



## xTier™ Cache Service

The xTier™ Cache Service provides a high-performance, non-replicable distributed object caching service.

### Key Benefits:

- High-performance non-replicable design
- Native xTier™ Cache interface and JCache JSR 107 compatible interface for cache operations.
- Full 2PC ACID cache transactions
- Full support for read through and write through operational modes.

## Introduction

A cache is a special high-speed, software-based storage mechanism that provides instant in-memory access to data that otherwise would have to be fetched over the network, usually from an underlying database. Caching is effective in applications with read-mostly behavior where data is accessed more frequently than it is changed. In many cases, caching can minimize latencies associated with data access over the network, reduce CPU utilization imposed by data serialization and de-serialization and significantly decrease the load on backend storage systems, such as a database.

Before examining the xTier™ Cache Service features in detail, the most commonly used caching architectures and approaches will be introduced. The three most common approaches for caching are Replicable, Non-replicable, and Distributed Map caches. There are also hybrid architectures that use components of several of these methods in combination. Additionally, a variety of messaging network protocols may be adopted to keep the cache data up-to-date. For instance, some solutions use TCP/IP and others use UDP or IP-multicast. Messaging protocols built on top of TCP or UDP, such as JMS, may also be used.

### Replicable Cache

The main property of a replicable cache is self-evident from its name – the state of every cached object is replicated across every node in the cluster. Whenever data modification occurs on one of the nodes, the whole object state (or sometimes only the state delta) is automatically copied to all other nodes. Such an approach guarantees the highest response times for read operations because at any given point of time all objects are theoretically cached in memory.

However, since an object may actually be accessed only on a single node, replicating its state to all other nodes unnecessarily enlarges the heap footprint for every node that in practical use may never be used to access this object. If the amount of memory needed for cached objects exceeds the memory available, objects have to be evicted from the cache utilizing some sort of eviction policy. Full state replication, when it becomes moderately frequent, can also burden CPU's with expensive serialization/de-serialization operations on all cluster nodes as well as possibly saturating the network.

It is also important to note that when dealing with non-persistent data, replicable cache often becomes a natural solution for storing data at the application level. For example, certain variations of replicable cache can be applied to support transparent Servlet/EJB session fail-over within J2EE application servers. In this case, in order to mitigate the issues described above, every cached session object often has one designated back-up cluster node to which the state is replicated.

### **Non-replicable Cache**

Non-replicable caches take the opposite approach toward state replication to achieve cache coherency. Upon every data modification, a small invalidation message containing the object key is sent to all nodes in order to effectively evict dirty references to the modified object from the cluster. The first read request following invalidation will reload the evicted object from the underlying storage mechanism. It should be evident that non-replicable caches can only be used for storing data backed by some secondary storage mechanism since the implied design requires reloading the state upon invalidation.

Non-replicable caches in certain cases may not offer response times as high as with replicable approach, however in any read-mostly system the difference should be insignificant. On the other hand, by minimizing the size of invalidation messages the non-replicable approach offers several benefits over the replicable one since short invalidation messages allow for negligible serialization/de-serialization and impose minimal load on the network. Loading objects on demand allows for every node to cache data that has only been locally accessed minimizing heap utilization. In addition, since invalidation messages are small and fixed in size, they can be effectively transmitted over lightweight UDP or IP-multicast protocols.

### **Distributed Map**

Distributed map is an interesting caching design paradigm which treats the whole cache cluster as one big hash map. In a pure distributed map approach there is only one copy of any piece of data within the whole cluster (semantically analogous to `java.util.HashMap`). Data access within the distributed map will usually entail a network trip. In the worst case scenario, N data requests initiated on the same node will require N-1 hops over the network to the nodes designated for storing the requested piece of data. In most cases a network trip to another cluster node incurs less overhead than a trip to the database, accessing cached data in a distributed map is generally faster than issuing the average size database queries directly.

Since the majority of read operations travel over the network, distributed maps suffer from significant object serialization/de-serialization overhead and large response time latencies. This renders a pure distributed map approach almost unusable in a significant majority of applications. To mitigate this issue distributed map caches are often used in

combination with local caches. Whenever data is updated in the local cache on some node, invalidation or state-replication messages (or some combination of both) are sent to all nodes in the cluster in order to evict or replace the updated references in all local caches and also on the designated node of the distributed map. Such behavior, however, is notably similar to replicable/non-replicable caches but does not enjoy the significantly less overhead and much lighter architectures that these latter methods have.

