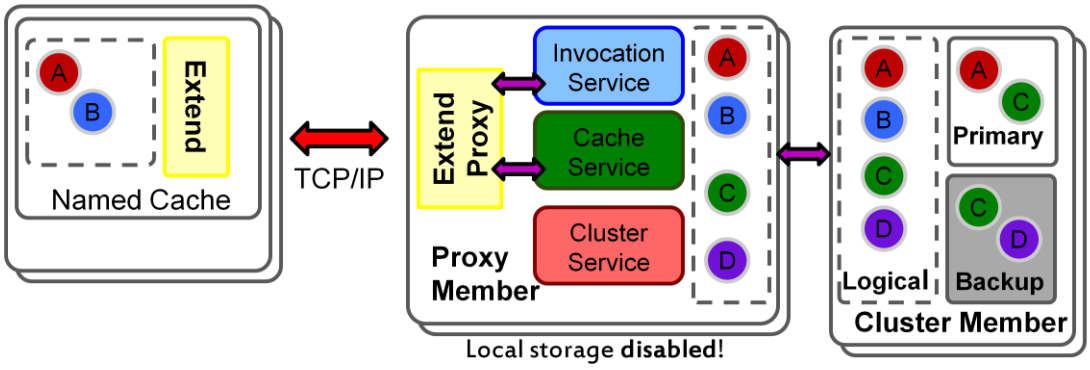


13 - 4

- **Java:** The Coherence*Extend Java client is included with the main Coherence product. It uses the same Java API that the clustered Java client programs use, but all calls are implemented using TCP/IP instead of TCMP.
- **C++:** The Coherence*Extend C++ client is available as a separate download. The client API follows the interface and concepts of the Java client, so users familiar with the Java API find it easy to code C++ clients. A `NamedCache` instance is retrieved by using the `CacheFactory::getCache(...)` API call. After it is obtained, a client accesses the `NamedCache` in the same way as it would if it were part of the Coherence cluster.
- **.Net:** The Coherence*Extend .Net client is available as a separate download. The client API is also easy for Java developers to understand. An `INamedCache` instance is retrieved by using the `CacheFactory.GetCache(...)` API call. When it is obtained, a client accesses the `INamedCache` in the same way as it would if it were part of the Coherence cluster. The .Net product also includes a session state store provider implementation for storing ASP .NET HTTP session objects in the data grid.

Coherence*Extend Architecture

- Native Clients:
  - Run outside the cluster
  - Make Request using a Proxy Service
  - Use TCP/IP as a transport
  - Use language-independent serialization
  - Use a proxy that delegates requests to Coherence cluster



13 - 5

Architecturally there are two main components to Coherence*Extend:

- A native client
- Proxy Service

The native client routes all requests through a supported transport binding to the Proxy Service, which in turn delegates the requests to a clustered service. Coherence*Extend supports a TCP/IP transport binding. Communication can also be secured using SSL, which is covered in the lesson titled “Coherence Administration.” There are Java, C++, and .NET clients available for Coherence*Extend. As such, Coherence*Extend clients use Portable Invocation Format (PIF) for remote invocation, and Portable Object Format (POF) for serialization because they are language-independent solutions. Java clients may also use Java (or `ExternalizableLite`) serialization. Obviously, these clients will not be able to share data with .NET and/or C++ clients, but `ExternalizableLite` is an option in a Java-only deployment.

The image in the slide illustrates Coherence*Extend’s architecture: The Coherence*Extend .Net, Java, or C++ client communicates with the Proxy Service running inside the Coherence cluster, using TCP/IP to access cached data inside clustered `NamedCache` objects.

Coherence*Extend Capabilities

Support for most facilities available to a TCMP client node:

- Life cycle (`MemberListener`) *
- Caching (`CacheMap`)
- Query (`QueryMap`)
- Events (`MapListener`)
- `InvocationService` *
- Aggregation and Processing (`InvocableMap`)
- Near Caching (`NearCache`)
- Cache Views (`ContinuousQueryCache`)
- SSL

With some restrictions!

13 - 6

Coherence*Extend client supports most of the capabilities offered by clustered clients.

Restrictions

- Life cycle (`MemberListener`): Coherence*Extend clients cannot witness (get callbacks) from Cluster Member events, except for the proxy to which they are connected. They can only register a listener on the proxy server, but not other members in the TCMP cluster.
- The `InvocationService` works normally when using Coherence*Extend. The caveat is that only the synchronous invocation model is available when using Coherence*Extend.
- SSL allows encrypted communication between Coherence*Extend clients and the cluster. Encryption Filters are going to be deprecated.

Transactions are not yet supported with Coherence*Extend.

TCMP stands for Tangosol Cluster Management Protocol, but that name was coined when the product was part of the Tangosol company, before the merger with Oracle.

Coherence*Extend Advantages

Advantages of using Coherence*Extend include:

- Direct access to the data grid from non-Java applications
- Access to multiple data grids from a single application
- Lightweight native client library
- Designed with transient applications in mind
- Designed with end-user machine environments in mind
- Relaxed memory and CPU requirements
- Relaxed network requirements
- Firewall-friendly transport bindings
- Support for numerous client applications

13 - 7

The slide lists some of the advantages of using Coherence*Extend. In particular, note that Coherence*Extend supports many clients connecting to a cluster. Coherence*Extend clients do not become a part of the cluster's TCMP communication, and therefore do not cause the extra overhead to the cluster as a whole. This allows Coherence to support tens of thousands of connected clients easily.

