Automated Reconciliation for Disconnected Operation in Object Oriented Middleware

Jason Zavislak

A dissertation submitted to the University of Dublin in partial fulfilment of the requirements for the degree of Master of Science in Computer Science

September 12, 2005

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: _____

Jason Zavislak
September 12, 2005

Permission to lend and/or copy

I agree that Trinity College Library may lend or copy this dissertation upon request.

Signed: _____

Jason Zavislak
September 12, 2005

Acknowledgements

I would like to acknowledge Mads Haahr for his contributions in selecting a topic for this dissertation and supervising my work throughout the development of this project. I would also like to acknowledge my fellow NDS classmates for their feedback on my project and especially for the occasional distractions from work.

Abstract

Modern distributed computing systems are no longer constrained to fixed, wired networks. Many computers have gained the ability to move within a network and operate across wireless links. Such mobile computing devices often suffer from limited, unreliable, or expensive network connectivity. These constraints make services dependent on other devices in the network difficult to maintain.

Disconnected operation refers to the operation of a device while disconnected from the network. Typically, disconnected operation is achieved by caching a copy of data and/or functionality from remote machines locally on a device. When network connectivity is lost, the disconnected device can still function by using its cache. Upon reconnection, any modifications are reconciled with the original on the remote machine.

This project focused on implementing a simple disconnected operation protocol and investigating automation of the reconciliation process. Reconciliation generally involves reintegrating disconnected data caches with original copies, while detecting and resolving conflicts that may have occurred during the disconnection period. Rather than relying on application specific routines to detect conflicts, this project attempted to simplify the process by developing a generic means of identifying data dependencies and monitoring data access. The approach chosen was to maintain metadata along with application data to define dependencies between application data and monitor access to the data. By using generic metadata to represent this information, generalised conflict detection algorithms can be implemented.

A successful implementation of these concepts was developed and partially tested for this project. Disconnected operation and limited capabilities of automated conflict detection were proven to work. Significant improvement over similar work in the field was not achieved, but the generic, modular framework created has potential for future development of superior automated conflict detection.

Table of Contents

List of Figures

1. Introduction

1.1 Software Reusability

As software systems have grown in scale over the years, an increasing focus has been put on reusability. Costs of implementing, debugging, and maintaining software can grow exponentially as systems scale. Creating software that is reusable has become essential in optimizing development efforts and ensuring reliability.

Object-oriented design techniques and programming languages have done much to promote the development of reusable software. Languages such as C++ and Java have built in support for object-oriented concepts such encapsulation, inheritance, and polymorphism, allowing users to easily create reusable units of code.

The term ‘middleware’ describes generalized software used to integrate a heterogeneous system in a standardised manner. Middleware is based on the concept of reusability. By separating generic, common functionality from application specific logic, a single middleware can be used across a diverse domain of applications.

Object-oriented middleware combines the benefits of object-oriented programming and reusable middleware. Such middleware packages integrate easily with projects implemented in object-oriented languages and provide support for a wide range of functionality that would otherwise require significant development efforts. Examples include Java RMI (Remote Method Invocation) and CORBA (Common Object Request Broker Architecture), both of which provide a framework for transparently distributing objects across a network. [15]

1.2 Mobile Computing

With the ever increasing usage of wireless networks and portable computers, mobile computing is consistently gaining in popularity. In addition to typical software concerns, mobile computing environments give rise to several other issues. Constraints on mobile devices such as processing power, storage capacity, and power consumption often require careful consideration when developing mobile software. Wireless networking also introduces several challenges beyond those encountered with traditional wired networking. Often, wireless networks suffer from low bandwidth, poor reliability, and diminished availability. Such connectivity issues require efficient and robust applications to function adequately. [3] Movement of mobile clients through a network can also make transparent network communications difficult or impossible.

One of the most challenging problems in mobile computing is loss of network connectivity. Simply having the mobile client cease operation until connectivity is re-established is often unacceptable behaviour. Supporting operations (possibly in a limited or modified form) while disconnected from the network is known 'disconnected operation'. When dealing with disconnected operation, two main issues must be considered: data replication and data reconciliation. Data replication deals with allowing a client to continue to function while removed from the network. Operating through periods of disconnection usually involves local caching of remote data, a process known as 'replication'. Whenever a disconnection occurs, the disconnected client accesses its local replicant rather than the original remote data.

The use of data copies in turn leads to the need for data reconciliation. When a disconnected client rejoins the network, any operations executed using its local replicant must be reconciled with those executed elsewhere in the network using the original data to ensure system wide consistency. The potential for conflicts to occur

during disconnected periods makes this problem particularly difficult to solve. This project examines disconnected operation in object-oriented middleware and explores a technique for automating data reconciliation.

1.3 Motivation

The primary motivation for this work is to further the development of automated data reconciliation techniques to improve the robustness and ease of programmability of applications using disconnected operation with object-oriented middleware. Various software packages exist for supporting disconnected operation, but in general data reconciliation is still a manual process. Typically, conflict detection is accomplished by using application specific logic or generically using overly simplistic algorithms. Similarly, conflict resolution usually relies exclusively on application logic. This project attempts to automate some of these processes by extending middleware functionality, thus reducing the programming effort required by applications.

1.4 Objectives

The primary objective of this project is to create a generic, reusable disconnected operation toolkit capable of performing semi-automated data reconciliation. This objective relies also on several secondary objectives. In order to automate reconciliation, a method of generically describing object interdependencies and data access must be defined. This project implements a framework for specifying object dependencies and access history using generalised metadata. Automated reconciliation also requires integration of application logic to handle conflict

resolution. Consequently, a framework for implementing conflict resolution is also included. Note that this project focuses on improving automation of conflict detection and not conflict resolution. The resolution of conflicts is viewed as too application specific to be worth attempting generalisation. It should also be mentioned that this project attempts to maintain simplicity in design and implementation in hopes off creating a more robust and easier to use software package. Priority is placed on making this software intuitive and user friendly for developers.

Some notable issues outside the scope of this project include client/server communications, including detection of disconnection/reconnection, and application concurrency. Client/Server communications are implemented by the application or middleware and integrated using an interface specified in this library. Connection awareness is, therefore, left to the application programmer. Within the frameworks developed here, there is no knowledge of connection state. It is the application's responsibility to decide when to use the caching framework and when to use normal client/server invocations. Finally, none of the classes developed in this project should be assumed thread-safe. Applications utilizing concurrency should manage serialised access to all classes in this library.

2. State of the Art

2.1 Coda

Coda is a distributed file system, based on the Andrew File System (AFS), with built in support for mobile computing. This support includes allowing disconnected operation for mobile clients. Coda design is based on a client/server architecture, with clients accessing files from one or more file servers on a network. Disconnected operation is achieved by caching files on clients when they are initially accessed. Future calls can then access the local copy, thus reducing required network communications. When a cached file is modified, the changes must be reported to the server. If the client is in connected mode, changes are synchronously sent to the server. While disconnected, changes are instead logged on the client machine, and then sent to the server upon reconnection. This also allows for potential optimisation of the client log before it is sent to the server (for example, the creation of a file followed by the deletion of the file would result in no net change to the system, so both operations could be removed from the log). Coda works based on the assumption that there are few write conflicts in most file systems. Given this assumption, deferring client writes while they are disconnected is relatively safe. In the event of a conflict (i.e. a disconnected client and another machine update the same file), the server can either invoke an application specific resolver algorithm or report the conflict for human correction. [3-10]

2.2 Bayou

Bayou is a distributed mobile database system designed for use on networks with frequent disconnections. Similar to the Coda file system, databases are cached on mobile clients to allow disconnected operation. Updates are allowed on disconnected clients and a peer-to-peer anti-entropy algorithm is used to propagate changes. The anti-entropy algorithm ensures that all replicated databases will eventually reach a consistent state, while the peer-to-peer aspect of the algorithm allows any two communicating machines to exchange updates. The peer-to-peer approach has the advantage of removing client dependencies on a single server, a feature that is especially useful on frequently partitioned networks. Update conflicts in Bayou are handled using application specific dependency sets. Structured as a set of queries and expected return values, dependency sets allow the Bayou system to detect update conflicts and delegate conflict resolution to application defined merge routines. [11-14]

2.3 Rover

Rover is a toolkit for developing applications for mobile, or roving, computers. Rover is based on the concept of a relocatable dynamic object (RDO). A RDO is basically an object-oriented caching mechanism. Each RDO encapsulates a set of data, methods, and possibly execution threads. At runtime, RDOs can be moved between machines, thereby minimizing communications required for clients accessing the RDOs. A technique called Queued Remote Procedure Call (QRPC) is also used to allow disconnected operation. QRPC works by allowing disconnected clients to make non-blocking requests which are then queued until reconnection. Once reconnection occurs, the queued requests are reconciled with the server.

The basic Rover caching algorithm uses a check-in, check-out approach. Each RDO has a designated home server which maintains the master state of the object and services relocation requests. To create a local replicant of a RDO, a client imports the object from its home server. The client may then invoke methods on the local RDO without communicating with the server. Modifications to replicant RDOs are considered tentative until reconciled with the server. When all client invocations are finished, the RDO is exported back to its home server. Upon receipt of an exported RDO, the home server checks for conflicts before integrating the updated RDO. Consistency between shared RDOs may be maintained through application level locking. If conflicts are not avoided through locking, application specific algorithms must be implemented to resolve any conflicts detected by the home server. [1-2]

2.4 ALICE

The Architecture for Location-Independent Computing Environments (ALICE) is a framework for adding mobility support to client/server protocols. ALICE may be used to enhance any protocol satisfying a set of requirements defined in the architecture specification. The architecture describes three types of components: mobile hosts (MH), remote hosts (RH), and mobile gateways (MG). A MH is a device capable of moving within a network. A RH is any device (fixed or mobile) that a MH communicates with. MHs and RHs may function as clients or servers. A MG is a fixed device capable of communicating with both MHs and RHs. ALICE handles movement of MHs by routing communications between each MH and RH through the MG. Using this level of redirection, applications are relieved of the responsibility of maintaining explicit location of MHs and RHs. ALICE includes a

disconnected operation layer which allows replication of server functionality such that clients can operate while disconnected from the network. [16]

3. Analysis and Design

3.1 Design Overview

All the disconnected operation approaches mentioned above have similar design characteristics. The basic method always involves copying server state (and possibly functionality) to a client. The client can then access the local copy while not connected to the server. Upon reconnection, local copies are reconciled with the server to ensure consistency across the system. The design of this project follows a similar design pattern.

