A Framework for Creating Distributed GUI Applications

Master's Project Report
Derek Snyder
May 15, 2006

Advisor: John Jannotti

Introduction

Creating distributed graphical user interface (GUI) applications is difficult because such applications must respond quickly to user actions in the presence of network communication delays. To further complicate matters, different users of an application may perform actions that conflict with each other; the application must be able to resolve these conflicts without violating users' expectations of the application's behavior.

One solution to this problem is optimistic concurrency. When the user performs an action, the local instance of the application updates the user interface based on its current understanding of the global state. If that action is later found to conflict with an action performed by another user, one of the actions is undone and the corresponding user interface is updated appropriately (possibly by displaying a notification to the user).

To the best of our knowledge, this technique has been implemented so far only on an ad hoc basis. The developers of each application must define a custom protocol and data structures in order to implement optimistic concurrency. In addition to being time consuming, this approach can easily lead to program errors because of the complexity of the concurrent updates.

The goal of this project was to create a library of shared data structures that can be used to build distributed applications. This library is called the Shared Object Framework. The framework is generic enough that it can be used in a wide variety of applications, without assuming any particular application semantics. On the other hand, it is powerful enough that the application developer only needs to be minimally aware of the existence of conflicts.

The development platform supported by the framework is web applications that use the asynchronous HTTP request object. There are several advantages to using the web as a development platform. First, the functionality supported by modern web browsers rivals that of many GUI libraries. Second, the use of HTTP eliminates the need to implement a low-level communication protocol. Third, web browser software is already widely deployed, and many browser programs are freely available. Finally, standards compliance among browsers is improving to the point where it will soon be possible to write sophisticated web applications that run on almost every modern desktop operating system.

Design Overview

The framework consists of both server-side and client-side code. The server code is written in the Python programming language; the client code is written in JavaScript. The framework exposes an application programming interface (API) on both the server and the client. To implement the application logic, the developer writes additional Python or JavaScript code that calls the functions in the API.

To simplify the framework implementation, all code is single-threaded. There is one thread of execution running on the server, plus one thread on each client. The server serializes overlapping requests from clients.

The framework currently supports two types of shared objects: arrays and maps (i.e., dictionaries). An application may create an arbitrary number of each type of object. The application manipulates shared objects by using transactions. Each transaction contains one or more calls to methods of the shared objects.

The server stores the authoritative versions of the shared objects. When a transaction is committed on the server side, the framework immediately updates the affected objects. When a transaction is committed on the client side, the framework sends a description of the transaction to the server-side framework code. If the transaction is acceptable, the server-side code updates the affected objects. On the other hand, if the transaction conflicts with an earlier committed transaction, the server-side code rolls back all effects of the transaction and notifies the client that the transaction has been aborted.

Local changes to shared objects on the client take effect immediately, but are not necessarily permanent. The client maintains two set of objects: an old set and a new set. When the client-side application code initiates a local object change, the client applies the change to the new set and sends the change to the server. Every time the client receives an update from the server, the client performs the following steps in order:

1. Discard the new set.
2. Apply the changes from the server to the old set.
3. Copy the old set to the new set.
4. Apply all uncommitted changes to the new set.

When the server first receives a request from a client, it generates a session ID and includes it in the response to the client. The session ID is a random 20-character string. The client includes the session ID in all subsequent requests so that the server knows that the requests originate from the same client. The server also includes an ID prefix in its initial response to the client. The client uses this prefix to form globally unique IDs for objects and object versions. The server also has its own unique prefix.

When a shared object is created, it is given a unique object ID formed with either the server prefix or the client prefix, depending on where the object was created. After each change to the

object, the object is stamped with a version ID, again formed using the server or client prefix. The version ID is globally unique for that object, but not necessarily unique among all objects.

The server detects conflicts between transactions by looking at the return values from method calls. When the client executes the methods in a transaction, it stores the return value from each method call and forwards them to the server. The server then executes the same method calls and compares the return values it obtains with the expected return values from the client. If the return values do not match, the server aborts the transaction because the effects of the transaction are potentially not what the client expected. (This is a conservative test because it is possible that the client didn't examine the values returned by the method calls.)