Coherence*Extend Disadvantages

Disadvantages of using Coherence*Extend include:

- No direct view of cluster and service membership
- No direct knowledge of data ownership
- Limited direct invocation support
- No asynchronous invocation support
- Limited clustered locking facilities
- No support for client-side transactions
- Extra network hop
- Connection-based design

13 - 8

The slide lists some of the disadvantages of using Coherence*Extend. It would be possible to bypass some Coherence*Extend limitations by invoking an invocation service on the Proxy Server with an agent that could perform work against the caches of the cluster. When the agent is executed on the Proxy Server, that code has the full visibility of the cluster that a cluster member has and is able to harness the full power of the cluster. For example, normally a Coherence*Extend-based invocation service can only invoke a service on the Proxy Server, because that is the only cluster member it can see. However, if the Coherence*Extend client were to invoke an agent on the Proxy Server that then executed another invocation service of its own, it could then invoke the service on any Coherence node in the cluster.

Note that Java and .NET support SSL, C++ does not.

When to Use Coherence*Extend?

- “Fat” client applications deployed on desktop machines
- Non-Java access to the data grid
- Access to the data grid over a high-latency, unreliable, heterogeneous, restrictive network (across switches, through firewalls, WAN, mixed bandwidth, and so on)
- Coherence Incubator project patterns. For example: Cluster connectors (hot/warm and hot/hot disaster recovery strategies, regional data access)

13 - 9

Coherence*Extend extends the reach of the core Coherence TCMP cluster to a wider range of consumers, including desktops, remote servers, and machines located across WAN connections. Typical uses of Coherence*Extend include:

- Providing desktop applications with access to Coherence caches (including support for near cache and continuous query)
- Coherence cluster “bridges” that link together multiple Coherence clusters that are connected via a high-latency, unreliable WAN

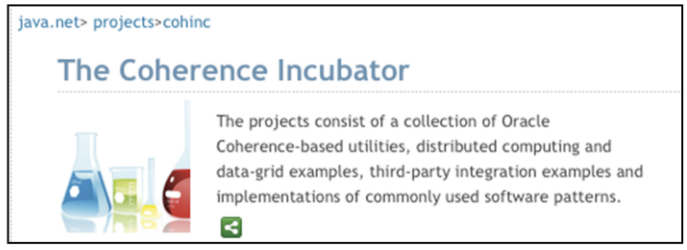
Coherence*Extend is the means to connect an application to the cluster. It supports “fat client” real-time applications, such as trading desks, as well as other server tiers. It provides almost all the features that a full cluster member can provide, including near-caching, event listeners, and the ability to execute `EntryProcessors`. The connection is over TCP/IP. It also provides the capability of continuous query, which is similar to a database materialized view. Continuous query is a query that is updated in real time.

Data Replication

- Coherence*Extend can be used replicate one cluster to another.

  An extend client can be connected to two clusters and proxy objects between the two!

  - Can be used for disaster recovery sites
  - Requires high-network bandwidth, depending on transaction load
  - Minimize the data to be replicated to what is absolutely necessary
- For more information about replicated clusters, see *Push Replication* in the Coherence Incubator project.



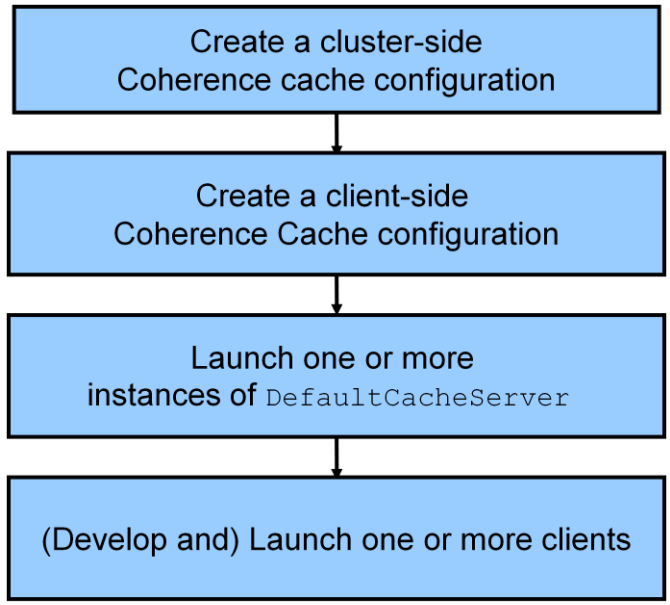
13 - 10

Coherence*Extend can also replicate data from one cluster to another over a WAN connection. Keep in mind that Coherence is so fast, it can actually saturate a gigabit ethernet network. A network between sites, particularly if the sites are in different cities, has very limited bandwidth. If the customer wants to replicate data between clusters, they should minimize the data to be replicated to that which is absolutely necessary. Otherwise, the cost of a network that could supply the required bandwidth would be prohibitively expensive.

The data replication link on the referenced page points to samples of Coherence*Extend data replication.

