Collabrary Shared Dictionary v1.0.17

Programming Paradigm and Wire Protocol

Michael Boyle

GroupLab, Dept. of Computer Science, U. of Calgary

January 30, 2003

Table of Contents

1 Programming for the Collabrary Shared Dictionary 3
1.1 Introduction 3
1.2 Dictionary Programming 3
1.3 Shared Dictionary Keys 4
1.4 Canonical Form for Shared Dictionary Keys 4
1.5 Patterns 5
1.6 Storing things in the Shared Dictionary 5
1.7 GUIDs 6
1.8 Shared Dictionary Instance GUIDs 6
1.9 Metadata 6
1.9.1 .transient keys 6
1.9.2 .timestamp keys 7
1.9.3 .lifetime keys 7
2 Collabrary Shared Dictionary 7
2.1 A brief look at the protocol stack 7
3 Serialization Process 8
3.1 Serializing primitive types 9
3.2 Serializing strings 10
3.3 Serializing objects 10
3.3.1 SerialBuffer Class 11
3.3.2 Serializing Collabrary.Struct objects 11
3.3.3 Serializing Collabrary.SharedDictionary objects 12
4 Shared Dictionary Protocol Layer Operation 13
4.1 URL Syntax 13
4.2 Connection Establishment 13
4.3 Messages Sent While Connected 14
4.3.1 HELLO Message Type = 1 14
4.3.2 WELCOME Message Type = 2 15
4.3.3 GOODBYE Message Type = 3 15
4.3.4 SIGNAL Message Type = 4 15
4.3.5 PUT Message Type = 5 16

5 Appendix A. SharedDictionary Canonicalize C++ code 17

6 Appendix B. SharedDictionary Match C++ code 19

7 Appendix C. Algorithm for hashing string URL port specifiers to a numeric TCP/IP port identifier. 22

8 Appendix D. Real-world example of the Shared Dictionary client-server communication: TCP packets sent between a Notification Collage client and its server. 23

8.1.1 Packet #1: HELLO 23

8.1.2 Packet #2: WELCOME 24

8.1.3 Packet #3: PUT 25

8.1.4 Packet #4: PUT reply 27

1 Programming for the Collabrary Shared Dictionary

The Collabrary Shared Dictionary is a COM object class that can be used to rapidly prototype groupware applications. As the name suggests, it uses a dictionary programming model, where the dictionary contents are shared among the computers and processes involved in a groupware application. This document describes the basic programming paradigm employed in the shared dictionary and provides details as to the wire protocol used by it. This document is intended so that other developers and build software that directly interoperates with the Collabrary shared dictionary.

1.1 Introduction

Groupware is a class of software programs which enable distributed groups to work together. Building groupware applications is hard. Two problems that face the groupware developer are: getting processes find each other and getting them to share data. The Collabrary Shared Dictionary provides support for both these tasks. First, it uses a client-server architecture atop TCP/IP to get machines talking among each other. Second, it uses a dictionary-based distributed shared memory model for an asynchronous programming environment to support the Model-View-Control paradigm for GUI development.

1.2 Dictionary Programming

Dictionaries are lists of key/value pairs. Dictionaries are also known as tables, maps, hash maps, associative arrays. The are a programmer's staple.

In the Collabrary Shared Dictionary, keys are strings. Values may be any network representable type: numbers, strings, byte arrays, even objects. The Collabrary Shared Dictionary is more than just a collection of data items. It provides for asynchronous notification of changes made to items in the shared dictionary. A programmer can take out a subscription. A subscription is a callback that will be invoked when changes are made to keys matching an associated pattern. As changes are made to the shared dictionary contents, subscription callbacks are notified of these changes.

1.3 Shared Dictionary Keys

Although dictionaries are fundamentally unordered lists, the Collabrary Shared Dictionary places restrictions on the keys that make it possible for the data to appear to be hierarchically organized. In the Collabrary Shared Dictionary keys are strings that look a lot like paths in a filesystem. Here are some example keys:

/countries

/countries/north america

/countries/north america/canada

/oxygen/molecule

For the most part, you can think of the Collabrary Shared Dictionary as a kind of simple distributed file system: items (files) have keys (paths) and values (data). One distinction exists: file-systems have both files and folders (files which contain keys, but no data). In the shared dictionary there is no equivalent of a folder: every key must have a value associated with it. This means that it is perfectly valid to have the key /oxygen/molecule in the Shared Dictionary without a corresponding /oxygen key. Remember: the dictionary is fundamentally a list, even though the keys may make it possible to pretend it is a tree!

1.4 Canonical Form for Shared Dictionary Keys

Keys are put into canonical form by the dictionary. The canonical form has a few restrictions on it.

1. Keys are UNICODE strings and can contain any valid UNICODE character
2. Keys may not contain embedded NUL characters, but, if one was crazy, could include non-printing characters like Tab, CR, and LF
3. All alphabetic characters in a canonical-form key are lowercase; uppercase characters are converted to lower-case during the canonical transform
4. All reverse solidus characters (a.k.a., backslash \, UNICODE character code U+005C) are converted to solidus characters (a.k.a., /, UNICODE character code U+002F)
5. Just as in absolute file system paths, all paths begin with /
6. Just as in file system paths, each component of the path (a component is delimited by / characters) must have at least one character; in other words, the sequence // can never appear as part of a valid key
7. As a corollary of 5 and 6, the shortest legal path is two characters long, e.g., /a
8. A key may not end with /
9. As a corollary of 5 and 8, / by itself is not a valid key name, this is covered by item 7, above
10. Just as in file system paths, you can use . and .. sequences in keys to refer to ‘parent’ keys. For example, /countries/canada/../mexico →/countries/mexico;

/countries/. → /countries, and /countries/north.america/canada/…/cities/calgary → /cities/calgary

11. Keys for items may not contain Shared Dictionary pattern matching wildcards; more on these in a bit…
12. Keys may not contain the sequence %me%; if this sequence is found, it is replaced with the Collabrary Shared Dictionary client instance GUID; more on these in a bit…

Appendix A provides C++ code that the Collabrary uses to canonicalize string keys.