### JCache - JSR 107 (Simple `java.util.Map` interface)

JSR 107 is one of the oldest open JSR's in the JCP. Thus far it has not been open for public review. Most participating vendors, however, talk openly about the shape and form JSR 107 is going to take. The current assessment is that it will be heavily based on the `java.util.Map` API with various flavors and additions.

The xTier™ design team carefully evaluated the caching products and APIs from all vendors on the JSR 107 panel that currently claim compliance with JSR 107 and concluded that other than using the `java.util.Map` API in one form or another, there are very little similarities between compliant vendors. Furthermore, there isn't even an agreement on such basic API's such as `CacheLoader` and `CacheStore`, which would have to be used within any project. Needless to say, there is almost nothing in common among the proprietary API additions on top of basic `java.util.Map` interface. Thus claiming JCache compliance or even calling JCache a standard is essentially a meaningless statement with respect to compatibility between vendors.

Furthermore, although the `java.util.Map` interface is convenient to work with, it was never designed to be used in scenarios where some methods can fail or produce errors. None of the methods from the `Map` interface throw any checked exceptions. However, any `Cache.get(...)` or `Cache.put(...)` operation may involve loading data from the database or propagating modifications to the database. Such operations will involve network trips, execution of SQL on backend data storages and may potentially fail. Some vendors try to resolve this situation with error handlers, some with runtime exceptions, but all these solutions seem rather like a hack introduced to artificially squeeze complex caching logic into the trivial `java.util.Map` interface.

The xTier™ cache is also heavily based on a Map interface, however the main Cache interface does not extend `java.util.Map`. The reason for this is that some methods on the Cache interface *must* account for failure and propagate it to the calling code via exceptions. The xTier™ design team believes that this approach is much cleaner than blind usage of the `java.util.Map` interface. However, for those who don't share this point of view, the xTier™ Cache Service also provides a JCache API which extends the `java.util.Map` and wraps all checked exceptions from the xTier™ Cache API into runtime exceptions.

### The xTier™ Cache Service

xTier™ provides a distributed non-replicable cache service. The following paragraphs provide a detailed explanation of the xTier™ Cache Service features.

#### Read-through and Write-through

The xTier™ Cache Service supports read-through and write-through operational modes, however, neither of these modes are mandatory.

Read-through behavior implies automatically loading data from backend storage into the cache when a cache miss occurs. Whenever an attempt is made to access some data from the cache within a given transaction, the cache implementation will first determine if this data is available in the per-transaction cache; if not, then the global cache will be checked. If the data is not available in the global cache, then the cache loader (the implementation of which is supplied by the developer) is used to load the data from backend storage, i.e. a database. The loaded data will then be stored in the cache and then returned to the calling code. In the case when a cache loader is not provided by the developer (not read-through mode), the user would have to manually load the data from the database and store it in the cache for every cache miss.

Write-through behavior implies automatically storing the data in the backend storage whenever it is updated in the cache. This behavior is analogous to read-through mode. Whenever a given piece of data is modified in the cache within a transaction, it is stored in the per-transaction cache and also propagated to the database using the cache store implementation provided by the user. If a cache store implementation is not provided (not write-through mode), the user would have to manually update the database every time data is modified in the cache. Upon transaction commit, data is flushed from the per-transaction cache into the global cache.

Read-through and write-through modes allow the user to interact exclusively with the cache API without polluting the code with sporadic SQL statements. It also allows concentrating all SQL logic in one central location.

### Cache Two-Phase-Commit (2PC) Transactions

The xTier™ Cache Service provides support for JTA-like 2PC transactions with **SERIALIZABLE** and **READ\_COMMITTED** isolation levels. The **READ\_COMMITTED** isolation level guarantees that there will be no dirty data read at any point. In other words, the user will never get data from the cache that has been changed but not committed. The **SERIALIZABLE** isolation level is the strictest transaction isolation level. It allows for transactions to seem as they execute serially to each other and strictly enforces the ACID property. Each transaction truly appears to be independent of the others. As with the **READ\_COMMITTED** isolation level, the user has a guarantee that no data will be read if it has not been committed. The user is also able to reread the same data again and again knowing that it has not been changed outside of the scope of the ongoing transaction. Additionally, the user can access multiple objects within the transaction knowing that they will not change during the scope of the ongoing transaction. The xTier™ Cache Service supports the **SERIALIZABLE** isolation level optimistically, i.e. an exception (*CacheOptimisticLockException* in particular) will be thrown if the serializable property of the transaction can not be guaranteed.