The Push Replication link on the referenced page points to the Coherence Incubator project, where Coherence engineers are experimenting with different ways to use Coherence that may end up as new features in the Coherence product. Oracle recommends using the Push Replication pattern to replicate data between Coherence clusters. URL Links are subject to change. The current Incubator Project page is https://java.net/projects/cohinc

Configuring Coherence*Extend



13 - 11

To configure Coherence*Extend:

1. Create a cluster-side Coherence cache configuration descriptor that includes one or more `<proxy-scheme>` configuration elements. The `<proxy-scheme>` element is for configuring a `ProxyService` on the cluster side that will accept Coherence*Extend client TCP connections.
2. Create a client-side Coherence cache configuration descriptor that includes one or more `<remote-cache-scheme>` and/or `<remote-invocation-scheme>` configuration elements.
   The `<remote-cache-scheme>` element is for accessing caches over Coherence*Extend. The `<remote-invocation-scheme>` element is for executing code in the cluster via Coherence*Extend.
3. Launch one or more instances of `DefaultCacheServer`, to support proxies instances.
4. Launch one or more clients.

The next few pages describe each of these steps in detail.

Cluster-Side Cache Configuration Descriptor

```
<proxy-scheme>
  <service-name>ProxyService1</service-name>
  <thread-count>2</thread-count>
  <acceptor-config>
    <tcp-acceptor>
      <local-address>
        <address>MyProxyHostname1</address>
        <port>9099</port>
        <reusable>true</reusable>
      </local-address>
    </tcp-acceptor>
  </acceptor-config>
  <autostart>true</autostart>
</proxy-scheme>
<proxy-scheme>
  <service-name>ProxyService2</service-name> . . .
  <acceptor-config>
    <tcp-acceptor>
      <local-address>
        <address>MyProxyHostname2</address>
        <port>9099</port>
. . .
```

Host and port match the settings in the client-side configuration.

proxy-coherence-cache-config.xml

13 - 12

Cluster nodes that need to support Coherence*Extend clients must be configured with a `proxy-scheme`. A `proxy-scheme` configures a Proxy Service to run within a cluster member. `Proxy-scheme`s must be configured with a network address and port to listen for requests. As mentioned in the previous slide, POF is the recommended serialization method for Coherence*Extend applications.

For a Coherence*Extend-TCP client to connect to a Coherence cluster, one or more `DefaultCacheServer` processes must be running that use a Coherence cache configuration descriptor, which includes a `<proxy-scheme>` element with a child `<tcp-acceptor>` element containing various TCP/IP-specific configuration information.

This cache configuration descriptor defines a Coherence*Extend TCP Proxy scheme that allows remote Coherence*Extend TCP clients to connect to the Coherence cluster. The `<proxy-scheme>` element has a `<tcp-acceptor>` child element, which includes all TCP/IP-specific information needed to accept client connection requests over TCP/IP. Because this descriptor is used by a `DefaultCacheServer`, it is important that the `<autostart>` configuration element for each service is set to `true` so that the clustered services are automatically restarted upon termination.

Client-Side Cache Configuration Descriptor

```
<remote-cache-scheme>
  <scheme-name>extend-dist</scheme-name>
  <service-name>ExtendTcpCacheService</service-name>
  <initiator-config>
    <tcp-initiator>
      <remote-addresses>
        <socket-address>
          <address>MyProxyHostname1</address>
          <port>9099</port>
        </socket-address>
        <socket-address>
          <address>MyProxyHostname2</address>
          <port>9099</port>
        </socket-address>
      </remote-addresses>
      <connect-timeout>10s</connect-timeout>
    </tcp-initiator>
  </initiator-config>
</remote-cache-scheme>
```

Host and port of the cluster Proxy Server

client-coherence-cache-config.xml

13 - 14

Coherence*Extend clients are configured via the `remote-cache-scheme` element. A `<remote-cache-scheme>` signifies that it is a Coherence*Extend client. The `<tcp-initiator>` must be configured to point at the address and port of a Proxy Service (`proxy-scheme`) running within the Coherence cluster. It is recommended that POF be used as the serialization method for Coherence*Extend clients. When the client application retrieves a `NamedCache` via the `CacheFactory` using, for example, the name "`dist-extend`", the Coherence*Extend adapter library connects to the Coherence cluster via TCP/IP (using the address "`MyProxyHostname1`" and port 9099) and returns a `NamedCache` implementation that routes requests to the `NamedCache` with the same name running within the remote cluster. Note that the `<remote-addresses>` configuration element can contain multiple `<socket-address>` child elements. The Coherence*Extend adapter library attempts to connect to the addresses in a random order, until either the list is exhausted, or a TCP/IP connection is established.

Launching a Coherence*Extend-Enabled `DefaultCacheServer` Process and Java Client Application

```
java -cp coherence.jar:<classpath to shared jars>
  -Dtangosol.coherence.cacheconfig=file://<path to the
  server-side cache configuration descriptor>
  com.tangosol.net.DefaultCacheServer
```

Cluster side

```
java -cp coherence.jar:<classpath to client application>
  -Dtangosol.coherence.cacheconfig=file://<path to the
  client-side cache configuration descriptor>
  <client application Class name>
```

Client side

13 - 15

To start a `DefaultCacheServer` that uses the cluster-side Coherence cache configuration discussed earlier to allow Coherence*Extend-TCP clients to connect to the Coherence cluster via TCP/IP, you must perform the following:

1. Change the current directory to the Coherence library directory (`%COHERENCE_HOME%\lib` on Windows and `$COHERENCE_HOME/lib` on UNIX).
2. Make sure that the paths are configured so that the Java command will run.
3. Start the `DefaultCacheServer` command-line application with the `-Dtangosol.coherence.cacheconfig` system property set to the location of the cluster-side Coherence cache configuration descriptor that was discussed earlier.