When the client sends a transaction request to the server, the server may not receive the request, due to a network timeout or other error. Also, transaction requests are not guaranteed to arrive at the server in order, because each one is a separate HTTP request. Because of this, when the client sends a transaction to the server, it continues to include the transaction in every subsequent request until the server acknowledges the transaction. Each transaction is labeled with a monotonically increasing transaction number that is unique for that client. The server keeps track of the highest transaction number that it has seen from each client. When the server receives a transaction number that is less than or equal to the highest transaction number, it ignores the transaction.

Updates from the server to clients are pull-based. The purpose of this is to minimize the state that needs to be stored on the server. Each client periodically sends a request to the server for all objects that the client is interested in. The request contains the version IDs of the objects that are currently stored on the client. When the server receives the request, it compares each of the client's version IDs to the version ID of the corresponding object on the server. If the version IDs match, the server sends no updates for that object. If the client's version is out of date, the server sends the sequence of method calls that are necessary to update the client's version to the current version on the server. If the server does not recognize the version ID from the client, it sends the current state of the object to the client.

Architecture

The following diagram illustrates the major components of the Shared Object Framework:

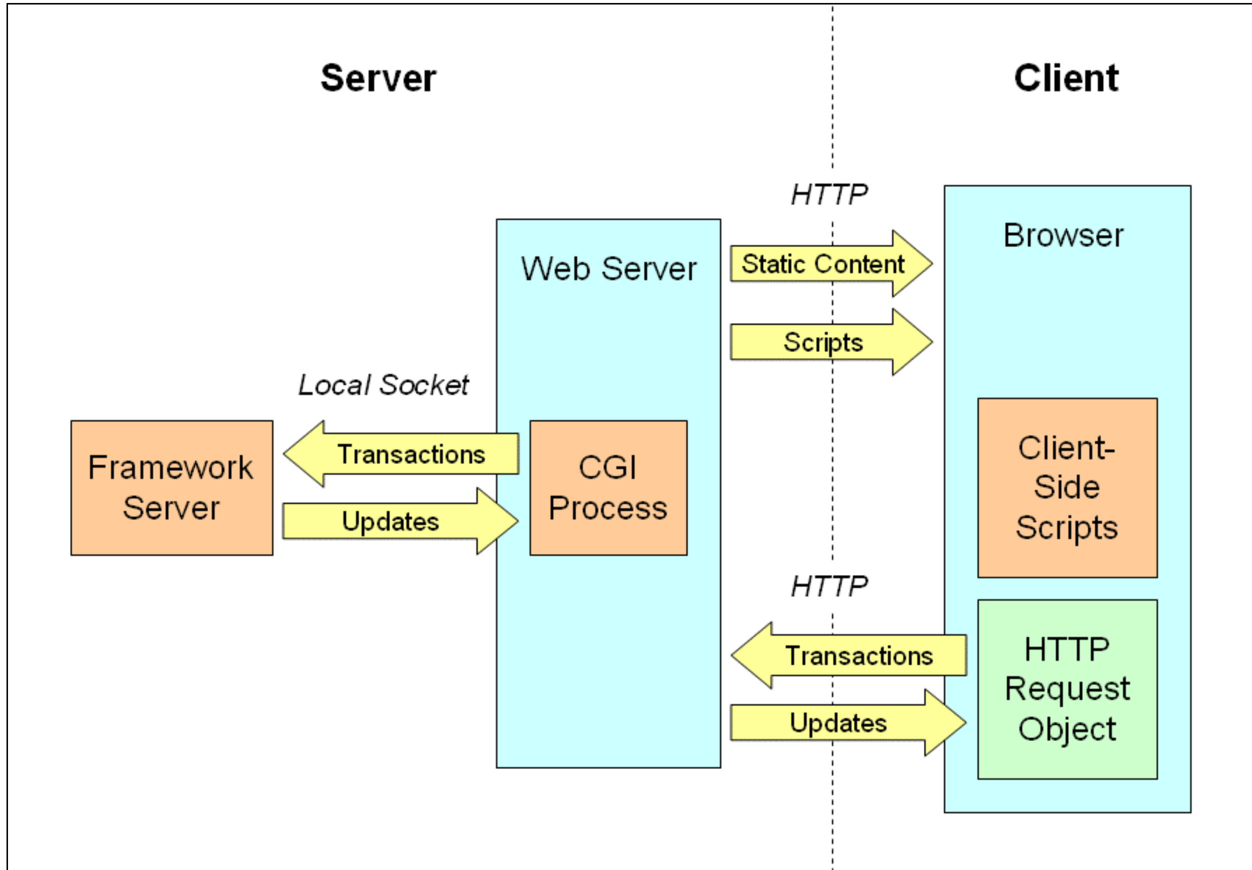

Figure 1: Components of the Shared Object Framework

Framework Server

The framework server maintains all server state. It receives transaction requests and object queries. It responds with transaction results and object updates.

The server is written in Python. The code resides in the `service` directory. The executable file is named `server`.

The server communicates with the CGI process via a local (UNIX type) socket. The socket name is specified in the configuration file `config.py` in the `service` directory.

Web Server

The web server is any HTTP server that supports the Common Gateway Interface (CGI). For this project I used Apache as the web server.

CGI Process

The CGI process receives requests from the web server and forwards them to the framework server. It uses a very simple protocol that consists of affixing the request length to the beginning of the request, followed by a line feed character.

The framework server and the CGI process are separate so that the web server can be restarted without causing the framework server to lose its state.

The CGI process is written in Python. The code resides in the `cgi` directory. The script is named `framework.py`.

The name of the socket that the CGI process uses to communicate with the framework server is specified in the configuration file `config.py` in the `cgi` directory. The socket name must be the same one specified in the configuration file for the framework server.