The primary focus of this project is to investigate improving the reconciliation support offered after disconnected operation. A typical client/server architecture is used to share data locally or across a network. A simple caching mechanism is provided to maintain replicated server state on clients. Metadata is kept with user data to record data access on both the server and clients. During reconciliation, this metadata is used to detect access conflicts and invoke application defined resolution methods. Applications interact with the framework primarily through the client and server caches. The overall design is meant to be generic and independent of any particular application or middleware. Well defined interfaces are used as integration points for applications. Actions such as message transport are left to the application to implement. Object-oriented design is used throughout the framework, with the intended usage being integration with object-oriented middlewares.

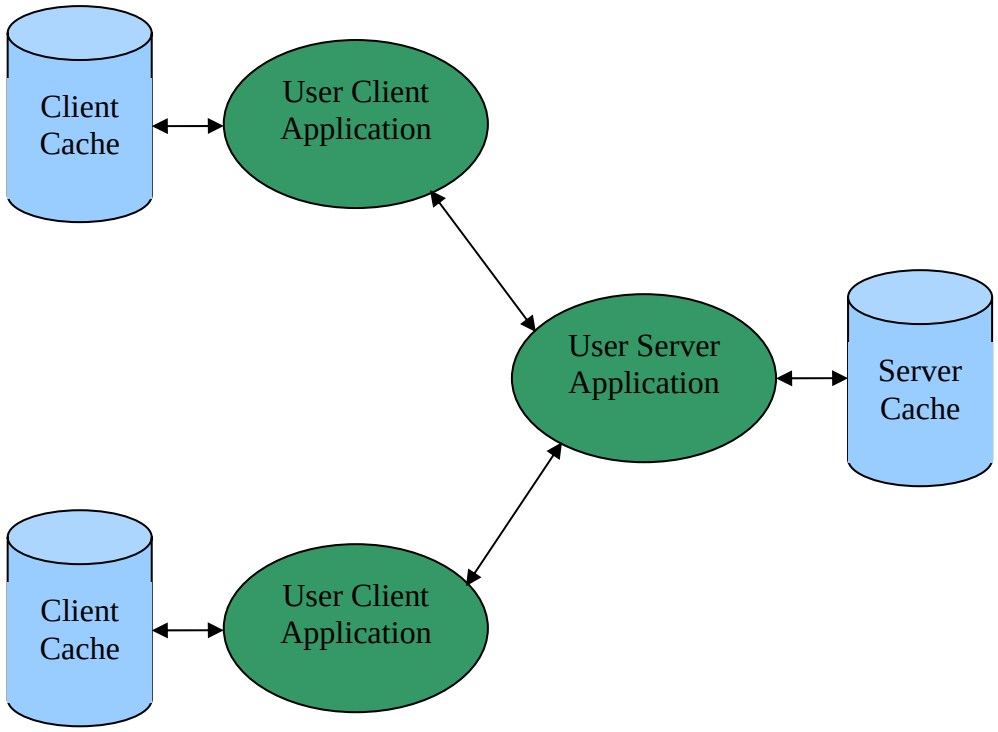

Figure 3.1: Architecture Overview

3.2 Data Structures

The most basic data structure in the framework is the Cached Object. A Cached Object is an encapsulation of a serialized user object (an object converted to a network communicable byte sequence) along with metadata describing that object. Attributes include a system wide unique identifier and a version identifier.

Cached Objects are managed by a container object called an Object Set. Object Sets are used to group Cached Objects according to their interdependencies. In the current design, all Cached Objects are required to be contained by exactly one Object Set. Object Set dependencies are defined as ‘read’, ‘write’, or ‘read/write’. All Cached Objects within an Object Set are defined as having the same type of dependency. For example, all objects in an Object Set with a read/write dependency

are assumed to be dependent on read and write access of all other objects in the set. All Cached Objects having any type of dependency relationship must be in the same object set as their dependents. In addition to the contained Cached Objects, each Object Set also maintains metadata describing characteristics of the set. Like the Cached Object, an Object Set is assigned a system wide unique identifier and a version identifier. The version identifier for an Object Set and each Cached Object contained by the set are identical. Access to the contained Cached Objects is also recorded in the Object Set. Both read and write access metadata are maintained in Object Sets, on a per set basis. In other words, if any object in a set is read, the set is marked as 'read'. Object Set metadata forms the basis for conflict detection.

The highest level data structure is the Cache. Two variants of the Cache are defined. The Server Cache is found only on the server and contains the master set of data. Client Caches are found on all clients and contain a copy of the Server Cache. A Cache contains one or more Object Sets and gives the user a single point of access to all Object Sets. In general, besides initializing the Server Cache with a group of Object Sets (see section 3.4), the user does not interact directly with Object Sets. The primary user interface deals only with Cached Objects and Caches.

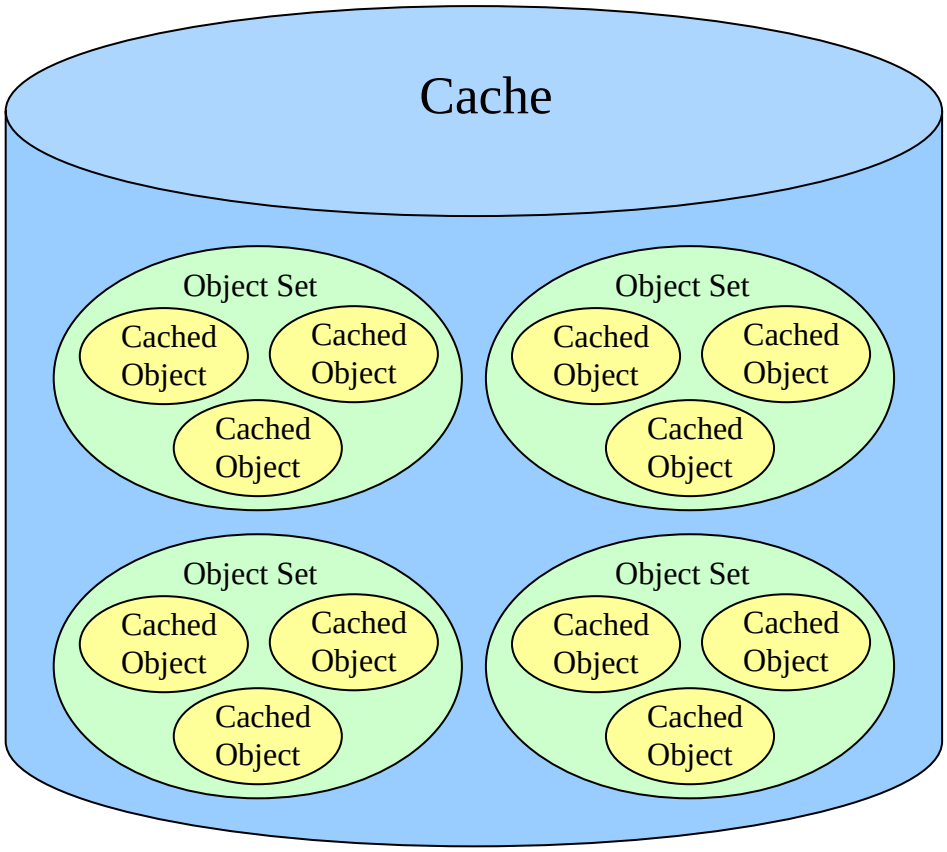

Figure 3.2: Cache Structure

3.3 Application Integration

The Cache classes provide the primary application interface and perform high level management of cached object data. To carry out these tasks, some application specific implementation is required. To isolate application specific details from the generic framework, interfaces are used. To create Cache objects, applications must implement the specified interfaces and use them to construct the Cache objects. The Cache interfaces are of two types: Communicators and Administrators. Communicators are responsible for transmitting messages between the Client and Server Caches. A Client Communicator interface and a Server Communicator

interface are defined for the Client and Server Cache respectively. Both interfaces have a ‘sendMessage’ method which must be implemented to reliably transmit a given message to the remote Cache. The second interface type is the Cache Administrator. The Client and Server Cache Administrators define application specific administration routines. The Client Administrator interface has a ‘notifyUpdate’ method to notify the application of an update to the Cache and a ‘getNextVersionId’ method to generate version identifiers for client updates. The Server Administrator has the same ‘notifyUpdate’ method plus ‘initializeCache’ to initialize the Server Cache, ‘getNextVersionId’ to generate Object Set version identifiers (used for conflict detection, see sections 3.6 and 3.7), and ‘resolve’ to resolve update conflicts.

Note that none of the Communicator or Cache Administrator methods are called directly by the user. Implementations of the interfaces are registered with the Cache when it is constructed and all calls to the interfaces are made through the Cache framework. Further details on usage of the Communicator and Cache Administrator interfaces are given in the following sections.

3.4 Server Cache Initialization

For the caching/replication mechanism to operate, the system must be bootstrapped with an initialized Server Cache. Once the server side has been initialized, the replication process will distribute the cache to all clients. Initializing the Server Cache is a manual process which must be implemented by the application using the Server Cache Administrator ‘initializeCache’ interface. Implementation of this method consists of creating Object Sets, adding Cached Objects to the sets, and finally adding the Object Sets to the Server Cache. Note that creating Cached Objects

involves serialization of application defined objects. Once the Server Cache has been constructed with the Server Cache Administrator implementation, calling the 'initialize' method on the Server Cache invokes the above mentioned administrator call.

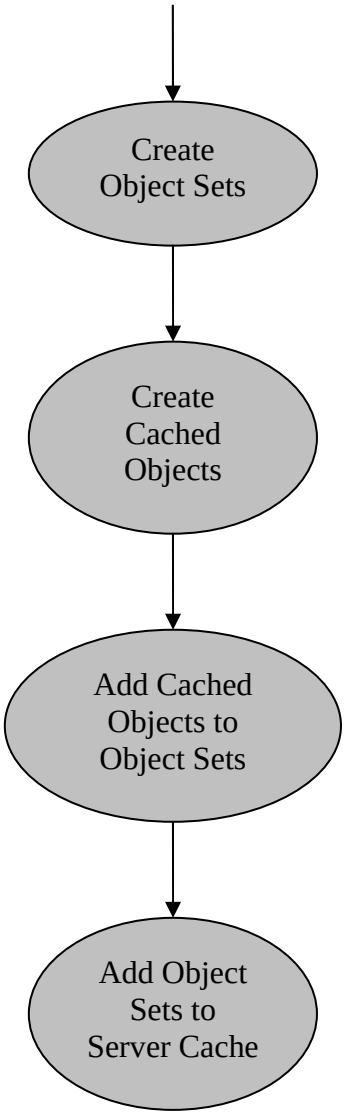

Figure 3.3: Server Cache Initialization

3.5 Client Cache Initialization

Initialization of the Client Cache is simply a matter of requesting an update from the Server Cache. The Client Cache method 'requestServerUpdate' must be called before using any other methods on the cache. This method is really a synonym for the 'sendClientUpdate' method used in the client update protocol. The process of sending a client update involves receiving a server update as acknowledgement (see section 3.7). Therefore, sending an initial empty client update triggers a server update to initialize the client cache.