The client steps are almost identical. To start a client application that uses Coherence*Extend-TCP to connect to a remote Coherence cluster via TCP/IP, you must perform the following:

Supported Environments

Coherence*Extend supports the following client environments:

Language	Environment
Java	• x86 and x64 environments running JDK 1.6.0_4 or later
C++	• x86 and x64 bit version of MS Windows Server 2003+ or environments supporting MSVC 2005 SP1+ or MSVC2010 • SPARC, x86 and x64 bit version of Solaris 10 with SunPro 5.9+ • x86 and x64 bit Linux with GCC3.4+ • x64 OSX10,7+ with GCC 3.4+
.NET	• x86 and x64 bit version of MS Windows Server 2003+ or environments supporting MSVC 2005 SP1+ or MSVC2010

13 - 17

Coherence*Extend supports a number of environments including Windows, Linux, and Apple OSX. See the Coherence Client Documentation, Part 1, section 2 for a complete list of all supported platforms and vendor-specific requirements.

When deploying on Microsoft Windows, just as with any MSVC-based application, the corresponding MSVC runtime libraries must be installed on the deployment computer.

Simplifying Coherence*Extend Configuration

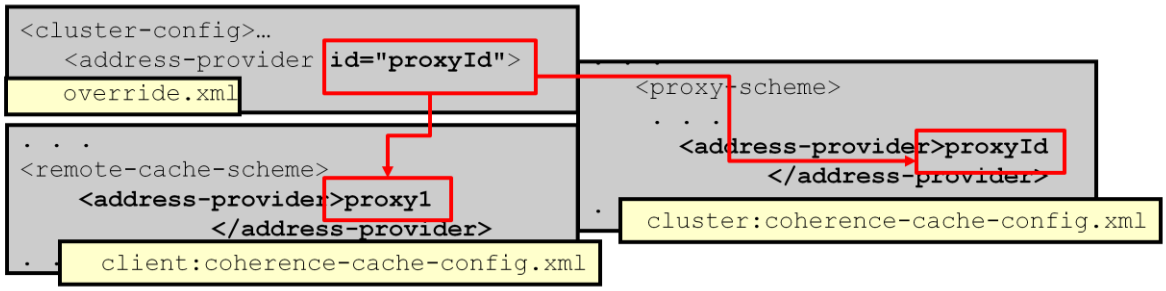
Coherence*Extend supports the use of a naming service and address provider to simplify configuration.

- Address Providers – Define addresses by ID, which can be accessed by cluster and remote client configuration
- Naming Service – Is a specialized TCP acceptor that allows Extend clients to connect to a proxy by specifying a proxy service name instead of a proxy service address

13 - 18

Address Provider References

- Coherence*Extend client and cluster address configuration can be minimized by using *address provider references.*
- Address provider references use `id` attributes in `address-provider` elements to assign addresses symbolic names.
- Address provider reference clients reference address providers by name, eliminating embedded addresses.

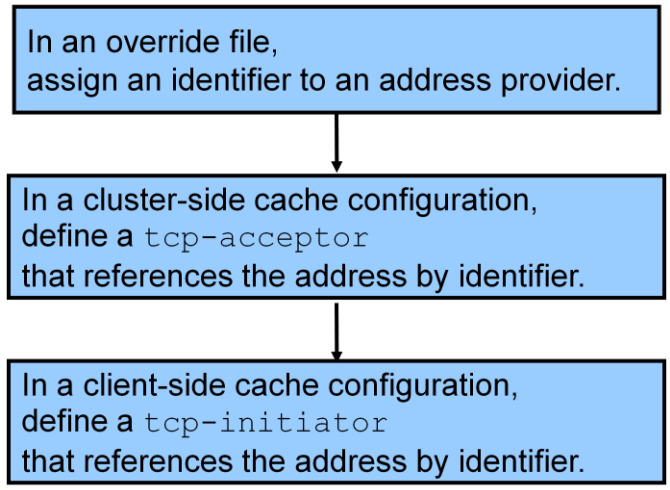


13 - 19

Proxy service, remote cache, and remote invocation definitions can use the `id` attribute of an `<address-provider>` element to reference a TCP socket address that is defined in an operational override configuration file instead of explicitly defining an addresses in a cache configuration file. Referencing socket address definitions allows network addresses to change without having to update a cache configuration file.

Configuring Address Provider References

Configuring address provider references is a three-step process.