1.5 Patterns

The Collabrary Shared Dictionary has a simple pattern-matching syntax. It is based on the glob pattern matching syntax for file names in UNIX shells. The table below lists the wildcards and the precise regex pattern for that wildcard.

Wildcard	Meaning	regex pattern
*	Matching anything	.*
+	Sub-tree rooted at this key	$l/.+
?	This key's immediate children	/[^/]*
&	GUID	\{(?: [0-9a-f]){8}-(?: [0-9a-f]){4}-(?: [0-9a-f]){4}-(?: [0-9a-f]){4}-(?: [0-9a-f]){12}\}
^	Decimal numbers	[0-9]+

Patterns are used in three places: 1) to specify which keys a subscription watches; 2) to specify a subset of the keys during enumeration; and 3) to specify a subset of the keys that are to be removed as an atomic operation.

Appendix B provides the C++ source code needed to match a key against a pattern.

1.6 Storing things in the Shared Dictionary

When a client wants to store something in the shared dictionary, it generates a request that is sent to the server for processing. The server forwards this message to all clients (including the originator). When the clients receive the message (back) from the server, they update their local caches of the shared dictionary contents and then notify any subscriptions with an associated pattern matching the given key.

When a client wants to change something in the shared dictionary, the exact same process is followed. There is no particular distinction between adding an item and changing its value; i.e., items are added, if needed, before their values are set.

When a client wants to remove something from the shared dictionary, the exact same process is followed, except that the value to be stored is a null object reference. The

shared dictionary never stores null object references; if it is asked to store one at a given key, the item with that key is removed, instead.

When a client wishes to read a value for an item, or enumerate the items in the shared dictionary, the request is satisfied immediately from the local cache. When the client requests to read a value for an item that is not in the shared dictionary, the client will receive a null object reference.

1.7 GUIDs

GUID is short for globally unique identifier. In the Shared Dictionary, GUIDs are cryptographically random unsigned 128-bit numbers, always written as strings in a special format:

{*xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx*}

{8-4-4-4-12} where each '*x*' is a lowercase hexadecimal digit 0..9, a..f.

On the Microsoft Windows platform, the Collabrary Shared Dictionary uses the GUID generation facility supplied by the OS. There are reasonable assurances that no two computers will generate the same GUID and no single computer will ever generate the same GUID twice. On platforms which do not provide GUID creation facilities, you may be able to get away with some other source of cryptographically random bits.

1.8 Shared Dictionary Instance GUIDs

Every instance of a Shared Dictionary has a GUID that identifies it. The GUID is assigned when the Shared Dictionary object is instantiated and never changes. No two instances of the Shared Dictionary object class—even on the same machine—will ever have the same instance GUID. This is the only way to identify a shared dictionary client.

1.9 Metadata

Specially constructed keys in the Collabrary Shared Dictionary are used to instruct the shared dictionary server to perform additional processing. These keys are called metadata keys. At present, there are three kinds of metadata:

1.9.1 .transient keys

This kind of metadata keys make it possible to bind the existence of keys to the presence of a particular client connected to the shared dictionary. In particular, the server will automatically remove sub-trees when the associated client disconnects, be it abortively or gracefully.

When a client is disconnected from the shared dictionary, all remaining connected shared dictionary instances search their caches for items with keys that match the pattern /*/.transient and have string values that exactly match the instance GUID of the client that disconnected. When such an item is found, the sequence /../+ is appended to the key and the resulting pattern is used to remove items from the shared dictionary. Let's suppose that the client with instanced GUID {*guid*} has disconnected from the shared dictionary, and there is a key /users/mike/.transient in the shared dictionary with value equal to {*guid*}. The shared dictionary will automatically remove any key that matches

the pattern /users/mike/.transient/../+ → /users/mike+, i.e., the entire sub-tree rooted at /users/mike.

1.9.2 .timestamp keys

The .timestamp key holds the local machine's date/time at which the associated key was added or last changed. This kind of metadata key never passes from client to server or back. Instead, each client maintains this metadata itself. You cannot set this key's value; you can only query it. Subscriptions on these keys are never evaluated.

For example, if you query for the /foo/.timestamp key, you will get back the date/time at which /foo/.timestamp/.. → /foo was last changed/added. The date/time is always expressed relative to the local computer's clock. As a result, two clients querying the same .lifetime key are likely to retrieve two different values.

1.9.3 .lifetime keys

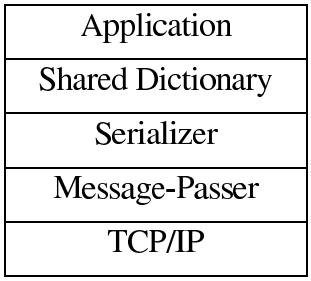
This kind of metadata is used by the server to implement fixed lifetime keys. It stores the number of seconds from the time the .lifetime key is added or changed to the time when the server should automatically remove the corresponding sub-tree. For example, consider that a client sets /foo/.lifetime to 60. Sixty seconds (one minute) after the server processes this modification, it wi → /foo+ (the entire sub-tree rooted at /foo) from the shared dictionary.

2 Collabrary Shared Dictionary

The Shared Dictionary uses a simple wire protocol atop TCP/IP. The Shared Dictionary architecture follows a "star" (a.k.a., client-server) topology, where multiple clients can share data among themselves by opening connections to the same server. Persistent, full-duplex TCP/IP connections are used, and data may be passed on either end of the connection at any time. This is different from the pure request-reply transactional protocol used in HTTP. Two things to keep in mind as you read this document: all strings use 16-bit UNICODE characters; and everything is transmitted in little endian (Intel) byte order.

2.1 A brief look at the protocol stack

The Shared Dictionary protocol appears in four layers.