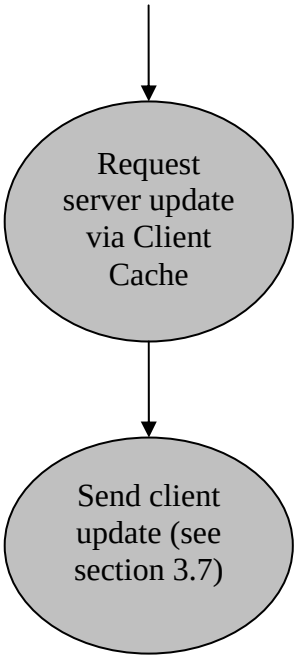

Figure 3.4: Client Cache Initialization

3.6 Client Object Access

Accessing objects from the Client Cache is done through the 'getObject' and 'setObject methods. To retrieve a Cached Object from the cache, the 'getObject' method is called with the identifier of the desired object. The method first maps the given object id to the Object Set owning the id. Note that membership of an object id in an Object Set only indicates the ability of the set to hold the object of that id. The

set may or may not actually contain the object instance. If the object id is not found in any Object Set in the cache, the ‘Object Not Found’ exception is thrown. Otherwise, the appropriate Object Set is queried for the object. If the set does not currently contain the object instance, the ‘Object Not Found’ exception is thrown. Otherwise, the Cached Object is returned to the application. From the Cached Object, the application must extract and deserialize the object data to access the application defined object. Getting an object from the cache results in its Object Set being marked as ‘read’ (see section 3.7).

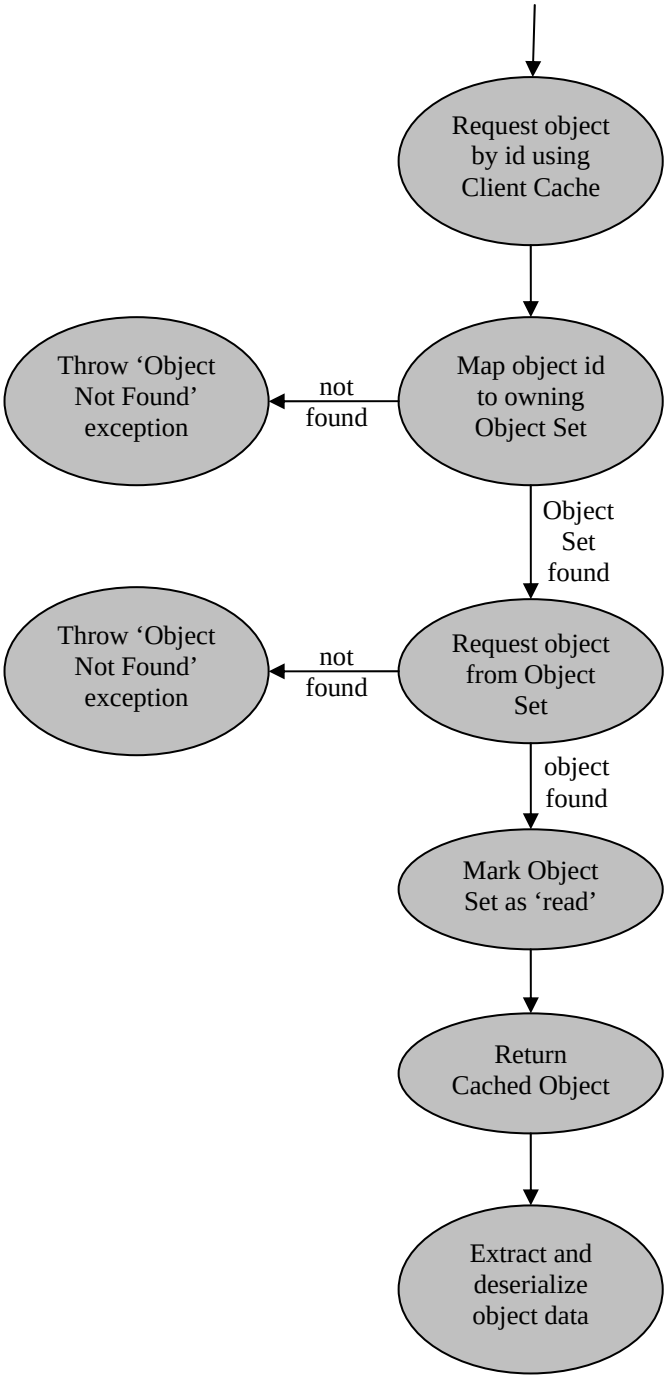

Figure 3.5: Client Object Get

To modify an object in the cache, the 'setObject' method is used. The set method requires the application object to be encapsulated within a Cached Object. Therefore the first step in setting an object is to obtain a Cached Object using the 'getObject' method. The application object, in serialized form, can then be replaced

in the Cached Object. Next ‘setObject’ is called with the updated Cached Object. Like the ‘getObject’ method, ‘setObject’ maps the Cached Object’s identifier to the appropriate Object Set. If no Object Set owns the object id, the ‘Object Not Found’ exception is thrown. This should never happen, assuming the ‘getObject’ method was used to retrieve the Cached Object. Before replacing the Cached Object in the cache, its version identifier is compared to the version id of its Object Set. The Cached Object contains the version id of its Object Set from the time it was retrieved with the ‘getObject’ call. A mismatch of the version id of a Cached Object and the version id of its Object Set at the time of a ‘setObject’ call indicates that the Object Set has been updated by the server since the object was read from the cache (i.e. since the ‘getObject’ method was called). In this scenario, the ‘Object Access Conflict’ exception is thrown to ensure that no updates to the Object Set are unintentionally overwritten. If the version ids are a match, the updated Cached Object is copied into the cache. Setting an object in the cache results in its Object Set being marked as both ‘read’ and ‘written’ (see section 3.7).

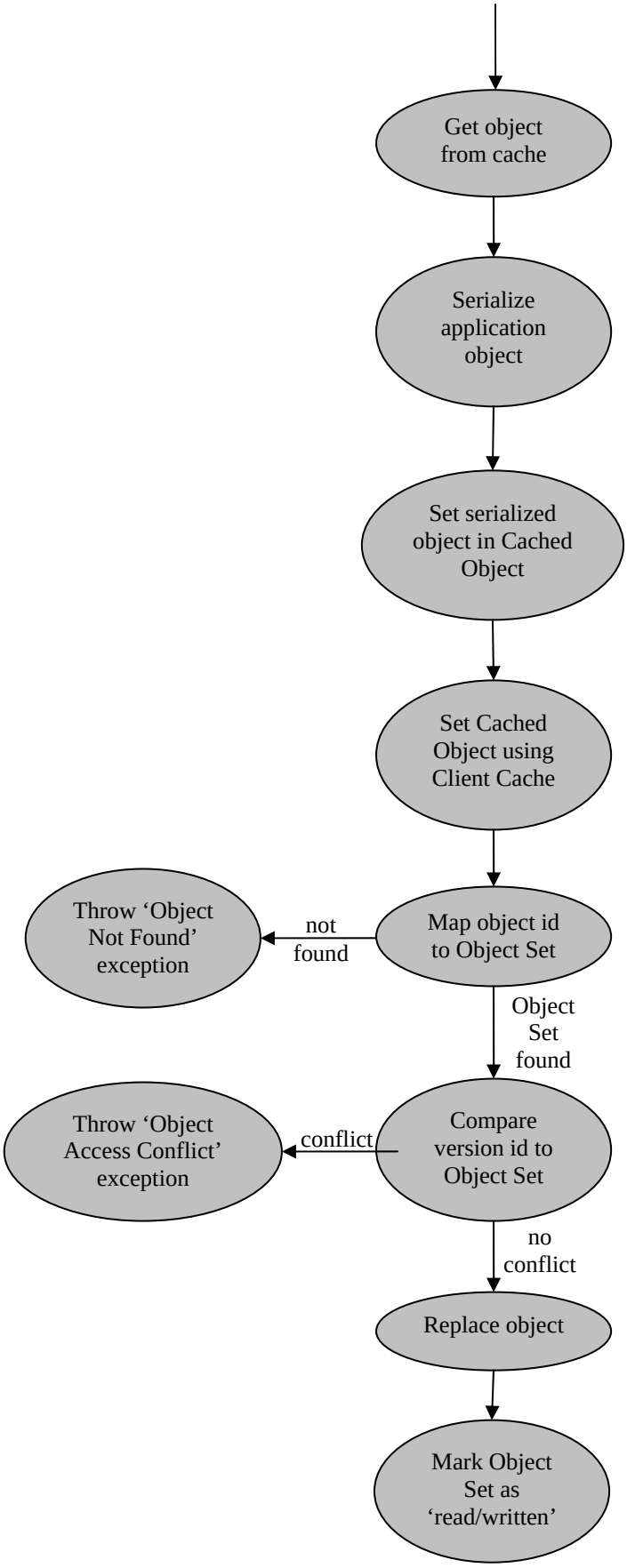

Figure 3.6: Client Object Set

3.7 Cache Replication and Synchronisation

3.7.1 Overview

Robust cache replication and synchronisation are essential to the usefulness of this software library. The replication process allows the distribution and synchronisation of the Server Cache to all clients in the system. Once the Server Cache is initialized (see section 3.4), synchronisation can be initiated from either the client or server side. Typically, clients initiate synchronisation to obtain an initial copy of the Server Cache (see section 3.5), when client updates occur while in connected mode, and when the client reconnects after being in disconnected mode. The server, on the other hand, generally only initiates synchronisation when an update is received from a client (i.e. the client initiated synchronisation). In this case, in addition to acknowledging the client with an update of its own, the server also initiates synchronisation with all other clients in the system to distribute the update it has just received.