13 - 20

Step 1: Override File

In an override file, define an `address-provider` with a `socket-address` and assign identifiers.

```
. . .
<cluster-config>
   <address-providers>
      <address-provider id="proxy1">
         <socket-address>
            <address>198.168.1.5</address>
            <port>9099</port>
         </socket-address>
      </address-provider>
      . . .
   </address-providers>
</cluster-config>
. . .
```

tangosol-coherence-override.xml

13 - 21

Create an override file, such as `tangosol-coherence-override.xml`. Add one or more `<address-provider>` elements, including a `<socket-address>` element with `address` and `port`. Use the `<address-provider>` element's `id` attribute to define a unique ID. The example defines an address with **`proxy1`**.

Step 2: Cluster Configuration

In the cluster-side cache configuration, create or update a `proxy-scheme` definition to reference the `socket-address` by identifier within the `tcp-acceptor`.

```
<caching-schemes>
   <proxy-scheme>
      <service-name>ExtendTcpProxyService</service-name>
      <acceptor-config>
         <tcp-acceptor>
            <address-provider>proxy1</address-provider>
         </tcp-acceptor>
      </acceptor-config>
      <proxy-config>
. . .
      </proxy-config>
. . .
   </proxy-scheme>
</caching-schemes>
...
```

cluster:coherence-cache-config.xml

13 - 22

Edit the cluster-side `coherence-cache-config.xml` and create or update a proxy service definition, and reference a socket address definition by providing the definition's ID as the value of the `<address-provider>` element within the `<tcp-acceptor>` element. The example defines a proxy service that references the address that is defined in the previous slide.

Step 3: Client Configuration

In the client-side cache configuration, create or update a `remote-cache-scheme` definition to reference the `socket-address` by identifier within the `tcp-initiator`.

```
. . .
<remote-cache-scheme>
. . .
   <initiator-config>
      <tcp-initiator>
         <remote-addresses>
            <address-provider>proxy1</address-provider>
            <address-provider>proxy2</address-provider>
         </remote-addresses>
         <connect-timeout>10s</connect-timeout>
      </tcp-initiator>
. . .
   </initiator-config>
</remote-cache-scheme>
. . .
```

client:coherence-cache-config.xml

13 - 23

Edit the client-side `coherence-cache-config.xml` and create or update a remote cache or remote invocation definition and reference a socket address definition by providing the definition's ID as the value of the <`address-provider`> element within the <`tcp-initiator`> element. The example defines a remote cache that references the address that was defined previously.

Configuring a Naming Service

- Coherence*Extend client configuration can be minimized by using naming services.
- Clients connect to the name service acceptor, which provides the actual address of the requested proxy.
- To configure the naming service, perform two steps:

In a cluster-side cache configuration,
define a `tcp-acceptor`
but DO NOT define a `socket-address` element.

↓

In a client-side cache configuration,
define a `tcp-initiator`
that references the address of the name server.

Using name services allows proxy addresses to change without having to update a cache configuration file.

13 - 24

A name service is a specialized TCP acceptor that allows Extend clients to connect to a proxy by specifying a proxy service name instead of a proxy service address. Clients connect to the name service acceptor, which provides the actual address of the requested proxy. The use of the name service acceptor allows actual proxy addresses to change without having to update a cache configuration file.

A name service acceptor automatically starts on the same port as the TCMP socket (8088 by default) if a proxy service is configured on a cluster member. In addition, multiple proxy services can also be configured to use the same listening port as is used by the TCMP socket on the cluster. The use of the same port for TCMP, the name service, and proxy services minimizes the number of ports that are used by Coherence and simplifies firewall configuration.

Step 1: Cluster Configuration

In the cluster-side cache configuration, create or update a `proxy-scheme` definition to define the `service-name`.

```
. . .
<caching-schemes>
   <proxy-scheme>
      <service-name>TcpExtend</service-name>
      <acceptor-config/>
      <autostart>true</autostart>
   </proxy-scheme>
</caching-schemes>
. . .
```

cluster:coherence-cache-config.xml

13 - 25

Edit the cluster-side `coherence-cache-config.xml` and create or update a proxy service definition and do not explicitly define a socket address within the `<tcp-acceptor>` element. The example defines a proxy service named **`TcpExtend`** that binds to the same port that is used by TCMP.

Step 2: Client Configuration

In the client-side cache configuration, create or update a `remote-cache-scheme` and add a `name-service-addresses` element within the `tcp-initiator` that includes the socket addresses of the name service initiator.

```
. . .
remote-cache-scheme>
   <scheme-name>extend-dist</scheme-name>
   <service-name>TcpExtend</service-name>
   <initiator-config>
      <tcp-initiator>
         <name-service-addresses>
            <socket-address>
               <address>198.168.1.5</address>
               <port>8088</port>
            </socket-address>
         </name-service-addresses>
         <connect-timeout>5s</connect-timeout>
      </tcp-initiator>
   </initiator-config>
</remote-cache-scheme>
. . .
```

client:coherence-cache-config.xml

13 - 26

Edit the client-side `coherence-cache-config.xml` and create or update a remote cache or remote invocation definition and add a `<name-service-addresses>` element, within the `<tcp-initiator>` element, which includes the socket address of the name service acceptor on the cluster. The example defines a remote cache definition that connects to a name service to get a connection to the **`TcpExtend`** proxy service that was configured in the previous slide.