Browser

The browser must support XHTML 1.0, CSS 2.1, JavaScript 1.5, and the asynchronous HTTP request object. For this project I used Firefox 1.5 as the web browser.

Client-Side Scripts

The client-side scripts implement the client framework and maintain all client state information. The scripts are written in JavaScript. The application's HTML pages must include the scripts so that the browser will load them from the web server.

The client-side scripts send requests the web server using the asynchronous HTTP request object that is built into the browser. The requests are forwarded to the CGI process and then to the framework server.

The client-side scripts are located in the `www/scripts` directory.

Protocol

The framework protocol is an application of JSON, a simple data exchange format that is a subset of JavaScript. (For more information, see the JSON website.)

Within the protocol, numbers, strings, Boolean values, and null values are represented as their literal JSON values. References to shared objects are coded as JSON objects with a single member named “object_id”. For example:

```
{ "object_id": "25_7" }
```

Request

The request is formatted as a single JSON object with the following members:

- **session_id** – The client’s session ID, received as part of a previous response from the server. If this is the client’s first request, the value of this member is the empty string.
- **query_object_map** – An object that maps object IDs to version IDs. Each object ID specifies an object for which the client is interested in receiving updates. The corresponding version ID indicates the version of the object that the client currently has.
- **transaction_list** – A list of transaction requests from the client. Each transaction request in the list is an object with the following members:
    - **transaction_num** – A number that uniquely identifies the transaction request for this client. The transaction number should increase with each new transaction request.
    - **operation_list** – A list of operation that comprise the transaction. Each operation in the list is an object with the following members:
        - **object_id** – A string that uniquely identifies an object. If the object ID is “Array” or “Map”, then the operation is a constructor call. Otherwise, the object ID specifies the existing object whose method should be invoked.
        - **method_name** – The name of the method that should be invoked. If the operation is a constructor call, then this member will be absent.
        - **param_list –** A list of parameters to pass to the method.
        - **return_value** – The expected return value from the method call.
        - **new_version** – The new version ID to apply to the object after its method has been invoked.
        - **new_object_id** – The new object ID to assign to the return value of a method call or to the object created by a constructor call.
        - **new_object_version** – The version ID to apply to the return value of a method call or to the object created by a constructor call.