It is important to note that all concurrency issues for either isolation level are effectively handled directly within the local VM without holding any internal locks. This means that the user can, in most cases, run the database in a lightweight **READ\_COMMITTED** mode (which incurs very little overhead), while enjoying all the benefits of **SERIALIZABLE** behavior at the application level. Taking advantage of this feature significantly improves database performance since the **SERIALIZABLE** transaction mode significantly hinders the performance of most databases.

It is a common misconception that it is enough to switch isolation level in the database, e.g. set it to **SERIALIZABLE**, in order to get the desired behavior at the application level. In systems implemented without caching such a statement would hold true. However, any cache requires special internal support for transaction isolation levels. The reason is that when cached data is accessed within a transaction, the underlying data store has no knowledge of it, and hence is not able to enroll this data into a transaction. Thus, comprehensive transaction management is important for any cache product.

## Two-Phase Commit Protocol (2PC)

Cache transactions must be coordinated across all participating cluster nodes. The nodes participating in a transaction are elected using a pluggable cache topology implementation provided by the developer. The xTier™ Cache Service utilizes a 2PC protocol in order to enforce transactional data integrity across all nodes and backend storage.

The first phase of the 2PC protocol is initiated by calling the **CacheTx.prepare()** method. During this phase a synchronous UDP message is sent to every participating node. This message, among other things, contains information about all keys enrolled in the transaction. Upon the receipt of the phase-one message the remote nodes will locally start a transaction, enroll all received keys into it, and send a reply back to the initiating node. For **SERIALIZABLE** transactions the remote nodes will also identify if any keys within this transaction conflict with other local ongoing transactions, and if so will notify the initiating node of an optimistic lock failure. After all nodes have successfully acknowledged the transaction, the first phase completes successfully. Immediately after completion of the first phase, the developer must commit the transaction to the backed storage such as a database.

If the database transaction was successfully committed, then the second phase of the cache transaction 2PC protocol is initiated by calling the **CacheTx.commit()** method. During the second phase, a small UDP message containing only the transaction ID is sent to all the participating nodes. Upon receiving this message, the nodes will fetch the active transaction and commit it. Note, that the second phase message is asynchronous and no reply is sent back. The initiating node returns immediately from the call to **CacheTx.commit()** after sending the message.

This approach relies on the fact, that although UDP/IP is an unreliable protocol, it rarely fails. In fact on any average LAN network, the failures almost never occur. However, failures do happen and nodes that do not get a second phase message must still complete the transaction. To handle such situations, the xTier™ Cache Service comes with a Distributed Garbage Collector (DGC) the sole responsibility of which is to detect long running transactions, make sure that they have already completed on the initiating nodes and, if so, complete them locally.

## Pluggable Expiration Policies

Evicting objects from the cache has a direct impact on the performance and the proper eviction policy must be carefully tailored for every application. The xTier™ Cache Service provides the four most commonly used expiration policies out-of-the-box. Additionally, the developer may choose to implement their own expiration policy if none of the pre-built ones fit the requirements. Expiration policies, as all other xTier™ pluggable elements, are specified in the XML cache configuration file as an IoC component.

The pre-built expiration policies shipped with cache service are:

- *Least Recently Used (LRU) Policy* – objects are evicted based on their access order (or, optionally, insertion order) whenever the number of objects in the cache exceeds the specified maximum.
- *Least Frequently Used (LFU) Policy* – objects are evicted according to their usage pattern. The ones that are accessed least frequently get evicted first once the number of objects exceeds the specified maximum.
- *Age (Time-To-Live) Policy* – objects get evicted from the cache once a predefined period of time passes since their insertion into the cache.
- *Idle Policy* – objects get evicted from the cache whenever they remain idle (or untouched) for a certain period of time.

## Pluggable Cluster Node Topology Resolution

All application level caches are distributed by nature. Hence, messages need to be exchanged between cluster nodes in order to guarantee cache coherency. Excessive messaging may slow down the network, overload the CPU, and hinder the overall performance of the application. That is why it is very important to carefully define the participating cluster nodes (also known as the cache topology) for every entry stored in the cache. The xTier™ Cache Service achieves this with the use of a cache topology resolver implementation which is supplied by the developer. The xTier™ Cache Service provides a basic topology resolver that comes out-of-the-box, which assumes that every cache entry may potentially be stored on every cluster node. Topology resolving takes place during phase one of the 2PC transaction protocol since at this point the implementation needs to know which cluster nodes should participate in the transaction.