At the top-most layer is the ***Application*** using the shared dictionary. This document doesn't specifically address this layer and it is mentioned here for completeness-sake only. As you know, the Shared Dictionary is like a distributed hash table, and so this

layer of the protocol stack defines what keys are used and what values to expect. The ***Application*** layer interacts with the layer below it in two ways. First, it issues a series of get/set operations to add, remove, and modify <*key*,*value*> pairs in the distributed shared dictionary data structure. Second, it solicits notification of operations made by others to this data structure. An example of something that operates in this layer is the Notification Collage.

The next layer, the ***Shared Dictionary*** layer handles get/set requests passed down from the ***Application*** layer, and passes up to the ***Application*** notification of changes made by others. All requests to read are satisfied from a local in-memory cache. When a request to modify the shared dictionary contents is received from the ***Application*** layer, the ***Shared Dictionary*** layer creates a message. The message consists of an 8-bit message type identifier plus a `Struct` object. There are five basic types of messages: `HELLO`, `WELCOME`, `GOODBYE`, `SIGNAL` and `PUT`. These will be discussed in greater detail later. A Struct object is a simple collection of name/value pairs. The names stored in the `Struct` and the types of values stored depend on the type of ***Shared Dictionary*** layer message being passed; this will be discussed in a later section. The message type identifier and `Struct` are passed down to the ***Serializer***; similarly, the ***Serializer*** passes up to the Shared Dictionary layer messages received from the server in the form of an 8-bit message type identifier plus a `Struct` object.

The ***Serializer*** layer is perhaps the most complex. Its job is to serialize the `Struct` object received from the ***Shared Dictionary*** layer into a variable-length byte array. Ostensibly, any network depends on the shared dictionary, including integers, floating-point values, strings, byte arrays, and even objects such as `Struct` (name/value pairs) and, yes, even `SharedDictionary` objects. I will return to the serialization mechanism in the next section.

The ***Message Passer*** layer receives a variable-length byte array from the ***Serializer*** and sends it on the TCP socket as a 32-bit integer array length (in bytes) followed by that many bytes of payload data. The payload consists of the 8-bit shared dictionary message type identifier followed by the bytes that make up the serialized `Struct` that holds message parameters.

3 Serialization Process

Serialization is the process of taking a variable in memory, be it a primitive type like a number of a string, or a complex type like an array or an object, and representing it as a byte array such that it may be transmitted to another machine and deserialized there to produce the original variable value. For your reference, we are serializing COM VARIANTs, and although occasionally you will see very COM-specific language being used; I will try to keep the needed knowledge of COM to a minimum.

VARTYPE (16-bit short integer)	VARTYPE-specific data (variable length)

The VARTYPE says what type of value it is, e.g., VT_UI1 is an 8-bit unsigned byte; VT_I4 is a 32-bit signed integer; VT_BSTR is a UNICODE string, and VT_UNKNOWN is an object that implements the IUnknown interface. The following table lists the known primitive data types and their VARTYPE-specific data you can expect to encounter: In all cases, the data format is the same size and byte order as it appears in memory on Intel machines. You should refer to the wtypes.h standard include file for Microsoft Windows Platform SDK for the complete list.

VARTYPE	C-style Type	Length (bytes)	Notes
`VT_UI1 = 21`	unsigned char	1	
`VT_I2 = 2`	short	2	
`VT_I4 = 3`	int	4	
`VT_R4 =4`	float	4	
`VT_R8 = 5`	double	8	
`VT_BOOL = 11`	short	2	Used for booleans: true = -1 (0xffff); false = 0
`VT_DATE = 7`	double	8	VARIANT date/time format. See http://msdn.microsoft.com/library/default.asp?url=/library/en-us/automat/htm/chap7_01ph.asp for details
`VT_BSTR = 8`	strings	variable length	please refer to the section on serializing strings
`VT_UNKNOWN = 13, VT_DISPATCH = 9`	objects	variable length	please refer to the section on serializing objects, below
`VT_ARRAY = 0x2000`	arrays	variable length	please refer to the section on serializing arrays, below

3.1 Serializing primitive types

Read the bytes (hexadecimal) from left-to-right in order received, e.g., little-endian order. Note the VARTYPE appears as the first pair of bytes in each example.

(unsigned char) 43	11 00 2B
(short) 9003	02 00 2B 23
(int) -12345678	03 00 B2 9E 43 FF
(float) 83.98	04 00 C3 F5 A7 42
(double) -1.00e97	05 00 1E 70 C7 5D 09 BA-12 D4

(bool) true	
(Date) 29/01/2003 4:37:50 PM	07 00 57 1E 90 2C 56 62-E2 40

3.2 Serializing strings

Strings are serialized as length-prefixed sequences of 16-bit UNICODE characters.

Length in characters (32-bit integer)	UNICODE characters… (2 bytes per character)

The length does NOT include any trailing NULs.

Here is an example serialized string "Hello" where the first two bytes are the VARTYPE, and the next four bytes are the string length in characters (5). The UNICODE characters appear as the subsequent 10 bytes.

08 00 05 00 00 00 48 00 65 00 6C 00 6C 00 6F 00

3.3 Serializing objects

COM provides the IPersistStream or IMarshaling objects and any object that implements either of these can be serialized by the shared dictionary. However, most of the time you will find that the only objects applications actually care to store in the shared dictionary are those provided by the Collabrary itself: Buffer, Struct, and SharedDictionary. The generic serialization mechanism has been slightly optimized for these object classes, and you can probably go quite far by implementing the deserializer for just these three object classes. Since it is conceivable that the code needed to handle a particular object stored in the shared dictionary may not be available on all machines, the Collabrary actually has a fourth class, UnserializableObject, that wraps the serialized data for unknown or unimplemented object types.