Here is a sample of a request in the protocol:

```
{
    "session_id": "u3UzzjPQWFShzxiuPav7",
    "query_object_map": {
        "_global_list": "9_43",
        "3_17": "5_58",
        "8_59": "_89",
        "7_26": "9_43",
        "4_49": "3_29"
    },
    "transaction_list": [
        {
            "transaction_num": 27,
            "operation_list": [
                {
                    "object_id": "_global_list",
                    "method_name": "get_length",
                    "param_list": [],
                    "return_value": 5
                },
                {
                    "object_id": "_global_list",
                    "method_name": "splice",
                    "param_list": [ 5, 1, { "object_id": "7_26" } ],
                    "new_version": "3_36"
                }
            ]
        },
        {
            "transaction_num": 28,
            "operation_list": [
                {
                    "object_id": "3_17",
                    "method_name": "update",
                    "param_list": [ { "object_id": "4_49" } ],
                    "new_version": "3_36"
                }
            ]
        },
        {
            "transaction_num": 29
            "operation_list": [
                {
                    "object_id": "8_59",
                    "method_name": "set_item",
                    "param_list": [ "color", "green" ],
                    "new_version": "3_52"
                }
            ]
        }
    ]
}
```

Response

The response is formatted as a single JSON object with the following members:

- **session_id** – The session ID that the client should include in all future requests to the server.
- **status** – “success” if the request was valid; otherwise, “error”. A response status of “success” merely means that the server was able to process the request. It does not mean that any of the transactions within the request were successful.
- **message** – If the response status is “error”, this is a description of the error. If the response status is “success”, this member is absent.
- **prefix** – The prefix that the client should append to all object IDs and version IDs that it generates.
- **transaction_result_list** – A list of results for the transactions in the request. Each transaction result in the list is an object with the following members:
  - **transaction_num** – The transaction number.
  - **status** – “success” if the transaction was committed successfully, “aborted” if the transaction failed because of a conflict, “ignored” if the server already processed the transaction in a previous request, or “error” if the syntax of the transaction was invalid.
  - **message** – If the transaction status is “error”, this is a description of the error. Otherwise, this member is absent.
- **operation_list** – A list of operations to be applied to objects on the client. Each operation in the list is an object with the following members:
  - **object_id** – A string that uniquely identifies an object. If the object ID is “Array” or “Map”, then the operation is a constructor call. Otherwise, the object ID specifies the existing object whose method should be invoked.
  - **method_name** – The name of the method that should be invoked. If the operation is a constructor call, then this member will be absent.
  - **param_list –** A list of parameters to pass to the method.
  - **new_version** – The new version ID to apply to the object after its method has been invoked.
  - **new_object_id** – The new object ID to assign to the return value of a method call or to the object created by a constructor call.
  - **new_object_version** – The version ID to apply to the return value of a method call or to the object created by a constructor call.

Here is a sample of a response in the protocol:


```
{
    "session_id": "u3UzzjPQWFShzxiuPav7",
    "status": "success",
    "prefix": "3_",
    "transaction_result_list": [
        {
            "transaction_num": 27
            "status": "ignored"
        },
        {
            "transaction_num": 28
            "status": "aborted"
        },
        {
            "transaction_num": 29
            "status": "success"
        }
    ],
    "operation_list": [
        {
            "object_id": "_global_list",
            "method_name": "splice",
            "param_list": [ 5, 1, { "object_id": "7_26" } ],
            "new_version": "3_36"
        },
        {
            "object_id": "3_17",
            "method_name": "update",
            "param_list": [ { "object_id": "12_3" } ],
            "new_version": "9_47"
        },
            {
            "object_id": "8_59",
            "method_name": "set_item",
            "param_list": [ { "color", "green" } ],
            "new_version": "3_52"
            }
    ]
}
```