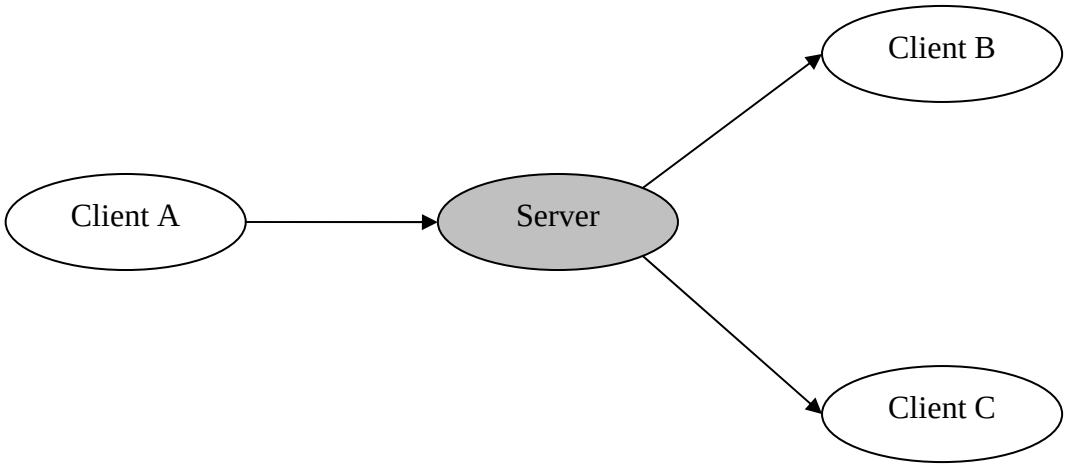

Figure 3.7: Client Update Distribution

Regardless of whether synchronisation is initiated from the client or server, both sides behave in the same manner. The client and server exchange update messages until the Client and Server Caches are equivalent. During the synchronisation process, conflicts may be detected. Conflicts are detected at the granularity of an Object Set. Typically, conflicts occur when multiple clients access the same Object Set in their local cache. When both attempt to propagate the changes, the updates will conflict.

Whether or not an Object Set is considered to have been accessed depends on the dependency type defined for the set. Each Object Set specifies a 'read', 'write', or 'read/write' dependency between its contained objects. Therefore, if an Object Set has a 'read' dependency type and one or more objects have been read from it (via the 'getObject' method, see section 3.6), it is considered to be accessed. Likewise, if an Object Set has a 'write' dependency type and one or more objects have been written to it (via the 'setObject' method, see section 3.6), it is considered to be accessed. In addition to access state, each Object Set also maintains a server assigned version id. The version id identifies the version of the Object Set as of the when it was copied from the Server Cache. The access state of an Object Set represents its access relative to the version of the set at the time it was replicated from the server. This version stamping is the basis for conflict detection.

Conflicts can be detected by both the Client and Server Caches. However, only the Server Cache can resolve a conflict. Conflicted updates detected on a client are simply discarded and the client is triggered to send its updates to the server for reconciliation. Due to the possibility of conflicts, the synchronisation process is designed to be iterative (see figure 3.8). Once synchronisation is started, the clients

and server exchange updates until a common state is achieved. More detailed descriptions of client and server updates are given in the following sections.

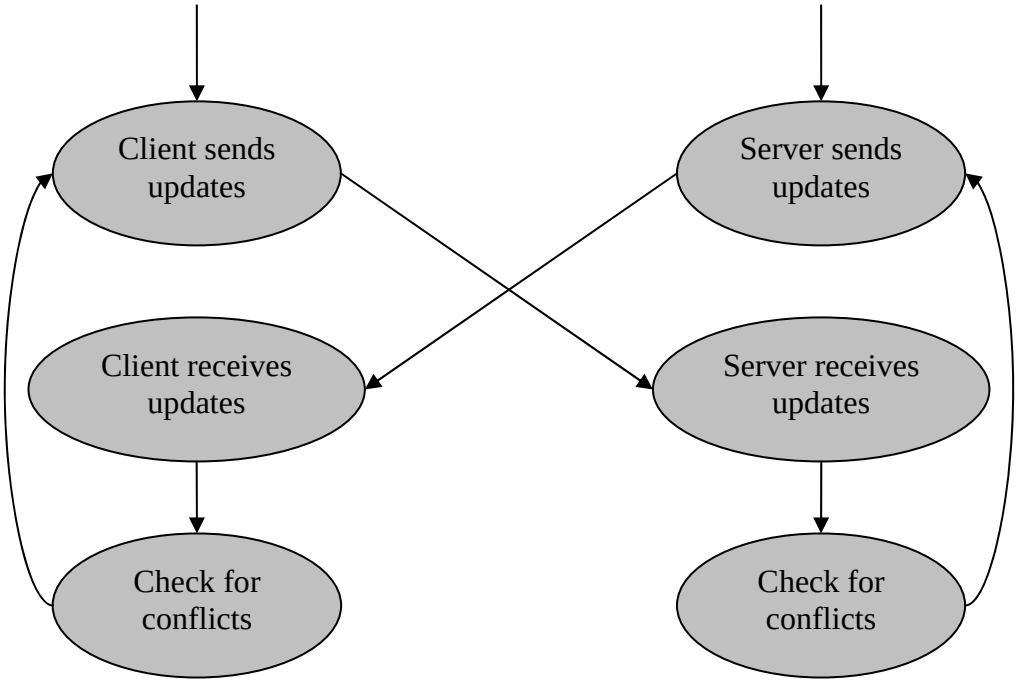

Figure 3.8: Cache Synchronisation Process

3.7.2 Client Update

Client updates occur in two parts. On the client side, the Client Cache method 'sendClientUpdate' is responsible for creating an update message and sending it to the server. On the server side, the Server Cache processes the client update message, looking for conflicts and applying changes to the Server Cache.

To start the update process, the client side application first calls the 'sendClientUpdate' method on the Client Cache. This method generates a version id using the Client Cache Administrator interface. The version id identifies the current state of the entire Client Cache and is used in an acknowledgement from the server

after the client updates have been processed. After obtaining the version id, the cache builds a message containing its id, the version id, and all Object Sets that have been accessed.

Once all accessed Object Sets have been added to the update message, the access states of those Object Sets are tentatively cleared. The access states will only be fully cleared once acknowledgement has been received (in the form of a server update, see section 3.7.3). In the meantime, the tentative access state indicates which updates have been sent to the server, but not acknowledged. This allows tentative updates to be resent if another client update is sent before the previous one is acknowledged. Finally, the update message is sent to the server using the Client Communicator interface.

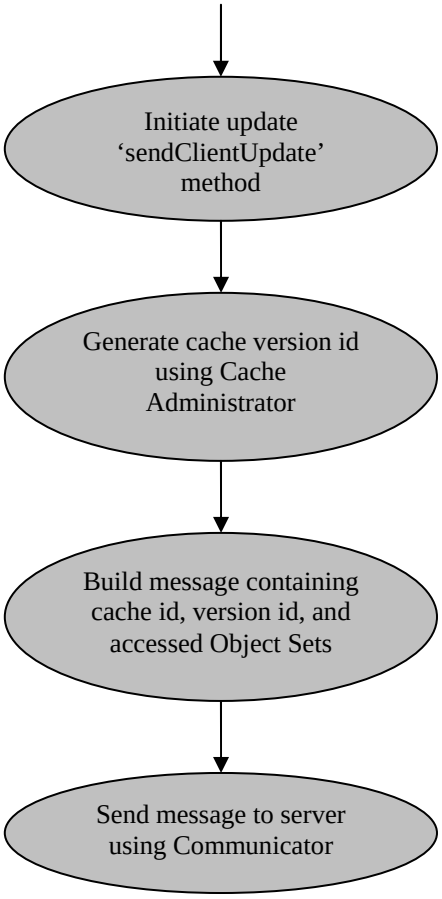

Figure 3.9: Client Update (client side)

Once the client update message is received by the server, it is passed to the Server Cache ‘processClientUpdate’ method. The version id of each Object Set in the message is compared to the version id of the Object Set in the Server Cache. Any mismatches indicate a conflict. Conflict resolution is delegated to the ‘resolve’ method on the Server Cache Administrator interface. Once the Object Set versions are verified and any conflicts are resolved, the Server Cache is updated with the new Object Set. A new version id is generated using the Administrator interface and assigned to the Object Set. Notification of the update is forwarded to the server side application via the Administrator ‘notifyUpdate’ method. To complete the synchronisation, acknowledgement of the update is sent to the originating client by sending a server update message (see section 3.7.3) containing the Client Cache version id sent in the client update message. The server update will include the new version ids assigned to the updated Object Sets. An identical server update is also sent to all other clients in the system to ensure consistency.

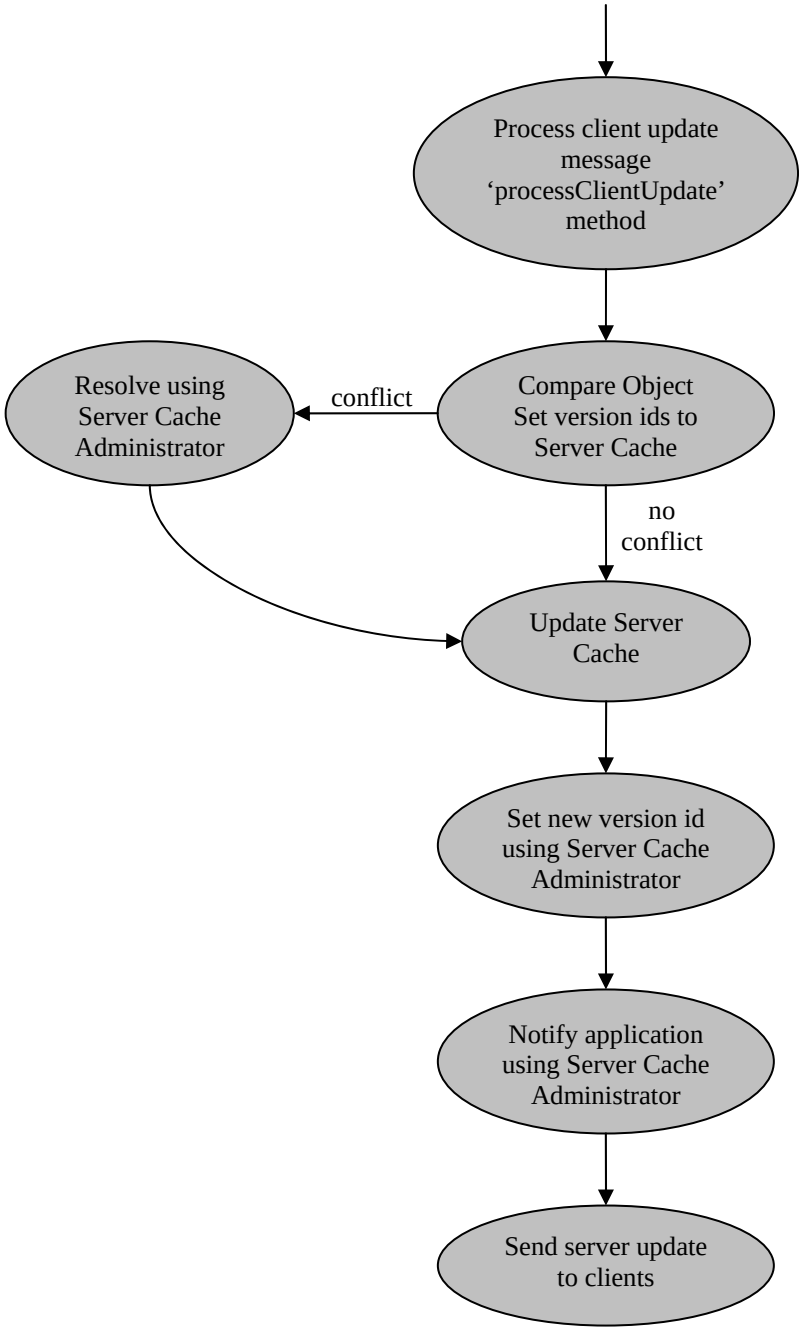

Figure 3.10: Client Update (server side)

3.7.3 Server Update

Server updates occur in two parts. On the server side, the Server Cache method 'sendServerUpdate' is responsible for creating an update message and

sending it to the client. On the client side, the Client Cache processes the server update message, looking for conflicts and applying changes to the Client Cache.