Tuning the Coherence*Extend Client

- Set a reasonable request timeout value.
- Configure a heartbeat to detect dropped connections.
- Always configure two or more remote addresses.

```
<remote-cache-scheme>
 . . .
        <initiator-config>
 . . .
      <outgoing-message-handler>
        <request-timeout>5s</request-timeout>
        <heartbeat-interval>60s</heartbeat-interval>
        <heartbeat-timeout>5s</heartbeat-timeout>
       </outgoing-message-handler>
    </initiator-config>
</remote-cache-scheme>
```

The <outgoing-message-handler> element is the last in the initiator-config

client-cache-config.xml

13 - 27

The `request-timeout` element is used after the client has already connected, and is used to detect fatal errors during request processing or a nonresponsive proxy service. Configure a heartbeat to help detect potentially dropped connections. Use the `<outgoing-message-handler/>` configuration elements to set timeout and heartbeat values. Always configure two or more remote addresses for both load balancing and failover of TCP/IP clients.

Always Run Proxy Server with Storage Disabled

- Every member in a Coherence cluster can store data, *including* proxy servers.
- In general, a *storage enabled* proxy server:
  1. Accepts POF formatted data from the client
  2. Deserializes POF data into Java objects In Java, C++, or .NET
  3. Serializes Java objects and places them into cache

These operations are costly in terms of memory and CPU and can be avoided if local storage is disabled!.

```
-Dtangosol.coherence.distributed.localstorage=false
```

13 - 28

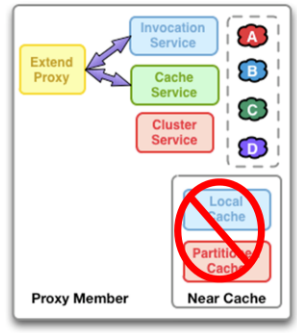
Each server in a partitioned cache, including the proxy server, can store a portion of the data. The proxy server has the responsibility of accepting POF formatted data from the client (Java, C++, or .NET), deserializing POF data to get the Java objects, serializing the Java objects, and then placing the resulting data in the cluster. These tasks can be expensive in terms of CPU and memory. You can preserve resources on the proxy server by disabling its local storage.

There are several ways in which you can disable storage:

- Local storage for a proxy server can be enabled or disabled with the `-Dtangosol.coherence.distributed.localstorage` Java property.
- You can also disable storage in the cache configuration file. See the description of the `<local-storage>` element in "`distributed-scheme`."

Never Run Near Cache on Proxy Servers

- Near cache provides local cache access to recently and/or often-used data.
- Proxy servers configured with a near cache will locally cache data accessed for remote clients.

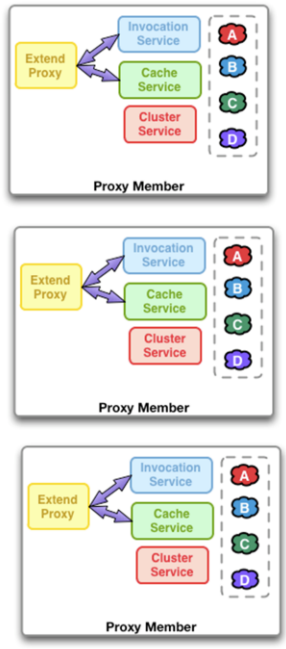


13 - 29

By definition, a near cache provides local cache access to recently and/or often-used data. If a proxy server is configured with a near cache, it will locally cache data accessed by its remote clients. It is unlikely that these clients will be consistently accessing the same subset of data, thus resulting in a low hit ratio on the near cache. This will result in higher heap usage and more network traffic on the proxy nodes with little to no benefit. For these reasons, it is recommended that a near cache not be used on a proxy server. To ensure that the proxy server is not running a near cache, remove all near schemes from the cache configuration being used for the proxy.

Running Multiple Proxies

- Proxies perform POF conversions in a service thread.
- A single proxy can bottleneck during conversations.
- Running multiple proxies helps spread processing.
- Clients:
  - Obtain a connection once per session
  - Randomize addresses on first request
  - Connect by using the first available connection

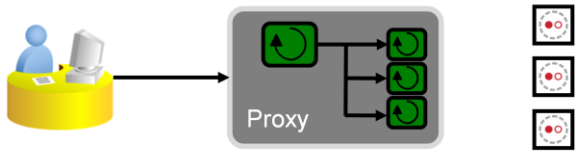


13 - 30

The proxy performs POF/EL conversions in the service thread. A single proxy instance can easily bottleneck on a single core due to POF/EL conversions. Running multiple proxy instances on the same box (instead of increasing the thread pool size) helps spread the load across more cores.

Dynamic Proxy Thread Pooling

- Proxy services use a dynamic thread pool that automatically manages threads based on client requests, backlog, and the idle thread count.
- **Feature:** Dynamic proxy thread pooling provides tight control over total proxy thread behavior.
- **Benefit:** Control thread minimum and maximum based on client application behavior rather than default definitions.



13 - 31

Proxy services use a dynamic thread pool for worker threads. Threads are managed based on the number of client requests, total backlog of requests, and the total available thread count. The thread pool helps ensure that there are enough threads to meet the demand of Extend clients and that resources are not wasted on idle threads.
Applications have different thread requirements based on the client counts and application workload. Client applications are generally classified into two categories: Active and passive. Active applications send many requests (`put`, `get`, and so on) that are handled by the proxy service. In such situations, the proxy service requires a large number of threads to handle these requests. Passive applications, on the other hand, are typically waiting on background events, such as event processors and interceptors. In such scenarios, events are handled by a distributed cache service and the proxy service uses worker threads to push events to the client. For these tasks, the thread pool configuration for the distributed cache service should include enough worker threads, but the proxy service can have fewer.