Implementation

The following diagram illustrates the relationships between the objects used to implement the Shared Object Framework:

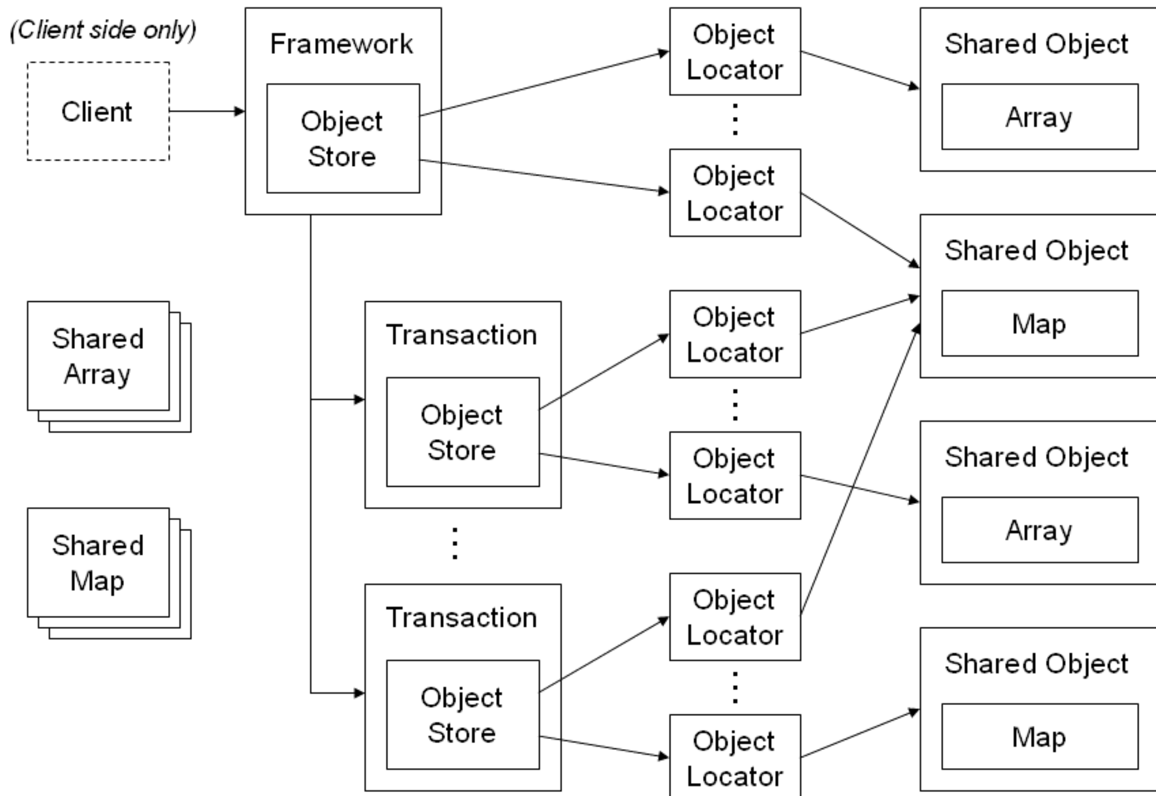

Figure 2: Object relationships

The object classes are described below. Except for Client, each class is implemented separately, on both the server side and the client side.

Array and Map

These classes store the actual values of the shared objects. The interfaces for these classes are similar to the interfaces for the native Python `list` and `dict` classes, respectively. The main difference is that these classes add methods to support functionality that is implemented in the native Python classes using operators (e.g., subscripting).

Within Array and Map objects, references to other (non-primitive) objects are stored as references to Shared_Object objects.

Shared_Object

This class stores a shared object's ID, version information, and change history. It also contains a reference to an Array or Map object.

Object_Locator

This class stores a reference to a Shared_Object object, along with a flag indicating whether the object has been modified. The purpose of this class is to support the copy-on-write functionality of the Object_Store class.