The update process is started by calling the 'sendServerUpdate' method on the Server Cache. This method can be called manually by a server side application, or automatically by the Server Cache to acknowledge and distribute a client update (see section 3.7.2). The 'sendServerUpdate' method builds a message containing all Object Sets in the Server Cache. If the 'sendServerUpdate' method is being executed to acknowledge a client update, the Client Cache version id is also included in the message. After constructing the message, it is transmitted to the client(s) using the Server Communicator interface.

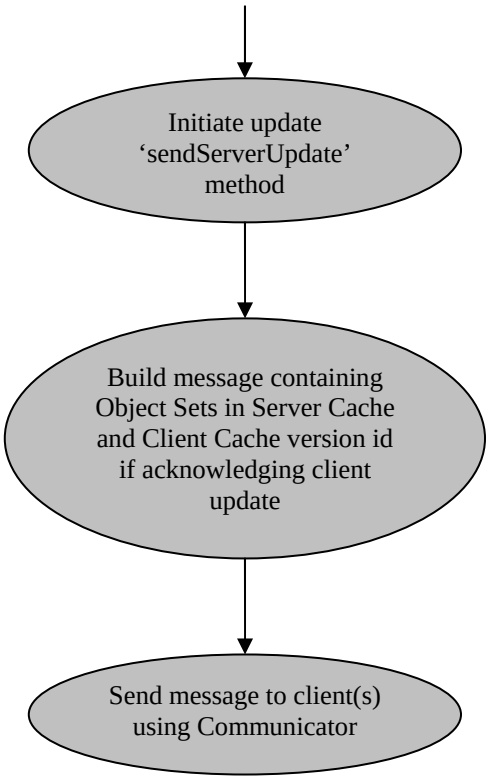

Figure 3.11: Server Update (server side)

When the server update message is received by the client, it is passed to the Client Cache method ‘processServerUpdate’. This method first checks if the Client Cache was expecting an acknowledgement for a client update. If an acknowledgement is expected, the server update message is checked for the correct client cache version id. If the version id does not match, the update is ignored, with the assumption that the correctly versioned update will follow. If the version id is a match, or if no acknowledgement is expected (an autonomous update from the server), the Object Sets in the message are examined. The version id of each Object Set in the update message is compared to the corresponding version id of the local Object Set copy. A mismatch of the version ids indicates that the local copy of the Object Set has been accessed while the server copy of the Object Set was also accessed. Such a conflict can not be resolved by the Client Cache (conflict resolution can only be done by the Server Cache) so the Object Set is not updated in the Client Cache. Furthermore, the Client Cache is triggered to send a client update to the server to allow the conflict to be resolved. If the version ids of an Object Set do not conflict, the Object Set is updated in the Client Cache. Notification of updates is communicated to the client application using the ‘notifyUpdate’ method of the Client Cache Administrator.

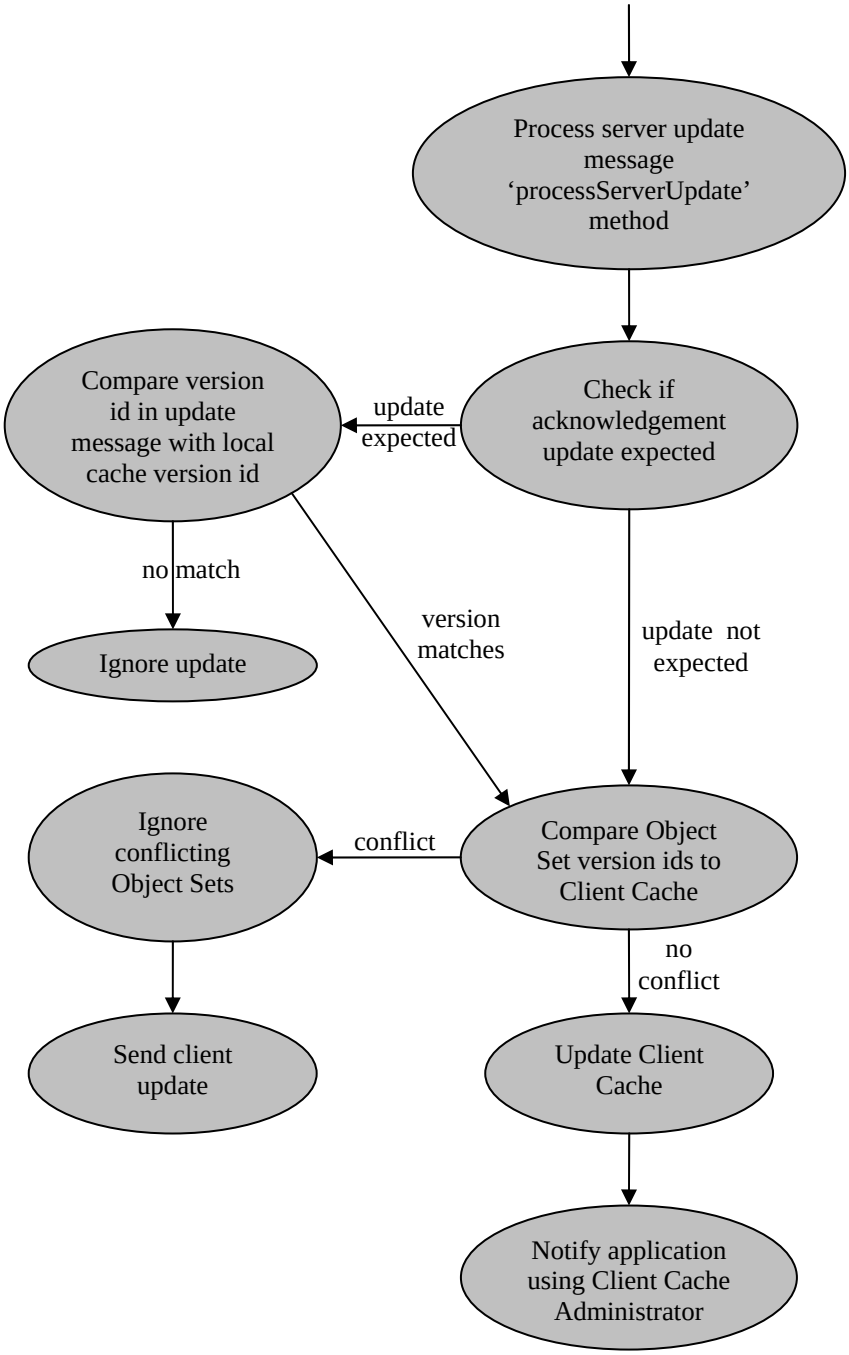

Figure 3.12: Server Update (client side)

3.7.4 Conflict Resolution

Access conflicts can be automatically detected by the Server Cache during client updates (see section 3.7.2). However, resolving a conflict requires application

specific logic. The ‘resolve’ method on the Server Cache Administrator interface permits integration of such application logic into the Server Cache framework. The ‘resolve’ method is passed the original Object Set (server’s copy), the conflicting Object Set (client’s copy), and an empty Object Set. The ‘resolve’ interface must be implemented to evaluate the client and server Object Sets and resolve the conflict. The results of the conflict resolution are placed in the empty Object Set which is then returned to the Server Cache.

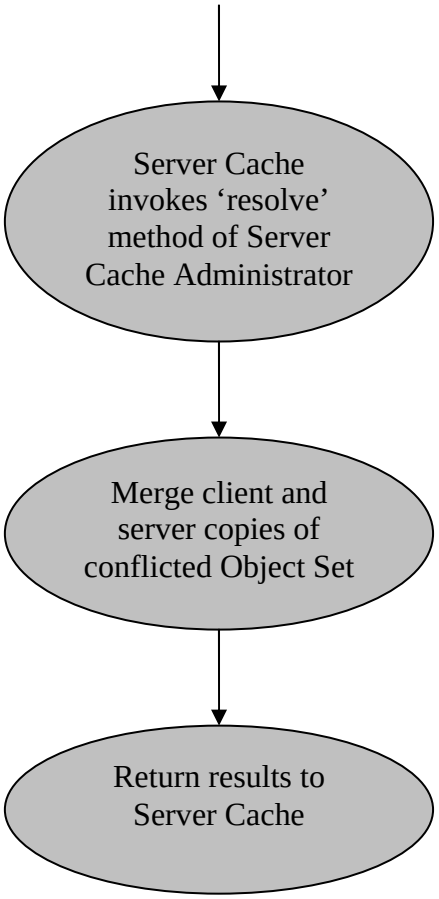

Figure 3.13: Conflict Resolution

4. Implementation

4.1 Programming Language

C++ was chosen as the programming language. In the interest of keeping the implementation as generic as possible, only standard C++ libraries are used. No operating system specific APIs (threads, semaphores, etc.) are used.

4.2 Development Environment

The RedHat Cygwin Linux emulator for Microsoft Windows was used as the development environment. Standard gnu utilities (*g++*, *gdb*) and *make* were used for development and testing purposes. Refer to Appendix C for installation guidelines.

Significant problems were encountered using the Cygwin platform. The gdb debugging utility did not work reliably when using the POSIX pthreads library. Consequently, much of the code developed during the testing phase of this project was debugged without the aid of a debugger, significantly increasing the effort.

4.3 Implementation Decisions

Several notable implementation decisions were made in the development of this project.