Coherence*Extend supports managing thread counts via settings in a proxy service definition.

Configuring Thread Counts

Proxy thread pool by default:

- Starts at 2 times core count
- Expands to 8 times core count
- Is more aggressive when creating threads than decreasing threads
- Can be configured by using `thread-count[min|max]`

When backlog increases, additional threads are created every 1/2 second.
Idle threads are removed every 60 seconds.

```
. . .
<proxy-scheme>
   <service-name>ExtendTcpProxyService</service-name>
   <thread-count-max>75</thread-count-max>
   <thread-count-min>10</thread-count-min>
   <acceptor-config>. . .</acceptor-config>
   . . .
</proxy-scheme>
. . .
```

coherence-cache-config.xml

13 - 32

Default thread pool behavior starts with a thread count that is twice the number of processor cores and grows to a maximum of 8 times the number of cores. The thread pool is more aggressive in creating threads than decreasing threads. If client requests and service backlog increases, additional threads are created every 0.5 second. If too many threads become idle, threads are removed from the thread pool every 60 seconds.

To set thread pooling thresholds for a proxy service, add the `<thread-count-max>` and `<thread-count-min>` elements within the `<proxy-scheme>` element. See the online documentation *Developing Applications with Oracle Coherence* for detailed information about these elements.

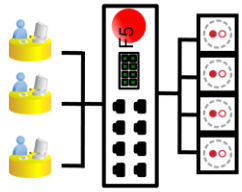
Note

- When enabling dynamic thread pool sizing, do not set the `<thread-count>` element. Setting a thread count value disables dynamic thread pool sizing.
- Setting a maximum thread count of zero forces the proxy service thread to handle all requests; no worker threads are used.

F5 BIG-IP Local Traffic Manager

BIG-IP Local Traffic Manager from F5:

- Is a hardware device that provides intelligent analysis of routing, server failure, and other issues
- Provides support for external interaction and integration
- Can be configured to provide support for Coherence*Extend clients. Coherence*Extend Clients are then configured to interact with a BIG-IP virtual server rather than a specific proxy.



F5
LTM

13 - 33

F5 BIG-IP LTM is a hardware device that sits between one or more computers running Coherence*Extend clients (client tier) and one or more computers running Coherence*Extend proxy servers (proxy tier). The LTM spreads client connections across multiple clustered proxy servers by using a broad range of techniques to secure, optimize, and load balance application traffic.

Big IP uses the concept of a node to control network traffic. A node is a logical object on the BIG-IP system that identifies the IP address of a physical resource on the network. For Coherence*Extend, configure a node for each computer on the internal network that hosts one or more proxy servers.

A load balancing pool is a group of logical devices, such as proxy servers, that receive and process traffic. Instead of sending client traffic to the destination IP address specified in the client request, the BIG-IP system sends the request to any of the servers that are members of that pool. This helps efficiently distribute the load on your server resources.

Dynamic Load Balancing for Coherence*Extend

- Feature:
  - Default (proxy) implementation balances connections based on a variety of metrics at both the client and proxy levels.
  - It provides support for custom pluggable *Extend Connection load balancing.

Using connection count, daemon pool size, and message backlog

- Benefits:
  - New connections directed to the least-used proxy
  - APIs to develop and provide custom connection load balancers

Using application specific metrics

  - Minimal configuration

13 - 34

Load balancing is provided in Coherence 12*c* in two ways. First, the default proxy-based mechanism has been improved to enable better load balancing. In general, the proxy-based load balancing algorithm distributes client connections equally across proxy service members. The algorithm redirects clients to proxy services that are being utilized the least. The following factors are used to determine a proxy's utilization:

- **Connection Utilization** – This utilization is calculated by adding the current and pending connection counts. If a proxy has a configured connection limit and the current plus pending connection counts equal the connection limit, the utilization is considered to be infinite.
- **Daemon Pool Utilization** – This utilization equals the current number of active daemon threads. If all daemon threads are currently active, the utilization is considered to be infinite.
- **Message Backlog Utilization** – This utilization is calculated by adding the current incoming and current outgoing message backlogs.

Additionally, Coherence 3.7 adds the `com.tangosol.coherence.net.proxy` package, which includes APIs for the development of custom load balancing plugins based on specific client algorithms.

Proxy-Based Load Balancing

Proxy-based load balancing is:

- The default if no method is specified
- Based on the load-balancer element that is set to `proxy`

```
. . .
<proxy-scheme>
   <service-name>ExtendTcpProxyService</service-name>
   <acceptor-config>
      <tcp-acceptor>
         <local-address>
            <address>192.168.1.5</address>
            <port>9099</port>
         </local-address>
      </tcp-acceptor>
   </acceptor-config>
   <load-balancer>proxy</load-balancer>
   <autostart>true</autostart>
</proxy-scheme>
. . .
```

coherence-cache-config.xml

13 - 35

Proxy-based load balancing is the default strategy that is used to balance client connections between two or more proxy services. The strategy is weighted by a proxy's existing connection count, its daemon pool utilization, and its message backlog.