Object_Store

This class maps object IDs to object locators. Initially there is only one instance of Object_Store, referenced in the Framework object. When a new transaction is begun, a new Object_Store instance is created that contains copies of all the Object_Locator objects from the previous instance. (For efficiency, the Shared_Object objects are not copied until they are modified.)

Transaction

This class represents a (possibly nested) transaction in the framework. The class stores a reference to an object store and a list of uncommitted operations.

When a transaction is begun, a new Transaction object is created by copying the object store from the enclosing transaction. When a transaction is committed, the object store is merged into the object store of the enclosing transaction and the operation list is appended to enclosing transaction's operation list. When a transaction is rolled back, the corresponding Transaction object is merely discarded.

Framework

This singleton class represents the Shared Object Framework itself. It contains a reference to an Object_Store object that contains the effects of all committed transaction. It also maintains a stack of Transaction objects.

Server-side application code calls the methods of this class directly to create shared objects and to execute transactions. Client-side application code uses this class indirectly through the Client class.

Shared_Array and Shared_Map

These classes act as proxies for the shared objects in the system. They exist as a convenience for the application programmer. The classes support the same methods as the Array and Map classes, respectively. When a method is called on a proxy object, the proxy automatically forwards the method call to the framework.

Client (client side only)

The Client class wraps the Framework class on the client side and provides additional client-specific functionality. It exploits the Framework class's built-in transaction capability to maintain two sets of object changes: committed changes from the server, and uncommitted local changes. The Client class also handles communication with the server, and notifies the client-side application code when an object changes or when an error occurs.

Application Programming Interface

This section describes the application programming interface (API) that the framework exposes to the application code on both the server side and the client side.

The method specifications below use Python syntax. The JavaScript syntax has the following differences:

- Constructor invocations must be preceded by the keyword `new`.
- The JavaScript value `null` should be used in place of the Python value `None`.
- The JavaScript values `true` and `false` should be used in place of the Python values `True` and `False`, respectively.

Framework

Method Call	**Description**
`Framework(id_prefix)`	Instantiate the Framework class. Only one object of this type should be created. `id_prefix` is the prefix to use to form object IDs and version IDs.
`begin_transaction()`	Start a new transaction. Transactions may be nested.
`commit_transaction (change_object_map = None)`	Commit the current transaction. The transaction will not be permanently applied to the object repository until the transaction depth reaches zero.

	If the change_object_map parameter is supplied, it will be populated with the objects that have changed. The keys in the map will be object IDs and the values will be proxy objects.
rollback_transaction()	Abort the current transaction and undo its changes.
get_transaction_depth()	Get the current transaction depth. The transaction depth starts at zero and is incremented every time a transaction is begun. The transaction depth is decremented every time a transaction is committed or rolled back.
create_object(object_value, object_id = None)	Create a shared object. object_value is either a native array, a native map, an Array object, or a Map object. If object_id is not specified, a new object ID will be generated. Note that a shared object can also be created by instantiating the Shared_Array or Shared_Map class.

Client

Method Call	Description
Client(web_service_url, refresh_interval_ms, object_change_handler, error_handler)	Instantiate the Client class. Only one object of this type should be created. web_service_url is the URL to use to contact the server. refresh_interval_ms is the delay, in milliseconds, between requests to the server when the client has no local changes to report. object_change_handler is the function to call when one or more shared objects have been modified. When the function is called, it will receive one parameter: a map of object IDs to proxy objects. error_handler is the function to call when an error occurs. When the function is called, it will receive one parameter: an error message string.
begin_transaction()	Start a new transaction. Transactions may be nested.

`commit_transaction()`	Commit the current transaction. The transaction will not be permanently applied to the object repository until the transaction depth reaches zero.
`rollback_transaction()`	Abort the current transaction and undo its changes.
`create_object(object_value, object_id = None)`	Create a shared object. `object_value` is either a native array, a native map, an Array object, or a Map object. If `object_id` is not specified, a new object ID will be generated. Note that a shared object can also be created by instantiating the Shared_Array or Shared_Map class.
`load_object(object_id)`	Make the specified shared object available on the client.