Custom topology resolving effectively removes a single point bottleneck from the cluster. If every message is always distributed to all nodes in the cluster, then the application performs as fast as the slowest node in the cluster. However, most data is usually only stored on one or two nodes. Properly configured topology resolution in many cases will entirely remove any need for cache transaction messaging or in the least it will distribute most messages only to one or two neighboring nodes.

For example, imagine an application where traders login at their workstations to perform equity trading. Every trader gets assigned specific orders to operate on. There is also a lead trader who can operate on orders assigned to him or to any other trader. If caching is utilized to store orders on every workstation, the topology resolution would always involve only two nodes: the workstation of the trader operating on the order, and workstation of the lead trader. In a financial organization with about 30 traders, the performance benefits of proper topology resolution over blind cluster-wide all node distribution are more than evident.

## Invalidation Scope and Query Result Caching

Often an application needs to cache collections of objects, for example, data store query results. Holding query results in cache is not trivial, since every query result collection may become invalid once any application object is updated. Evicting all query results from the cache is usually not an option since the overhead implied would probably erase any benefits of caching. The xTier™ Cache Service addresses this problem with the notion of *groups* and *depended objects*. Group ID and depended flags are assigned to every entry stored in the cache via the cache key attributes resolution.

Groups are introduced to define a scope for invalidations. In short, any modification of an object within a certain group cannot affect any object belonging to other groups and vice versa. All objects in a group can be either *depended* or *not-depended*. In the case of any object modification within a group, all *depended* objects within that group will be invalidated. However, *not-depended* objects within a group are not affected if other objects within the same group are changed. Often *not-depended* objects represent objects keyed by their database primary keys, and *depended* ones usually represent collections of *not-depended* objects, e.g. data store query results.

This approach allows for storing query results in the cache without having to recalculate them for every access or, even worse, querying the database. Note, however, that the API design is flexible enough to configure any object in the system as *depended* or *not-depended*. The developer can assemble any combination of objects into a *depended* object; it does not necessarily have to be a query result, although it often is.

## Sample Usage

The way in which the xTier™ Cache Service is used is similar to the usage pattern for J2EE transactions. The first step is to start a transaction, next the data is retrieved from the cache or stored in the cache and finally the cache transaction is prepared and committed.

The following code snippet illustrates a simple example of cache usage.

```
private static void example() {
    // 1. Get a handle on default cache instance.
    Cache cache = XtierKernel.getInstance().cache().getCache();

    CacheTx tx = null;

    try {
        // 2. Start serializable cache transaction.
        tx = cache.startTx(CacheTx.SERIALIZABLE);

        // 3. Retrieve objects from cache.
        Object obj1 = cache.get(new Integer(1));
        Object obj2 = cache.get(new Integer(2));
        Object obj3 = cache.get(new Integer(3));

        // 4. Update single object.
        Object newObj = new Object();

        cache.put(new Integer(1), newObj);

        // 5. Prepare cache transaction.
        tx.prepare();

        // 6. Commit cache transaction.
        tx.commit();
    }
    catch (CacheTxOptimisticLockException e) {
        // Some implementations may choose to retry the transaction whenever
        // optimistic lock failure happens. For simplicity sake here we simply log it.
        logger.warning("Cache optimistic lock failure occurred.", e);

        // 7. Rollback cache transaction.
        rollbackTx(tx);
    }
}
```

```
catch (CacheException e) {
    logger.error("Unexpected cache error.", e);

    // 8. If transaction was started, roll it back.
    rollbackTx(tx);
}

private static void rollbackTx(CacheTx tx) {
    if (tx != null) {
        try {
            tx.rollback();
        }
        catch (CacheException e) {
            logger.error("Error when rolling back cache transaction: " + tx, e);
        }
    }
}
```

Note that the preceding code snippet assumes that database connections and transactions are handled via **CacheTxListener** notifications. **CacheTxListener** is invoked through the transaction event notification mechanism that allows the user to be notified whenever the transaction is started, prepared, committed, or rolled back. Often it is advantageous to handle all SQL connection logic in the listener class. This allows the developer to deal exclusively with the **Cache** and **CacheTx** interfaces while all database-related logic is handled transparently behind the scenes by the **CacheStore** and **CacheTxListener** implementations.