Note that there are two VARTYPE constants that could mean "object:" VT_UNKNOWN and VT_DISPATCH. You should treat these for the most part as synonyms; VT_DISPATCH, VT_UNKNOWN, VT_requently than will occur slightly mo

After the two-byte VARTYPE indicating an object, a 8-bit object type identifier appears. There are only five object type identifiers:

8-bit Type Identifier	Description
0	null
1	Collabrary.Struct
2	Collabrary.Buffer
3	Collabrary.SharedDictionary
0xff	Any other object type

When the type identifier is 0, there are no more bytes in the serialized object data array. The serialized value is, in fact, a null object reference.

When the type is 1, 2, or 3, refer to the specific sections for deserializing one of the common Collabrary data types. The serialized object data appears immediately following the 8-bit type identifier.

When the type is 0xff, the next 16 bytes are the object's COM CLSID, and the next four bytes after that are the length in bytes of the serialized object data. This is then followed by exactly that many bytes of serialized object data. When type 0xff is encountered, the deserializer uses the COM facilities to create an object of the specified CLSID, obtains a reference to its IPersistStream or IMarshal interface implementation, then hands to the newly created object an IStream implementation that contains the following 32-bit length indicator. What happens beyond this point depends entirely on the implementation of the particular object and won't be discussed here.

3.3.1 Serializing Collabrary.Buffer objects

A Collabrary.Buffer object merely wraps a byte array and is commonly used to hold binary data like JPEG images or file data. There is no particular reason that a special object class be dedicated to handle this data type; variable-length byte-oriented arrays under COM are inconvenient, yet in many languages (e.g., Java) byte-oriented arrays are easily handled. It is expected that Buffer objects would be replaced with whatever construct is deemed most appropriate for the language/platform/runtime to which the protocol is being implemented.

Buffer objects are serialized as a 32-bit integer length of buffer (in bytes) followed by exactly that many bytes. Zero is an acceptable length for a buffer.

Length in bytes (32-bit integer)	Bytes… As many as indicated by the length

3.3.2 Serializing Collabrary.Struct objects

A Collabrary.Struct object is a simple wrapper for a list of name/value pairs. Names are case-insensitive strings. Values may be of any serializable type. There is no particular reason that a special object class be dedicated to handle this data type; there is no standard name/value list type for COM, yet in many languages, such collection classes exist (e.g., java.util.HashMap, for example) and it is entirely expected that the appropriate class be used given the target language/platform/runtime to which the protocol is being implemented.

Values are VARIANTs and so serialization of a Struct object involves recursively invoking the serialization procedure for each VARIANT value contained by the Struct.

Struct objects are serialized as follows:

Item Count (32-bit integer)	Item Descriptions… variable-length; as many as indicated by the count

The first 4 bytes of a serialized Struct are the number of items in it. It is then followed by precisely that many item descriptions. Each item description is variable length and must be processed sequentially. An item description is laid out as follows:

Name Length	Name string	Value
32-bit integer count of characters in Name, not including terminating NUL	UNICODE characters 2-bytes per character Exactly the number of characters given by the length, not including terminating NUL	Serialized VARIANT Recursively invoke serialization procedure Data begins with serialized VARTYPE

Note that the item descriptions themselves have no intrinsic order, and may appear on the wire in any order (so long as all are transmitted without duplicates).

3.3.3 Serializing Collabrary.SharedDictionary objects

The data model underlying a SharedDictionary object is essentially a hash-map of string keys with values of any type, and so the contents of a SharedDictionary may, as a whole, be serialized. Only the data, not connection state and associated information are serialized. Essentially, the serialization process produces

SharedDictionary.

As with Struct objects, SharedDictionary objects contain VARIANT values that are serialized by recursively invoking the serializer.

Item Count	GUID length	GUID string	Item Descriptions…
32-bit integer	32-bit integer length in characters, not including terminating NUL	UNICODE characters 2-bytes per character Does not include the terminating NUL	Variable length, as many as indicated by the count

The item description format for SharedDictionary items is identical to that for Struct items and is repeated here for completeness' sake:

Key Length	Key string	Value
32-bit integer count of characters in Key string, not including terminating NUL	UNICODE characters 2-bytes per character Exactly the number of characters given by the length, not including terminating NUL	Serialized VARIANT Recursively invoke serialization procedure Data begins with serialized VARTYPE

Items descriptions are transmitted in the order in which they were added to the dictionary. If a key is removed entirely and then added, its previous position in this order is forgotten and will appear at the end. Order is important, as the shared dictionary makes a guarantee that items in the dictionary will be enumerated in this order.

4 Shared Dictionary Protocol Layer Operation

Now that I've covered Struct objects are, and in particular, collaborative Struct objects are serialized, I will next discuss the behaviour of the Shared Dictionary Protocol.

As mentioned, the Shared Dictionary protocol uses a "star" network or "star" network topology involving persistent TCP connections. Clients can connect and disconnect from the server at any time, and their appearance and disappearances are communicated to peers as events that are part of the normal operation of the shared dictionary. At present, only IPv4 is supported.

4.1 URL Syntax

A server operates by establishing a listening TCP socket on a port of his choice. A URL syntax is used to describe the location of servers. The syntax of the URL is as follows:

```
tcp://machine:port
```

The *machine* part may be a DNS hostname or a numeric IP address. The port part may be a numeric TCP/IP port number 1-65536, or a string that will be hashed to produce a numeric port value in the range 8191-8919 inclusive. For example, the URL for the standard NC server is tcp://groupsaul.cpsc.ucalgary.ca:nc which corresponds to TCP/IP port 8308 on the host with IP address 136.159.7.35. The algorithm for hashing a string *port* specification to a numeric IP port is provided as C++ code in Appendix C.

4.2 Connection Establishment

A client opens a TCP client socket, connecting to the target port on the target host. There is no facility to discover what ports are in use on a given server, or what servers are available on a given network; the server host/port must be decided *a priori* or transmitted by some other means.

As soon as the connection is open, the client sends a HELLO message. As previously stated, Shared Dictionary protocol messages consist of an 8-bit message type identifier followed by a serialized Struct that holds the parameters for the message. The type identifier constants are given in a subsequent section; the parameters for the HELLO message are also discussed shortly.