Shared_Array

Method Call	**Description**
`Shared_Array(array_value = [])`	Create an array-type shared object from an ordinary array object.
`has_item(value)`	Return True if the value is present in the array; otherwise, False.
`concat(other)`	Append the elements in another array to this array.
`get_item(index)`	Return the element at the given index.
`get_slice(start, end)`	Return the subarray starting at the index `start` and ending just before the index `end`.
`get_length()`	Return the number of elements in the array.
`set_item(index, value)`	Set the element at the specified index to the given value.
`delete_item(index)`	Remove the element at the specified index, shifting all subsequent elements to the left by one.
`set_slice(start, end, sequence)`	Remove the subarray at [`start`, `end`) and replace it with the array `sequence`.
`delete_slice(start, end)`	Remove the subarray at [`start`, `end`).
`append(value)`	Append the given value to the end of the array.
`count(value)`	Return the number of occurrences of the given value in the array.
`index(value)`	Return the index of the first occurrences of the given value in the array.
`insert(index, value)`	Insert the given value at the specified index in the array, shifting all subsequent elements to the right by one.
`remove(value)`	Remove the first occurrence of the specified value in the array.

`reverse()`	Reverse the order of elements in the array.

Shared_Map

Method Call	Description
`Shared_Map(map_value = {})`	Create a map-type shared object from an ordinary map object.
`get_length()`	Return the number of keys in the map.
`get_item(key)`	Return the value for the given key.
`set_item(key, value)`	Associates the given value with the specified key.
`delete_item(key)`	Remove the specified key from the map.
`clear()`	Remove all keys from the map.
`copy()`	Return a shallow copy of the map.
`has_key(key)`	Return True if the key exists in the map; otherwise, False.
`items()`	Return an array of key/value pairs in the map. Each element in the returned array is a two-element array that contains a key and a value.
`keys()`	Return an array that contains the keys in the map.
`update(other)`	Add the entries in another map to this map. If a key exists in both maps, the value in the other map will overwrite the value in this map.
`values()`	Return an array that contains the values in the map.
`get(key, default_value)`	Return the value associated with the specified key. If the key doesn't exist in the map, return the specified default value instead.
`setdefault(key, default_value)`	Return the value associated with the specified key. If the key doesn't exist in the map, associate the key with the specified default value and also return the default value.
`pop(key, default_value)`	If the key exists in the map, remove the key and return its associated value. If the key doesn't exist in the map, return the specified default value.
`popitem()`	Remove an arbitrary key/value pair from the map and return it as a two-element array.

Known Bugs

Array and map objects returned by methods are not correctly wrapped inside proxy objects.

Examining the value of a shared object on the client side requires a method call, which triggers a request to the server. There are two possible solutions to this problem. One solution is to designate certain methods as read-only. If a top-level transaction contains only read-only

methods, it should not be sent to the server. The other solution is to ignore method calls that do not occur inside an explicit transaction. This solution would require the application programmer to use explicit transactions for all object changes.

Garbage collection has not been implemented. Once a shared object is created in the framework, it is never deleted.

The server does not purge old object histories. If an object is continually changed, its memory usage will grow without bound.

Circular object references can cause an infinite loop in the framework code.

Future Work

The framework should support other shared object types besides arrays and maps. It would be nice if there were an automated process for converting ordinary classes (written in a language like Python) to shared object types.

The framework already contains many aspects of an interpreter. It has a symbol table, an object store, and an execution engine. Modifying the framework to incorporate a complete interpreter would provide significant utility. It would then be possible for an application programmer to write a complete GUI application in a language like Python, and then convert it to a distributed application with little effort. Ideally, the source language would allow programmers to reference document object model (DOM) objects directly so that, for example, a single function could both modify a back-end database and update the user interface.