The following section examines the code example one step at a time.

```
// 1. Get a handle on default cache instance.
Cache cache = XtierKernel.getInstance().cache().getCache();
```

The first step is to obtain a handle on the default cache. In a similar way we can obtain a handle on any named cache in the system. Note that we could also have chosen to work with the JCache wrappers instead of the xTier™ Cache Service specific API.

```
// 2. Start serializable cache transaction.
tx = cache.startTx(CacheTx.SERIALIZABLE);
```

The next step is to start a cache transaction. A cache transaction is required in order to perform any data modification. When retrieving data from the cache, the transaction can be omitted, but in such cases an implicit **READ\_COMMITTED** cache transaction will be created. In this example we choose to use a **SERIALIZABLE** cache transaction to guarantee that all reads and modifications are consistent with each other and happen absolutely independently of any other concurrent transactions. Note that starting a transaction initiates a cache transaction event notification.

```
// 3. Retrieve objects from cache.
Object obj1 = cache.get(new Integer(1));
```

```
Object obj2 = cache.get(new Integer(2));  
Object obj3 = cache.get(new Integer(3));
```

Now several single objects are retrieved from the cache. Every key will be automatically assigned a Group ID and depended flag in order to allow the cache implementation to control the invalidation scope of all cache entries. Note that the same would happen if an object was to be stored in the cache.

```
// 4. Update single object.  
Object newObj = new Object();  
cache.put(new Integer(1), newObj);
```

In this step an attempt is made to store a new object in the cache. As with calls to get methods, the cache key will be assigned key attributes. Additionally, a new order will be stored in a per-transaction cache so it can be retrieved from the cache later within the same transaction.

```
// 5. Prepare cache transaction.  
tx.prepare();
```

During this step the cache coordinates the transaction with all other participating nodes in the cluster. For every key participating in the transaction, the implementation will call the **CacheTopology.getNodes(...)** method. Invalidation messages will be sent out only to the nodes supplied by the **CacheTopology** implementation. After all participating nodes successfully acknowledge this transaction the transaction event notification will be triggered to notify the listeners that the transaction completed successfully. Note that in the code snippet above, the database transaction is committed within the transaction listener callback.

```
// 6. Commit cache transaction.  
tx.commit();
```

Next the cache sends a commit message to all nodes participating in the transaction. Note that unlike the **CacheTx.prepare()** method, a call to the **CacheTx.commit()** or **CacheTx.rollback()** methods does not have to wait for replies from all participating nodes and returns immediately. If the message does not reach any of the designated nodes, then the Distributed Garbage Collector (DGC) will detect that and will properly complete the cache transaction. Note that committing or rolling back a transaction will trigger a cache transaction event notification.

Also note that in this cache service example the database connection will be closed in this step within the cache transaction listener callback. Also, if the transaction event notification was caused by a rollback event, then the database transaction will be rolled back in the transaction event listener as well.