- All object and version identifiers are of type std::string from the C++ standard template library.
- The C++ standard template library std::map class is used for cache lookups.
- Serialized object data (byte sequences) are defined as type (char *)

4.4 Class API

This section outlines the primary set of classes implemented according to the design described in section 3. A brief description of the public API of each class is also given.

4.4.1 CachedObj

The CachedObj class implements the Cached Object design. CachedObj is used to encapsulate serialized application objects. This class is used on the server side to initialize the Server Cache and on the client side when getting and setting objects in the Client Cache.

Constructors

Default constructor
```
CachedObj()
```

Construct with id and serialized object data
```
CachedObj(
    const std::string & anId,
    const char * anObject,
    const long anObjectSize
    )
```

Copy constructor
```
CachedObj(
    const CachedObj & aCachedObj
    )
```

Accessors

Get serialized object data
```
const char * getObject() const
```

Set serialized object data
```
void setObject(
    const char * anObject,
    const long anObjectSize
```

)

Stringify CachedObj for display
std::string toString() const

4.4.2 ObjectSet

The ObjectSet class implements the Object Set design. ObjectSet encapsulates a group of interdependent CachedObj objects as well as versioning and access metadata used for conflict detection. The ObjectSet class is used on the server to initialize the Server Cache.

Constructors

Default constructor
ObjectSet(
    const std::string anId = "",
    const std::string aVersionId = "",
    const DependencyType aDependencyType = DependencyType::WRITE
)

Copy constructor
ObjectSet(
    const ObjectSet & anObjectSet
)

Accessors

Add object id to set (just id, not object instance)
void addObjectId(
    const std::string & anObjectId
)

Add/replace object instance in set
void setObject(
    CachedObj & anObject
)

Stringify ObjectSet for display
std::string toString() const

4.4.3 ClientCache

The ClientCache class implements the Client Cache design. ClientCache encapsulates a set of ObjectSet objects replicated from the Server Cache. The ClientCache class is used on the client to get and set CachedObj objects, to request a server update (initialize cache), to send client updates to the server, and to process server updates.

Constructors

Construct ClientCache with id, communicator implementation, and administrator implementation
```
ClientCache(
    const std::string anId,
    ClientCommunicator & aClientCommunicator,
    ClientCacheAdministrator & anAdmin
    )
```

Accessors

Get object from cache
```
CachedObj getObject(
    const std::string anId
    )
```

Set object in cache
```
void setObject(
    CachedObj & anObject
    )
```

Synchronisation

Request update from server (initialize client cache)
```
void requestServerUpdate()
```

Send update to server
```
sendClientUpdate()
```

Process update from server
```
void processServerUpdate(
    const char * aServerUpdate
    )
```

4.4.4 ClientCommunicator

The ClientCommunicator class implements the Client Communicator design. ClientCommunicator is an abstract class which must be implemented to construct a ClientCache object. The ClientCommunicator class is used by the ClientCache class to send messages to the server.

Message Transmission

```
Send a message to the server
virtual void sendMessage(
    const char * aMessage,
    const long aMessageSize
    ) = 0
```

4.4.5 ClientCacheAdministrator

The ClientCacheAdministrator class implements the Client Cache Administrator design. ClientCacheAdministrator is an abstract class which must be implemented to construct a ClientCache object. The ClientCacheAdministrator class is used by the ClientCache class to notify the client application of cache updates and to generate cache version identifiers used in the synchronisation protocol.

Cache Administration

```
Notify the client application of a cache update
virtual void notifyUpdate(
    const ObjectSet & anObjectSet
    ) = 0

Generate a version id for use in the synchronisation protocol
virtual std::string getNextVersionId(
    const std::string aCurrentVersionId = ""
    ) = 0
```

4.4.6 ServerCache

The ServerCache class implements the Server Cache design. ServerCache encapsulates the master set of ObjectSet objects replicated to Client Caches. The ServerCache class is used on the server to initialize the system cache, to send server updates to clients, and to process client updates.

Constructors

Construct ServerCache with communicator implementation and administrator implementation

```
ServerCache(
    ServerCommunicator & aServerCommunicator,
    ServerCacheAdministrator & anAdmin
    )
```

Synchronisation

Initialize ServerCache

```
void initialize()
```

Send update to one or more clients

```
void sendServerUpdate(
    const std::string aClientId = "",
    const std::string aClientVersionId = ""
    )
```

Process update from client

```
void processClientUpdate(
    const char * aClientUpdate
    )
```

4.4.7 ServerCommunicator

The ServerCommunicator class implements the Server Communicator design. ServerCommunicator is an abstract class which must be implemented to construct a ServerCache object. The ServerCommunicator class is used by the ServerCache class to send messages to the client.

Message Transmission

Send a message to the client
```
virtual void sendMessage(
    const char * aMessage,
    const long aMessageSize
    ) = 0
```

4.4.8 ServerCacheAdministrator

The ServerCacheAdministrator class implements the Server Cache Administrator design. ServerCacheAdministrator is an abstract class which must be implemented to construct a ServerCache object. The ServerCacheAdministrator class is used by the ServerCache class to initialize the ServerCache, to resolve conflicts during client updates, to notify the server application of cache updates, and to generate cache version identifiers used in the synchronisation protocol.

Cache Administration

Initialize the ServerCache
```
virtual void initializeCache(
    ServerCache & aServerCache
    ) = 0
```

Resolve conflict detected during client update
```
virtual void resolve(
    const ObjectSet & anObjectSet,
    const ObjectSet & aConflictingSet,
    ObjectSet & aResolvedSet
    ) = 0
```

Notify the server application of a cache update
```
virtual void notifyUpdate(
    const ObjectSet & anObjectSet
    ) = 0
```

Generate a version id for use in the synchronisation protocol
```
virtual std::string getNextVersionId(
    const std::string aCurrentVersionId = ""
    ) = 0
```

5. Results and Evaluation

5.1 Test Environment

The software libraries developed in this project were tested using a simulated middleware environment. The following test classes were implemented.

5.1.1 Data Structures

TestObject – A simple class encapsulating a std::string, capable of serialization.

TestObjectStore – A container of TestObjects, used primarily as a convenient method of retrieving, deserializing, and displaying a set of TestObjects.

SocketCommunicator – A façade class encapsulating a simple UDP socket interface. Local host and fixed port numbers are used for testing.

TestClientCommunicator – An implementation of the ClientCommunicator interface using the SocketCommunicator as a transport mechanism.

TestClientCacheAdmin – An implementation of the ClientCacheAdministrator interface. Version ids are implemented as consecutive stringified integers, beginning with '1'.

TestServerCommunicator – An implementation of the ServerCommunicator interface using the SocketCommunicator as a transport mechanism.

TestServerCacheAdmin – An implementation of the ServerCacheAdministrator interface. Initialization of the ServerCache creates six CachedObj objects, each containing a serialized TestObject. The six TestObject/CachedObj objects, named 'A' through 'F', are placed in two ObjectSet objects, named '1' and '2'. Objects 'A', 'B', and 'C' are in ObjectSet '1' and objects 'C', 'D', and 'E' are in ObjectSet '2'. Conflict resolution is handled by keeping only the last written changes. In other words, no merging is done. Version ids are implemented as consecutive stringified integers, beginning with '1'.

5.1.2 Test Applications

Two menu driven command line test applications are implemented, one simulating a client and one simulating a server.

5.1.3 Client Test Application

The Client Test Application creates a ClientCache object with a TestClientCommunicator and a TestClientAdmin object. A TestObjectStore object is created to simplify retrieving, deserializing, and printing TestObjects 'A' through 'F'. The ClientCache 'requestServerUpdate' method is called to initialize the ClientCache. Note that the Server Test Application must be started before the Client Test Application. Following cache initialization, the program spawns a thread (using the POSIX pthreads library) to listen for and process server updates using the SocketCommunicator class. The main application thread then enters a menu loop with the following selections: Print cache, Set object, Disconnect client, and Connect

client. ‘Print cache’ retrieves and displays the six TestObjects using the TestObjectStore. ‘Set Object’ calls the ClientCache ‘setObject’ method. ‘Disconnect client’ simulates a network disconnection on the client by intercepting the SocketCommunicator ‘send’ and ‘receive’ methods used by the TestClientCommunicator. ‘Connect client’ undoes ‘Disconnect client’ and sends a client update. A pthread mutex is used to ensure safe access to the ClientCache object from the two application threads.

5.1.4 Server Test Application

The Server Test Application creates a ServerCache object with a TestServerCommunicator and a TestServerAdmin object. A TestObjectStore object is created to simplify retrieving, deserializing, and printing TestObjects ‘A’ through ‘F’. The ServerCache ‘initialize’ method is called to initialize the ServerCache. Note that the Server Test Application must be started before the Client Test Application. Following cache initialization, the program spawns a thread (using the POSIX pthreads library) to listen for and process client updates using the SocketCommunicator class. The main application thread then enters a menu loop with the following selections: Print cache, Disconnect server, and Connect server. ‘Print cache’ retrieves and displays the six TestObjects using the TestObjectStore. ‘Disconnect server’ simulates a network disconnection on the server by intercepting the SocketCommunicator ‘send’ and ‘receive’ methods used by the TestServerCommunicator. ‘Connect server’ undoes ‘Disconnect server’ and sends a server update. A pthread mutex is used to ensure safe access to the ServerCache object from the two application threads.

5.2 Test Cases and Results

Four test cases were covered in the scope of this project. These test cases are far from an exhaustive test suite. However, time constraints resulted in the need to focus testing on what are viewed to be the most common use cases.

Each of the scenarios described below utilizes the same basic application setup. One Server Test Application instance and two Client Test Application instances (described in section 5.1) were run on a single machine. In general, two criteria were looked at when evaluating the success of each test. The primary goal was consistency across all three applications. At the end of each test, printing the contents of each application's cache should display the same values for TestObjects 'A' through 'F'. The second concern, examined for test cases 3 and 4, was accurate conflict detection. The 3rd and 4th test cases intentionally create a conflict using the Client Test Application's disconnected mode. Conflict resolution keeps the last update received by the server. Note that issues such as performance and software footprint were not considered in testing. Only basic functionality was reviewed. A detailed description of each test case follows.

5.2.1 Test Case 1 – Object Set, Connected Mode

This test case evaluates replication while in normal, connected mode.