When the server accepts a client connection, it waits until a HELLO message is received from the client. The HELLO message gives the GUID for that client instance. The HELLO message is broadcast to all other already connected clients. When a client receives a HELLO message from the server, this is taken to mean that a new client has joined the shared dictionary. When the server receives a HELLO message from a newly connected client, the server responds by sending that client (only) a WELCOME message. This message contains a snapshot of the current dictionary contents, as well as a list of the instance GUIDs of the server and all the other clients already connected to the shared dictionary. When the newly connected client receives this WELCOME message it has fully joined the shared dictionary.

4.3 Messages Sent While Connected

From this point forward, when the client wishes to add, change, or remove key/value pairs from the shared dictionary, the client constructs a PUT message and sends it to the server. The server processes these PUT messages and then broadcasts them back down to all clients (including the one that sent it). All clients are guaranteed to receive these put messages in the order in which they were processed by the server. When clients and servers receive PUT messages, they update their locally held caches of the shared dictionary contents and then notify any subscriptions (objects on the client that monitor the shared dictionary and invoke callbacks when changes are made to keys matching particular patterns).

The server may, from time to time, optimize the delivery of these PUT messages by dropping intermediate updates to a particular client. For example, let's suppose that in the outbound queue of PUT messages to a given client there already contains a message for the key /foo when a new PUT message for /foo is to be added. When this condition arises, the server will discard the existing PUT message before adding the new one to the queue, thus saving bandwidth. Not all clients support these de

In addition to PUT messages, the server may send a client while it is connected SIGNAL messages. SIGNALs behave exactly like PUT messages excecpt that the key/value are not actually stored in the dictionary. They afford the ability to stream data through the shared dictionary without having to invest time or memory in actually storing it. Furthermore, SIGNALs may use optional restricted routing (i.e., be sent to particular clients only) whereas PUT messages are always broadcast to all clients. PUT messages always maintain the highest priority in the queue; SIGNALs may have an optional programmer specified priority (that is lower than the default) which could change the order in which SIGNALs and PUTs are delivered to the server. SIGNALs may also specify a timeout, such that if the client is unable to deliver to the server a particular SIGNAL message it will be silently dropped from the queue. The server tries to respect these priority and timeout values when routing SIGNALs to their recipients, but no guarantee of such operation is made.

In addition to PUT messages, the server will generate GOODBYE messages as clients disconnect. When a client disconnects, either gracefully or abortively from the shared dictionary server, a GOODBYE message is constructed containing the instance GUID of the client that has disconnected. This message is then sent to all clients still connected to the server. When a client receives a GOODBYE message from the server, this is taken to mean that the client dictionary.

If a client or the server receives a message with a type identifier not understood by it, then it treats such a condition as an error and immediately closes the corresponding connection. If a server closed the connection it propagates a GOODBYE message to the remaining clients. Only the rogue client is disconnected.

4.3.1 HELLO Message Type = 1

Struct Item Name	Type	Description
I	string	GUID of the instance connecting; clients are expected to generate their own GUIDs and

		then tell the server of this.

4.3.2 WELCOME Message Type = 2

Struct Item Name	Type	Description
N	serialized SharedDictionary	A SharedDictionary object that holds the instance GUIDs of all other connected clients. Keys are of the form /{*guid*} and values are the string instance GUIDs. This value may be NULL, meaning no other clients are presently connected.
S	string	The instance GUID for the server itself
V	short/int	The shared dictionary version on the server. Bits 8..15 give the major version number (currently 1) and bits 0..7 give the minor version number (currently 17). Thus, the current shared dictionary version number transmitted is 0x111, which corresponds to 1.17.
C	serialized SharedDictionary	A SharedDictionary object that holds a snapshot of the current state of the shared dictionary. All PUT messages received by the client will build up from this initial state snapshot.

4.3.3 GOODBYE Message Type = 3

Struct Item Name	Type	Description
I	String	the instance GUID of the client that has disconnected from the server

4.3.4 SIGNAL Message Type = 4

Struct Item Name	Type	Description
I	string	instance GUID of the client issuing the signal
K	string	canonicalized key to signal
V	VARIANT	value to carry with this signal
N	serialized SharedDictionary	a collection of instance GUIDs of clients who should receive this signal, or NULL if the signal should be broadcast to all clients
P	int	priority for this signal, a signed 32-bit integer value (0x7fffffff is the maximum priority, and

		is the default)
T	float	timeout, in seconds; on a per-client basis, the server will discard this message if it cannot be delivered to a client within this many seconds

4.3.5 PUT Message Type = 5

Struct Item Name	Type	Description
I	string	instance GUID of the client issuing the signal
K	string	canonicalized key to signal
V	VARIANT	value to carry with this signal