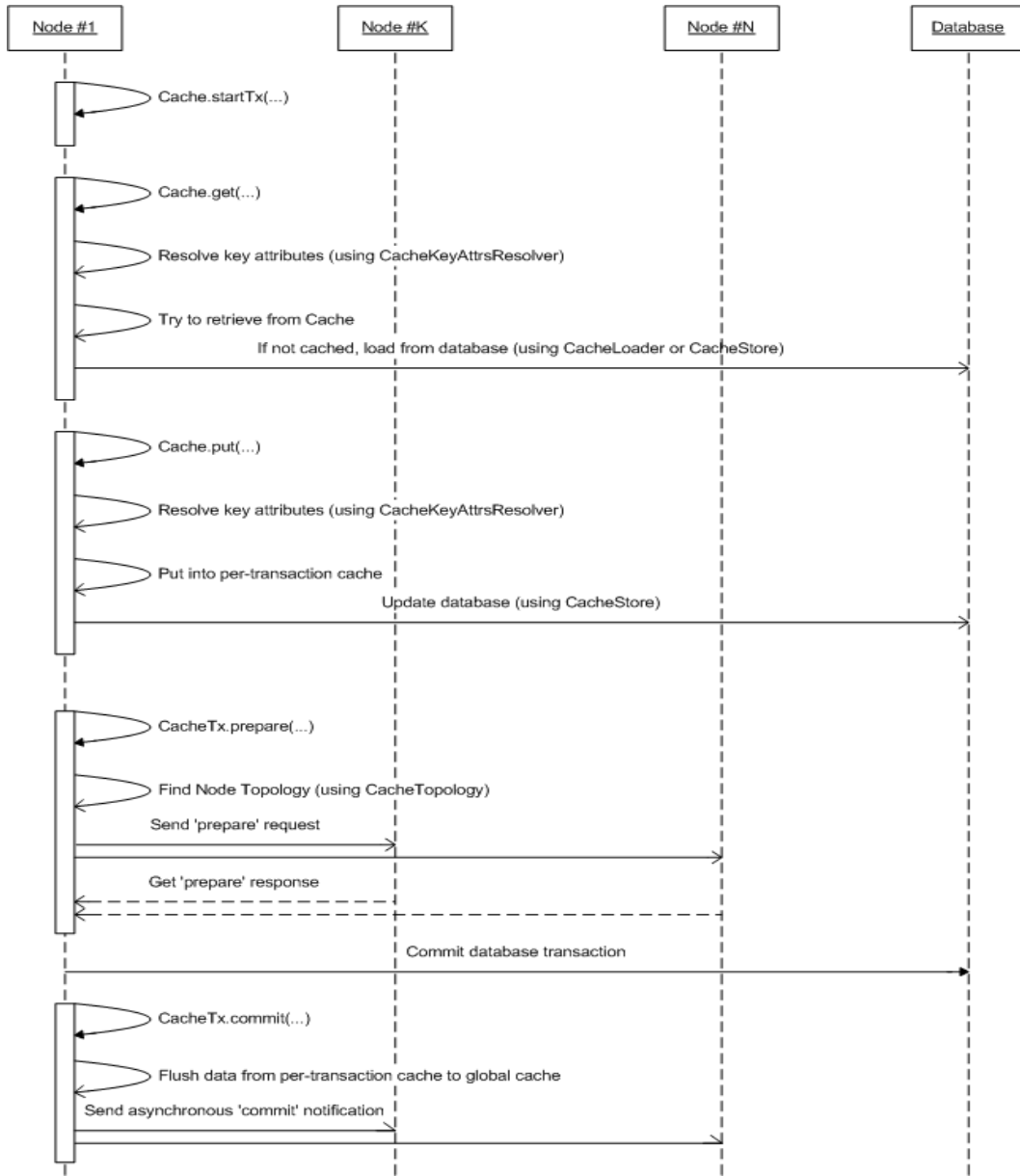
```
catch (CacheTxOptimisticLockException e) {  
    // Some implementations may choose to retry the transaction whenever  
    // optimistic lock failure happens. For simplicity sake here we simply log it.  
    logger.warning("Cache optimistic lock failure occurred.", e);  
    // 9. Rollback cache transaction.  
    rollbackTx(tx);  
}
```

If the application logic makes use of **SERIALIZABLE** transactions, then any transaction in the system may produce an optimistic lock failure exception. This simply means that there were one or more concurrent transactions and the cache implementation could not resolve all concurrency issues to allow for all concurrent transactions to execute. In this step the user must rollback the cache transaction.

```
catch (CacheException e) {  
    logger.error("Unexpected cache error.", e);  
  
    // 10. If transaction was started, roll it back.  
    rollbackTx(tx);  
}
```

In this step any exceptions other than an optimistic lock failure are caught and the transaction must be rolled back. Please see the **CacheTx** documentation for more information on what may fail during the course of a cache transaction.

The following page contains the sequence diagram representing the code snippet.



**Fitech Laboratories Inc.**

**Corporate Headquarters**  
 300 Montgomery St., Suite 621  
 San Francisco, CA 94104  
 USA

Phone: 1-415-371-8234  
 Fax: 1-415-371-8237

**East Coast Sales Office**  
 330 Madison Ave., 9th floor  
 New York, NY 10017  
 USA

Phone: 1-646-495-5076

**Fitech Laboratories Japan**  
 Toranomon40 MT Bldg. 3F  
 5-13-1 Toranomon Minato-ku  
 Tokyo 105-0001, Japan

Phone: +81-3-5402-7711  
 Web: [www.fitechlabs.co.jp](http://www.fitechlabs.co.jp)