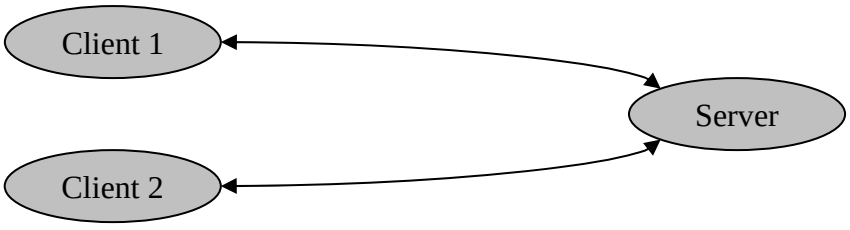

Figure 5.1: Test Case 1 Connectivity

Test Sequence

1. Server Test Application is started.
2. Client Test Applications 1 and 2 are started.
3. The cache is printed from all three applications.
4. Client 1 sets Object A.
5. The cache is printed from all three applications.

Expected Results

1. At step 3, all caches should be equivalent.
2. At step 5, all caches should be equivalent and contain the update to Object A.

Actual Results

1. At step 3, all caches were equivalent. PASS
2. At step 5, all caches were equivalent, including the updated Object A. PASS

5.2.2 Test Case 2 – Object Set, Disconnected Mode

This test case evaluates disconnected operation and non-conflicting reconciliation.

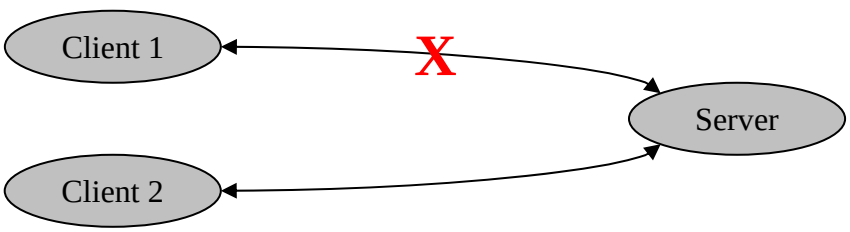

Figure 5.2: Test Case 2 Connectivity

Test Sequence

1. Server Test Application is started.
2. Client Test Applications 1 and 2 are started.
3. The cache is printed from all three applications.
4. Client 1 disconnects.
5. Client 1 sets Object A.
6. The cache is printed from all three applications.
7. Client 1 reconnects.
8. The cache is printed from all three applications

Expected Results

1. At step 3, all caches should be equivalent.
2. At step 6, the Client 1 cache should contain the update to Object A. The Client 2 cache and the Server cache should not contain the update to Object A.
3. At step 8, all caches should be equivalent and contain the update to Object A.

Actual Results

1. At step 3, all caches were equivalent. PASS
2. At step 6, the Client 1 cache contained the update to Object A. The Client 2 cache and the Server cache did not contain the update to Object A. PASS
3. At step 8, all caches were equivalent, including the update to Object A. PASS

5.2.3 Test Case 3 – Conflicting Object Set, Disconnected Mode

This test case evaluates disconnected operation and conflicting reconciliation.

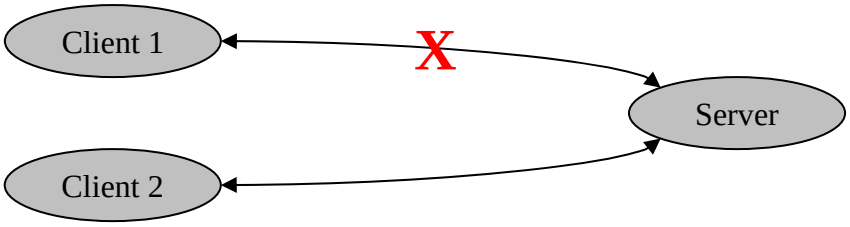

Figure 5.3: Test Case 3 Connectivity

Test Sequence

1. Server Test Application is started.
2. Client Test Applications 1 and 2 are started.
3. The cache is printed from all three applications.
4. Client 1 disconnects.
5. Client 1 sets Object A.
6. Client 2 sets Object A to a different value than Client 1.
7. The cache is printed from all three applications.
8. Client 1 reconnects.
9. The cache is printed from all three applications

Expected Results

1. At step 3, all caches should be equivalent.
2. At step 7, the Client 1 cache should contain its update to Object A. The Client 2 cache and the Server cache should contain the update to Object A from Client 2.
3. At step 9, all caches should be equivalent and contain the update to Object A from Client 1 (the last update to reach the server).

Actual Results

1. At step 3, all caches were equivalent. PASS
2. At step 7, the Client 1 cache contained its update to Object A. The Client 2 cache and the Server cache contained the update to Object A from Client 2. PASS
3. At step 9, all caches were equivalent, including the update to Object A from Client 1. PASS

5.2.4 Test Case 4 – Conflicting Object Set, Dual Disconnected Mode

This test case evaluates dual disconnected operation and conflicting reconciliation.

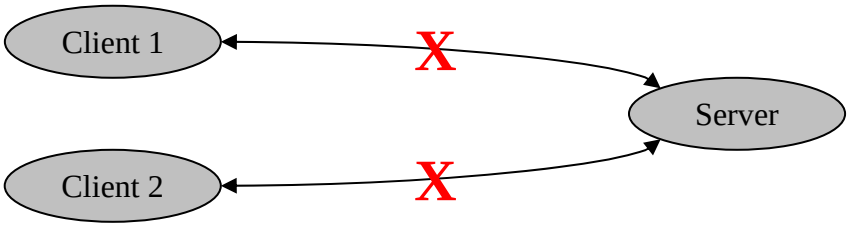

Figure 5.4: Test Case 4 Connectivity

Test Sequence

1. Server Test Application is started.
2. Client Test Applications 1 and 2 are started.
3. The cache is printed from all three applications.
4. Client 1 disconnects.
5. Client 2 disconnects.
6. Client 1 sets Object A.

7. Client 2 sets Object A to a different value than Client 1.
8. The cache is printed from all three applications.
9. Client 2 reconnects.
10. Client 1 reconnects.
11. The cache is printed from all three applications

Expected Results

1. At step 3, all caches should be equivalent.
2. At step 8, the Client 1 cache should contain its update to Object A. The Client 2 cache should contain its update to Object A. The Server cache should contain no update to Object A.
3. At step 11, all caches should be equivalent and contain the update to Object A from Client 1 (the last update to reach the server).

Actual Results

1. At step 3, all caches were equivalent. PASS
2. At step 8, the Client 1 cache contained its update to Object A. The Client 2 cache contained its update to Object A. The Server cache contained no update to Object A. PASS
3. At step 11, all caches were equivalent, including the update to Object A from Client 1. PASS

5.3 Achievements

The successful test cases demonstrate several achievements of this project. The basic functional goals of the software have been met. A simple, effective method

of defining object dependencies is implemented. This dependency specification interface allows groups of interdependent objects to be stored in a server administered cache. The replication algorithm successfully distributes and synchronises the server cache to one or more clients. Additionally, conflict detection has been proven to work in common synchronisation use cases.

The implementation also meets the goal of generalisation. All application objects are referenced as byte sequences. This approach allows the application to use a data structure of any form. Anything from a file to a C++ object is handled the same in this framework. Most points in the framework requiring application specific programming are isolated through the use of abstract interface classes. This modular design allows for future enhancements and extensions to be easily added to the framework.

5.4 Limitations

The primary limitation of the framework created in this project is that it does not provide significantly improved automated reconciliation functionality over existing middleware solutions. In its current state, the framework operates similarly to most other disconnected operation products. Automated conflict detection occurs at the Object Set level rather than at the Object level. This limitation restricts the level of sophistication supported in object dependency specifications. This implementation focused more on simplicity and basic functionality, but given the generic design approach of the framework, it should be easily extensible. More sophisticated dependency specification support and finer grained conflict detection should, therefore, be possible.

6. Conclusion

6.1 Summary

With the advent of mobile computing, disconnected operation has become a fact of life for mobile clients. While many applications can remain productive when disconnected from the network, such operation does introduce many challenges. Caching local copies of remote data can allow mobile clients to operate off the network, but issues of data accuracy and consistency arise. Furthermore, multiple clients operating on the same cached data can result in conflicts. Detecting and reconciling such conflicts can be a difficult and costly process.

This project attempted to develop a generic means of defining data dependencies and implement a framework for supporting generic disconnected operation functions. Automation of sophisticated conflict detection and resolution was a primary goal of this work. In its current state, the framework demonstrated basic replication and reconciliation functionality. While the project fell short of drastically improving on reconciliation capabilities of previous solutions, the generic design of the resulting framework has potential to be enhanced to support superior reconciliation mechanisms.

6.2 Future Work

6.2.1 Optimizations

Performance was not targeted as a major implementation goal of this project. Consequently, there is room for significant improvement in the framework's efficiency. Reducing the frequency of dynamic memory allocation and memory

copies would benefit the performance of the caching framework. Likewise, optimizing the amount of data transmitted between clients and servers would also improve performance. In cases when clients are collocated with the server, applications could possibly short circuit the normal caching process and operate directly on the server cache.

6.2.2 Server Cache Initialization

Currently, Server Cache initialization is straightforward, but a bit tedious. Object dependency specifications and initial object data values are also hard coded. This process could be improved by implementing a configuration framework for the cache. A markup language such as Extensible Markup Language (XML) could be used to specify object dependencies and values rather than hard coding the information. Such a system could potentially redefine object dependencies or reinitialize a cache simply by loading a new XML document. The increased flexibility required to implement a configuration framework could also involve implementing an API usable by applications to dynamically build and change object dependencies at runtime.

6.2.3 Selective Caching

The current design of this framework operates on an entire cache of objects. Replication involves copying the entire Server Cache to a client, regardless of what the actual client's needs are. Considering the limited storage capacity and bandwidth available on many mobile clients, selectively caching data would be a definite advantage. A client could specify statically which objects it needs, or potentially the server could learn which objects are needed based on a client's usage behaviour.

6.2.4 Granularity of Conflict Detection/Resolution

Supporting conflict detection and resolution at the object level rather than the Object Set would allow more sophisticated degrees of automation to be utilized by applications. This could result in simplified application logic by pushing more of the complexities of dealing with conflict detection and resolution into the framework.

6.2.5 Encapsulation of Id Types

In the current implementation of this framework, identifiers are typed as standard template library strings. Specifying a fixed id type requires applications to convert native id types to the fixed type. This conversion could be avoided if id types were encapsulated such that applications could insert their own type. An id interface class could be used, or the existing classes could be implemented as templates, with all id references parameterized.

6.2.6 Object Base Class

Similar to the encapsulation of id types, defining a class interface for application defined objects would have several benefits. Currently, applications are forced to convert all data to serialized byte sequences. Such a format is rarely the form used by applications, especially in an object-oriented environment. Defining an abstract object class for applications to derive from could avoid the use of serialization (except when transmitting objects across the network) and rely on polymorphism instead. Performance benefits and simplified programming API would result.