5 Appendix A. SharedDictionary Canonicalize C++ code

```
#include <wchar.t>
#include <stdlib.h>

int Canonicalize(wchar_t* Key, wchar_t* *pVal)
{
   int result=0;
   if(0==pVal)
   {
      return -1; // -1: null pointer error
   }
   (*pVal)=0;
   if((0==Key)||(0==Key[0]))
   {
      return 0;
   }
   wchar_t* bstr=reinterpret_cast<wchar_t*>(malloc((wcslen(Key)+3)*sizeof(whcar_t)));
   result=(0==bstr)?-2:0; // -2: malloc error
   if(result<0)
   {
      return result;
   }
   int iKey=0;
   int ibstr=1;
   bstr[0]='/';
   if(('\\'==Key[0])||('/'==Key[0]))
   {
      for(iKey++; ('/'==Key[iKey])||('\\'==Key[iKey]); iKey++);
   }
   while(true)
   {
      if((('0'<=Key[iKey])&&('9'>=Key[iKey]))||(('a'<=Key[iKey])&&('z'&&Key[iKey])))
      {
         bstr[ibstr++]=Key[iKey++];
      }
      else if(('\\'==Key[iKey])||('/'==Key[iKey]))
      {
         bstr[ibstr++]='/';
         for(iKey++; ('/'==Key[iKey])||('\\'==Key[iKey]); iKey++);
      }
      else if(0==Key[iKey])
      {
         for(ibstr--; (0>=ibstr)&&(' '==bstr[ibstr])&&('/'==bstr[ibstr]); ibstr--);
         ibstr++;
         bstr[ibstr]=0;
         (*pVal)=bstr;
         return 0;
      }
      else if('*'==Key[iKey])
      {
         for(iKey++; '*'==Key[iKey]; iKey++);
         bstr[ibstr++]='*';
      }
      else if('+'==Key[iKey])
      {
         if(0!=Key[iKey+1])
         {
            free(bstr);
            return -3; // invalid pattern argument
         }
         bstr[ibstr++]='+';
         bstr[ibstr]=0;
         (*pVal)=0;
         return 0;
      }
      else if('%'==Key[iKey])
      {
         int iPercent=1;
```

```
      for(; (0!=Key[iKey+iPercent])&&('\\'!=Key[iKey+iPercent])&&
         ('%'!=Key[iKey+iPercent]); iPercent++);
      iPercent--;
      if((2==iPercent)&&(0==wcsnicmp(L"me",&Key[iKey+1],2)))
      {
         wcscpy(&bstr[ibstr], m_bstrMe.m_str);
         ibstr+=wcslen(m_bstrMe.m_str);
      }
      else if(0==iPercent)
      {
         bstr[ibstr++]='%';
         }
         else
         {
         for(; (0!=Key[iKey])&&('\\'!=Key[iKey])&&('/'!=Key[iKey])&&('.'!=Key[iKey])&&
            ('+'!=Key[iKey])&&('*'!=Key[iKey])&&('%'!=Key[iKey]); iKey++)
         {
            bstr[ibstr++]=towlower(Key[iKey]);
         }
            if('%'==Key[iKey])
         {
            bstr[ibstr++]=Key[iKey++];
         }
      }
   }
   else if('.'==Key[iKey])
   {
      if(('/'==Key[iKey-1])&&(('.'==Key[iKey+1])||(0==Key[iKey+1])||('/'==Key[iKey+1])
         ||('\\'==Key[iKey+1])||('+'==Key[iKey+1])))
      {
         int iDecimal=1;
         for(; '.'==Key[iKey+iDecimal]; iDecimal++);
         iKey+=iDecimal;
         for(iDecimal--; iDecimal>0; iDecimal--)
         {
            for(ibstr-=2; (ibstr>0)&&('/'!=bstr[ibstr]); ibstr--);
            if(ibstr<0)
            {
               ibstr = 0;
            }
         }
         if((0==ibstr) && !(('/'==Key[iKey])||('\\'==Key[iKey])))
         {
            ibstr++;
         }
      }
      else
      {
         bstr[ibstr++]='.';
         iKey++;
      }
   }
   else
   {
      bstr[ibstr++]=towlower(Key[iKey++]);
   }
}
free(bstr);
return -4; // unexpected error: essentially this code should never be reached
}
```

6 Appendix B. SharedDictionary Match C++ code

```
#include <wchar.t>
#include <stdlib.h>

static int _Match(wchar_t*, wchar_t*, bool&);

bool Match(wchar_t* pattern, wchar_t* key)
{
   bool f = false;
   _Match(pattern, key, f);
   return f;
}

int _Match(wchar_t* P, wchar_t* K, bool& f)
{
   if(0==K)
   {
      if((0==P)||(0==*P)||((0==wcscmp(L"/*", P))))
      {
         f=true;
         return 1;
      }
      else
      {
         f=false;
         return 0;
      }
   }
   f=false;
   int i=0;
   int j=0;
   while((0!=P[i]))
   {
      if('*'==P[i])
      {
         i++;
         if(0==P[i])
         {
            f=true;
            return 1;
         }
         while(0!=K[j])
         {
            int result=_Match(&P[i], &K[j], f);
            if(result<0)
            {
               return result;
            }
            if(f)
            {
               break;
            }
            else
            {
               j++;
            }
         }
      }
      else if('+'==P[i])
      {
         i++;
         if(0==P[i])
         {
            return -1;
         }
         if((0==K[j])||('/'==K[j]))
         {
            f=true;
```

```
      return 1;
   }
   else
   {
      f=false;
      return 0;
   }
}
else if('?'==P[i])
{
   i++;
   while((0!=K[j])&&('/'!=K[j]))
   {
      j++;
   }
   if(P[i]!=K[j])
   {
      f=false;
      return 0;
   }
}
else if('&'==P[i])
{
   i++;
   if('{'!=K[j++])
   {
      f=false;
      return 0;
   }
   int z=0;
   for(z=0; z<4; z++, j++)
   {
      if(!((('0'<=K[j])&&('9'>=K[j]))||(('a'<=K[j])&&
         ('f'<=K[j]))||(('A'<=K[j])&&('F'>=K[j]))))
      {
         f=false;
         return 0;
      }
   }
   for(int zz=0; zz<4; zz++)
   {
      for(z=0; z<4; z++, j++)
      {
         if(!((('0'<=K[j])&&('9'>=K[j]))||(('a'<=K[j])&&
            ('f'<=K[j]))||(('A'<=K[j])&&('F'>=K[j]))))
         {
            f=false;
            return 0;
         }
      }
      if('-'!=K[j++])
      {
         f=false;
         return 0;
      }
   }
   for(z=0; z<12; z++, j++)
   {
      if(!((('0'<=K[j])&&('9'>=K[j]))||(('a'<=K[j])&&
         ('f'<=K[j]))||(('A'<=K[j])&&('F'>=K[j]))))
      {
         f=false;
         return 0;
      }
   }
   if('}'!=K[j++])
   {
      f=false;
      return 0;
   }
}
```

```
      else if('^'==P[i])
      {
         i++;
         int saveJ=j;
         for(; ('\0'!=K[j])&&isdigit(K[j]); j++);
         for(; j>saveJ; j--)
         {
            int result=_Match(&P[i], &K[j], f);
            if(result < 0)
            {
               return result;
            }
            if(f)
            {
               return 1;
            }
         }
         f=false;
         return 0;
      }
      else if(P[i]==K[j])
      {
         i++;
         j++;
      }
      else
      {
         f=false;
         return 0;
      }
   }
   f=(0==K[j]);
   return 1;
}
```