The proxy-based load balancing strategy is configured within a `<proxy-scheme>` definition using a `<load-balancer>` element that is set to `proxy`.

Connection Load Balancing

Coherence*Extend supports:

- Client-based load balancing
  - Uses a client address provider implementation
  - Accesses each proxy randomly until a connection is made
- Proxy-based load balancing is:
  - The default strategy
  - Used to balance client connections
  - Weighted by using various proxy factors including:
    - Existing connection count
    - Daemon pool utilization
    - Message backlog
- Custom load balancing

Between two or more proxy services

13 - 36

Coherence*Extend client connections are load-balanced across proxy service members. By default, a proxy-based strategy is used that distributes client connections to proxy service members that are least loaded. Custom proxy-based strategies can be created or the default strategy can be modified as required. Alternatively, client-based load balance strategies can be implemented by creating a client-side address provider or by relying on randomized client connections to proxy service members. The random approach provides minimal balancing as compared to proxy-based load balancing.

Client-Based Load Balancing: Per Proxy

Client-based load balancing can be specified:

- Per proxy
- As the system default

```
. . .
<proxy-scheme>
    . . .
    <load-balancer>client</load-balancer>
    . . .
</proxy-scheme>
. . .
```

coherence-cache-config.xml

13 - 37

The client-based load balancing strategy relies on a client address provider implementation to dictate the distribution of clients across proxy service members. If no client address provider implementation is provided, the Extend client tries each configured proxy service in a random order until a connection is successful. The client-based load balancing strategy is configured within a `<proxy-scheme>` definition by using a `<load-balancer>` element that is set to `client`.

Client-Based Load Balancing: System-Wide

Client-based load balancing can be specified as the system default.

```
<?xml version='1.0'?>
<coherence xmlns:xsi=". . . ">
   <cluster-config>
      <services>
         <service id="7">
         <init-params>
             <init-param id="12">
                 <param-name>load-balancer</param-name>
                 <param-value>client</param-value>
            </init-param>
         </init-params>
      </service>
    </cluster-config>
</coherence>
```

13 - 38

To set the client strategy as the default strategy for all proxy services if no strategy is specified, override the `load-balancer` parameter for the proxy service type in the operational override file. Note that `7` is the ID of the proxy service and `12` is the ID of the `load-balancer` parameter.

Proxy-Based Load Balancing Algorithm

The default proxy load balancing algorithm attempts to distribute client connections evenly based on:

- Connection Utilization – Is calculated by adding the current and pending connection counts
- Daemon Pool Utilization – Equals the current number of active daemon threads
- Message Backlog Utilization – Is calculated by adding the current incoming and current outgoing message backlogs

Each proxy service maintains a list of all proxy services ordered by their utilization. The ordering is weighted first by connection utilization, then by daemon pool utilization, and then by message backlog.

13 - 39

Each proxy service maintains a list of all proxy services ordered by their utilization. The ordering is weighted first by connection utilization, then by daemon pool utilization, and then by message backlog. The list is re-sorted whenever a proxy service's utilization changes. The proxy servers send each other their current utilization whenever their connection count changes, or every 10 seconds (whichever comes first).

When a new connection attempt is made on a proxy, the proxy iterates the list as follows:

- If the current proxy has the lowest connection utilization, the connection is accepted; otherwise, the proxy redirects the new connection by replying to the connection attempt with an ordered list of proxy servers that have a lower connection utilization. The client then attempts to connect to a proxy service in the order of the returned list.
- If the connection utilizations of the proxies are equal, the daemon pool utilization of the proxies takes precedence. If the current proxy has the lowest daemon pool utilization, the connection is accepted; otherwise, the proxy redirects the new connection by replying to the connection attempt with an ordered list of proxy servers that have a lower daemon pool utilization. The client then attempts to connect to a proxy service in the order of the returned list.

Custom Load Balancing

Custom load balancing:

- Is specified via the instance element
- Implements `com.tangosol.coherence.net.proxy.ProxyServiceLoadBalancer`

```
. . .
<proxy-scheme>
    . . .
    <load-balancer>
        <instance>
            <class-name>
            com. . . MyCustomLoadBalancer
            </class-name>
        </instance>
    </load-balancer>
    . . .
</proxy-scheme>
. . .
```

coherence-cache-config.xml

13 - 41

The client-based load balancing strategy relies on a client address provider implementation to dictate the distribution of clients across proxy service members. If no client address provider implementation is provided, the Extend client tries each configured proxy service in a random order until a connection is successful. The client-based load balancing strategy is configured within a `<proxy-scheme>` definition by using a `<load-balancer>` element that is set to `client`.

Summary

In this lesson, you should have learned how to:

- Describe what Coherence*Extend does, its architecture, its advantages and disadvantages, and when to use it
- List and explain the three types of Coherence*Extend clients and their uses
- Configure, run, and tune Coherence*Extend
- Explain the basic development of Coherence*Extend client applications

13 - 42

This lesson covers the basic concepts of Coherence*Extend and Coherence*Web features. When this lesson is completed, you should be able to configure and use both products.