6.2.7 Customizable Memory Management

Applications often have very particular memory management requirements, especially on mobile clients where memory resources might be scarce. The use of the standard template library map class might not be sufficient in all cases. For example, memory pools could be used to avoid dynamic memory allocations, or a database could be used to persist cache data. A memory management interface could be defined to allow applications to plug in a management implementation of their choosing. Such a framework could even support multiple memory management implementations and choose the best one based on a client's resources, for example.

6.2.8 Configurable Automated Update Policies

The current framework design provides an API for sending updates between clients and servers, but leaves it up to the applications to initiate the process. Potentially, the update process could be automated, relieving the applications of any involvement. Different policies could be implemented and configured by applications. For example, a client update could be sent every time an object is accessed for maximum consistency behaviour. Or perhaps updates would be sent only after certain high priority objects are accessed to save on bandwidth usage. Or alternatively, updates could be sent at fixed intervals to ensure a throttling of updates regardless of object access frequency. Similar policies would be equally applicable on the server side.

7. References

[1] Anthony D. Joseph, Alan F. deLespinasse, Joshua A. Tauber, David K. Gifford, and M. Frans Kaashoek. *Rover: A Toolkit for Mobile Information Access.* M.I.T. Laboratory for Computer Science, December 1995.

[2] Anthony D. Joseph, Joshua A. Tauber, and M. Frans Kaashoek. *Mobile Computing with the Rover Toolkit.* IEEE Transactions on Computers: Special issue on Mobile Computing, 46(3). March 1997.

[3] M. Satyanarayanan. *Fundamental Challenges in Mobile Computing.* Fifteenth ACM Symposium on Principles of Distributed Computing, May 1996.

[4] P. J. Braam. *The Coda Distributed File System.* Linux Journal, #50, June 1998, pages 46-51.

[5] M. Satyanarayanan. *Mobile Information Access.* IEEE Personal Communications, Vol. 3, No. 1, February 1996.

[6] M. Satyanarayanan. *Coda: A Highly Available File System for a Distributed Workstation Environment.* Proceedings of the Second IEEE Workshop on Workstation Operating Systems, September 1989, Pacific Grove, CA.

[7] Y.W. Lee, K.S. Leung, M. Satyanarayanan. *Operation-based Update Propagation in a Mobile File System.* Proceedings of the USENIX Annual Technical Conference, June 1999, Monterey, CA.

[8] J. J. Kistler, M. Satyanarayanan. *Disconnected Operation in the Coda File System.* ACM Transactions on Computer Systems, February 1992, Vol. 10, No. 1, pages 3-25.

[9] P. Kumar, M. Satyanarayanan. *Flexible and Safe Resolution of File Conflicts.* Proceedings of the USENIX Winter 1995 Technical Conference, January 1995, New Orleans, LA.

[10] P. Kumar. *Coping with Conflicts in an Optimistically-Replicated File System.* Proceedings of the IEEE Workshop on Management of Replicated Data, November 1990, Houston, TX.

[11] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch. *The Bayou Architecture: Support for Data Sharing among Mobile Users.* Proceedings of the Workshop on Mobile Computing Systems and Applications, Santa Cruz, California, December 1994, pages 2-7.

[12] D. B. Terry, K. Petersen, M. J. Spreitzer, and M. M. Theimer. *The Case for Non-transparent Replication: Examples from Bayou.* IEEE Data Engineering, December 1998, pages 12-20.

[13] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. *Flexible Update Propagation for Weakly Consistent Replication.* Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-16), Saint Malo, France, October 5-8, 1997, pages 288-301.

[14] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. Hauser. *Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System.* Proceedings 15th Symposium on Operating Systems Principles (SOSP-15) , Cooper Mountain, Colorado, December 1995, pages 172-183.

[15] Java Remote Method Invocation. 2005. http://java.sun.com/products/jdk/rmi/

[16] Mads Haahr, Raymond Cunningham and Vinny Cahill. *Supporting CORBA Applications in a Mobile Environment.* MobiCom '99: 5th International Conference on Mobile Computing and Networking. Seattle, August 1999.

[17] Michael Philippsen and Bernhard Haumacher. *More Efficient Object Serialization.* University of Karlsruhe, 1999.

[18] Sidney Chang, Dorothy Curtis. *An Approach to Disconnected Operation in an Object-Oriented Database.* 3rd International Conference on Mobile Data Management, January 2002.

[19] Marc E. Fiuczynski, David Grove. *A Programming Methodology for Disconnected Operation.* Department of Computer Science and Engineering, University of Washington, FR-35, March 1994, Seattle, WA.

8. Appendix A: Class Summary

This appendix contains descriptions of all classes developed for this project, listed alphabetically and grouped by library.

8.1 libcommon

libcommon contains classes used by both client and server applications.

8.1.1 Cache

The Cache class encapsulates a set of ObjectSet objects. A map of each CachedObj id to its owning ObjectSet is also kept to optimize object look ups.

8.1.2 CachedObj

The CachedObj class encapsulates a serialized application object in a form acceptable to the ObjectSet and Cache classes. Attributes include the serialized object data, the size of the object data, and a unique id naming the contained object.

8.1.3 DependencyType

The DependencyType class encapsulates an enumeration of object set dependency types. Values include READ, WRITE, and READ_WRITE.

8.1.4 ObjectAccessConflict

The ObjectAccessConflict class is a derivative of std::exception. This exception is thrown when object set access conflicts are detected in the system.

8.1.5 ObjectNotFound

The ObjectNotFound class is a derivative of std::exception. This exception is thrown when non-existent CachedObj objects are accessed.

8.1.6 ObjectSet

The ObjectSet class encapsulates a group of CachedObj objects that are dependent on each other. A DependencyType attribute and access metadata are also included.

8.2 libclient

libclient contains classes used only by client applications.

8.2.1 ClientCache

The ClientCache class derives from the Cache class, adding additional methods to integrate with the client application and communicate with the ServerCache. The ClientCache class contains the primary public API used by the client application. The ClientCache API includes methods to get and set objects as well as send updates to the ServerCache and process updates from the ServerCache.

8.2.2 ClientCacheAdministrator

The ClientCacheAdministrator class is an abstract class used by the ClientCache class to execute application specific operations. Applications must implement the ClientCacheAdministrator interface in order to instantiate a ClientCache. The interface includes a method to notify the client application of a server update and a method to generate version ids for client updates.

8.2.3 ClientCommunicator

The ClientCommunicator class is an abstract class used by the ClientCache class to communicate messages to the ServerCache. Applications must implement the ClientCommunicator interface in order to instantiate a ClientCache. The interface includes a method to send a message to the ServerCache.

8.3 libserver

libserver contains classes used only by server applications.

8.3.1 ServerCache

The ServerCache class derives from the Cache class, adding additional methods to integrate with the server application and communicate with the ClientCache. The ServerCache class contains the primary public API used by the server application. The ServerCache API includes methods to send updates to the ClientCache and process updates from the ClientCache.

8.3.2 ServerCacheAdministrator

The ServerCacheAdministrator class is an abstract class used by the ServerCache class to execute application specific operations. Applications must implement the ServerCacheAdministrator interface in order to instantiate a ServerCache. The interface includes methods to notify the server application of a client update, initialize the server cache, generate object set version ids, and resolve update conflicts.

8.3.3 ServerCommunicator

The ServerCommunicator class is an abstract class used by the ServerCache class to communicate messages to the ClientCache. Applications must implement the

ServerCommunicator interface in order to instantiate a ServerCache. The interface includes a method to send a message to the ClientCache.

9. Appendix B: Example Usage

This appendix demonstrates usage of the classes developed in this project. Examples are based on the test applications described in section 5.

9.1 Client Application

```
//--- INITIALIZATION ---
// create implementation of ClientCommunicator interface
TestCommunicator comm;

// create implementation of ClientCacheAdministrator interface
TestClientCacheAdmin admin;

// create client cache
DISCOP::ClientCache cache("Cache 1", comm, admin);

// initialize client cache
cache.requestServerUpdate();


//--- OBJECT GET ---
// create object id
std::string objectId = "MyId";

try
{
   // get cached object from cache
   DISCOP::CachedObj cachedObj = cache.getObject(objectId);

   // create application object
   TestObject testObj;

   // deserialize cached object data
   testObj.deserialize(cachedObj.getObject());
}
catch(ObjectNotFound onf)
{
   std::cerr << onf.what() << std::endl;
}


//--- OBJECT SET ---
// using previous application and cached object
// serialize application object
char * serializedObj = testObj.serialize();

try
{
   // set object data in cached object
   cachedObj.setObject(serializedObj, testObj.getSerializeSize());
```

```
   // set object from cache
   cache.setObject(cachedObj);
}
catch (ObjectAccessConflict oac)
{
   std::cerr << oac.what << std::endl;
}

// delete serialized object
delete [] serializedObj;
serializedObj = 0;


//--- CLIENT UPDATE ---
// send client update
cache.sendClientUpdate();


//--- PROCESS SERVER UPDATE ---
// assume server update message is contained in ‘char serverUpdate[]’
cache.processServerUpdate(serverUpdate);
```

9.2 Server Application

```
//--- INITIALIZATION ---
// create implementation of ServerCommunicator interface
TestCommunicator comm;

// create implementation of ServerCacheAdministrator interface
TestServerCacheAdmin admin;

// create server cache
DISCOP::ServerCache cache(comm, admin);

// initialize server cache
cache.initialize();


//--- SERVER UPDATE ---
// send server update
cache.sendServerUpdate();


//--- PROCESS CLIENT UPDATE ---
// assume client update message is contained in ‘char clientUpdate[]’
cache.processClientUpdate(clientUpdate);
```

10. Appendix C: Installation Guide

This section gives a brief description of how to install the software libraries outlined in this document. All source code and makefiles are contained in the file 'discop.zip'. Note that this installation procedure has only been tested on the RedHat Cygwin platform.

1. Extract the file discop.zip.
2. Set the environment variable MAK_ROOT to the 'discop' directory.
3. Optionally set the environment variable MAK_DEBUG to YES to compile the libraries with debug symbols.
4. From the $(MAK_ROOT)/src directory, type 'make'. This should result in the following directories:
    - $(MAK_ROOT)/inc – contains library header files
    - $(MAK_ROOT)/lib – contains libcommon.a, libclient.a, libserver.a

The 'common' library must be linked with client and server applications, the 'client' library must be linked with client applications, and the 'server' library must be linked with server applications. Header files are stored in subdirectories named for their corresponding library (i.e. $(MAK_ROOT)/inc/common, $(MAK_ROOT)/inc/client, $(MAK_ROOT)/inc/server).