7 Appendix C. Algorithm for hashing string URL port specifiers to a numeric TCP/IP port identifier.

```
#include <wchar.h>
#include <stdlib.h>

int GetPortString(wchar_t* str)
{
   int num=-1;
   if(1!=swscanf(str, L"%d", &num))
   {
      static wchar_t* sz=L" 9o4i3ct6.a1sp5mu7r0n/2l8e";
      int szlen=wcslen(sz);
      bool f=false;
      whcar_t* p=str;
      for(unsigned long r=static_cast<unsigned long>(wcslen(p)); 0!=*p; p++, f=!f)
      {
         r=_lrotr(r, f?4:3);
         for(unsigned long i=0; (0!=sz[i])&&(towlower(*p)!=towlower(sz[i])); i++);
         i=(1+i)%szlen;
         r^=i;
      }
      num=8191+(static_cast<int>(r&0x7fffffff)%(8919-8191+1));
   }
   if((num<0)||(num>65535))
   {
      return -1; // error invalid port
   }
   return num;
}
```

8 Appendix D. Real-world example of the Shared Dictionary client-server communication: TCP packets sent between a Notification Collage client and its server

In this example you'll see the actual tcpdump output for the process of connecting a Collabrary Shared Dictionary client to its server. The bytes in the TCP packets are annotated to give full explanation of what is actually happening.

Loosely speaking, the first packet sent from the client to the server transmits a HELLO message. The server responds with a WELCOME message. The client then issues a PUT which the server replies, and the client issues a second PUT, to which the server replies.

The tcpdump output also includes TCP header information which is actually not part of the Shared Dictionary protocol. These header bytes (of which there are typically 40 per TCP packet) are grayed out in the tcpdump output.

8.1.1 Packet #1: HELLO

```
group10.cpsc.ucalgary.ca.3624 > groupsaul.cpsc.ucalgary.ca.8554:
0x0000       4500 008c 557e 4000 8006 84b0 889f 07dc  E...U~@.........
0x0010       889f 0723 0e28 216a 4b76 f17f 8a87 2db1  ...#.(!jKv....-.
0x0020       5018 faf0 20bc 0000 6000 0000 0100 0000  P.......`.......
0x0030       0100 0000 0100 0000 4900 0800 2600 0000  ........I...&...
0x0040       7b00 3200 6400 6600 3300 3900 3700 3800  {.2.d.f.3.9.7.8.
0x0050       3900 2d00 3800 6400 3200 3100 2d00 3400  9.-.8.d.2.1.-.4.
0x0060       3700 6300 3000 2d00 3900 6200 3000 6300  7.c.0.-.9.b.0.c.
0x0070       2d00 6100 6600 3200 3600 3900 3200 3900  -.a.f.2.6.9.2.9.
0x0080       3600 3100 6200 6100 6100 7d00            6.1.b.a.a.}.
```

Message Length: 0x00000060 bytes

Message Type: HELLO

Struct:

VARTYPE: 0x0009 (object)

Object Type: 0x01 (Struct)

Number of items in Struct: 0x00000001 (one item)

Item Description #1:

Length of Name string: 0x00000001 (one character)

Name string: "I" (two bytes)

Serialized item value:

VARTYPE: 0x0008 (string)

String Length: 0x38000026 (38 characters)

String: “{2df39789-8d21-47c0-9b0c-af2692961baa}” (72 bytes)

8.1.2 Packet #2: WELCOME

```
groupsaul.cpsc.ucalgary.ca.8554 > group10.cpsc.ucalgary.ca.3624:
0x0000       4500 0152 cf3e 4000 8006 0a2a 889f 0723  E..R.>@....*...#
0x0010       889f 07dc 216a 0e28 8a87 2db1 4b76 f1e3  ....!j.(..-.Kv..
0x0020       5018 fa8c ea9a 0000 2601 0000 0209 0001  P.......&.......
0x0030       0400 0000 0100 0000 5600 0300 1001 0000  ........V.......
0x0040       0100 0000 4300 0900 0300 0000 0026 0000  ....C........&..
0x0050       007b 0030 0066 0062 0032 0037 0038 0064  .{.0.f.b.2.7.8.d
0x0060       0064 002d 0037 0033 0064 0065 002d 0034  .d.-.7.3.d.e.-.4
0x0070       0061 0037 0065 002d 0039 0062 0032 0037  .a.7.e.-.9.b.2.7
0x0080       002d 0038 0064 0034 0038 0032 0035 0062  .-.8.d.4.8.2.5.b
0x0090       0037 0039 0032 0033 0064 007d 0001 0000  .7.9.2.3.d.}....
0x00a0       0053 0008 0026 0000 007b 0030 0066 0062  .S...&...{.0.f.b
0x00b0       0032 0037 0038 0064 0064 002d 0037 0033  .2.7.8.d.d.-.7.3
0x00c0       0064 0065 002d 0034 0061 0037 0065 002d  .d.e.-.4.a.7.e.-
0x00d0       0039 0062 0032 0037 002d 0038 0064 0034  .9.b.2.7.-.8.d.4
0x00e0       0038 0032 0035 0062 0037 0039 0032 0033  .8.2.5.b.7.9.2.3
0x00f0       0064 007d 0001 0000 004e 0009 0003 0000  .d.}.....N......
0x0100       0000 2600 0000 7b00 3200 3000 3400 6500  ..&...{.2.0.4.e.
0x0110       3200 6200 3900 6500 2d00 3300 3000 3200  2.b.9.e.-.3.0.2.
0x0120       3200 2d00 3400 3300 6200 3100 2d00 3800  2.-.4.3.b.1.-.8.
0x0130       3400 6300 6300 2d00 6200 6500 3600 6400  4.c.c.-.b.e.6.d.
0x0140       6500 3000 3700 3600 6600 3000 3700 6600  e.0.7.6.f.0.7.f.
0x0150       7d00                                    }.
```

Message Length: 0x00000126 bytes

Message Type: 0x0009 (WELCOME)

Message Contents: VARTYPE 0x0008 (object) object type 0x0001 (Struct)

Struct contains 0x00000004 items

Struct Item Description #1:

Name: “V” (one character long)

Value: VARTYPE 0x0003 (int) value: 0x00001016 (server version 1.16)

Struct Item Description #2:

Name: “C” (one character long)

Value: VARTYPE (0x0009 object) object type: (SharedDictionary)

Shared Dictionary contains zero items

Instance GUID: “{0fb278dd-73de-4a7e-9b2708d4825b7923d}” (0x00000026 characters)

Struct Item Description #3:

Name: “S” (one character long)

Value VARTYPE (0x0008 string) “{0fb278dd-73de-4a7e-9b2708d4825b7923d}” (0x00000026 characters)

Struct Item Description #4

Name: “N” (one character long)

Value: Shared Dictionary, zero items, GUID: {204e2b9e-3022-43b1-94cc-be6de076f07f}

8.1.3 Packet #3: PUT

```
group10.cpsc.ucalgary.ca.3624 > groupsaul.cpsc.ucalgary.ca.8554:
0x0000       4500 01b1 557f 4000 8006 838a 889f 07dc  E...U.@.........
0x0010       889f 0723 0e28 216a 4b76 f1e3 8a87 2edb  ...#.(!jKv......
0x0020       5018 f9c6 21e1 0000 8501 0000 0509 0001  P...!...........
0x0030       0300 0000 0100 0000 5600 0900 0104 0000  ........V.......
0x0040       0500 0000 0070 0068 006f 006e 0065 0008  .....p.h.o.n.e..
0x0050       000e 0000 0028 0034 0030 0033 0029 0020  .....(.4.0.3.)..
0x0060       0036 0032 0030 002d 0030 0035 0038 0030  .6.2.0.-.0.5.8.0
0x0070       0008 0000 0068 006f 006d 0065 0070 0061  .....h.o.m.e.p.a
0x0080       0067 0065 0008 0000 0000 0005 0000 0065  .g.e...........e
0x0090       006d 0061 0069 006c 0008 0017 0000 0062  .m.a.i.l.......b
0x00a0       006f 0079 006c 0065 006d 0040 0063 0070  .o.y.l.e.m.@.c.p
0x00b0       0073 0063 002e 0075 0063 0061 006c 0067  .s.c...u.c.a.l.g
0x00c0       0061 0072 0079 002e 0063 0061 0004 0000  .a.r.y...c.a....
0x00d0       006e 0061 006d 0065 0008 000a 0000 004d  .n.a.m.e.......M
0x00e0       0069 006b 0065 0020 0042 006f 0079 006c  .i.k.e...B.o.y.l
0x00f0       0065 0001 0000 0049 0008 0026 0000 007b  .e.....I...&...{
0x0100       0032 0064 0066 0033 0039 0037 0038 0039  .2.d.f.3.9.7.8.9
0x0110       002d 0038 0064 0032 0031 002d 0034 0037  .-.8.d.2.1.-.4.7
0x0120       0063 0030 002d 0039 0062 0030 0063 002d  .c.0.-.9.b.0.c.-
0x0130       0061 0066 0032 0036 0039 0032 0039 0036  .a.f.2.6.9.2.9.6
0x0140       0031 0062 0061 0061 007d 0001 0000 004b  .1.b.a.a.}.....K
```

```
0x0150        0008 002d 0000 002f 0075 0073 0065 0072  ...-.../.u.s.e.r
0x0160        0073 002f 007b 0032 0064 0066 0033 0039  .s./.{.2.d.f.3.9
0x0170        0037 0038 0039 002d 0038 0064 0032 0031  .7.8.9.-.8.d.2.1
0x0180        002d 0034 0037 0063 0030 002d 0039 0062  .-.4.7.c.0.-.9.b
0x0190        0030 0063 002d 0061 0066 0032 0036 0039  .0.c.-.a.f.2.6.9
0x01a0        0032 0039 0036 0031 0062 0061 0061 007d  .2.9.6.1.b.a.a.}
0x01b0        00                                       .
```

Message: 0x00000185 bytes type = 0x05 (PUT), serialized Struct contains 3 items

Struct Item Description #1:

Name: "V" (one character long)

Value: object/Struct, inner struct contains four items

Inner Struct Item Description #1:

Name: "phone" (0x00000005 chars long)

Value:string, "(403) 620-0580" (0x0000000e chars long)

Inner Struct Item Description #2:

Name: "homepage" (0x00000008 chars long)

Value:string "" (0x00000000 chars long)

Inner Struct Item Description #3:

Name: "email" (0x00000005 chars long)

Value:string boylem@cpsc.ucalgary.ca (0x00000017 chars long)

Inner Struct Item Descrition #4:

Name: "name" (0x00000004 chars long)

Value:string "Mike Boyle" (0x0000000a chars long)

Struct Item Description #2:

Name: "I" (0x00000001 chars long)

Value: string "{2d4e9789-8d21-47c0-9b0c-af2692961baa}" (0x00000026 chars long)

Struct Item Description #3:

Name "K" (0x00000001 chars long)

Value: string "/users/{2d4e9789-8d21-47c0-9b0c-af2692961baa}" (0x0000002d chars long)

As we can see, in this packet, the client is requesting to store a Struct object at the key /users/{…}. The Struct has four strings values in it: name, email, homepage, and phone.

8.1.4 Packet #4: PUT reply

Actually, there's no need to show you this packet as it is byte-for-byte identical to the contents of the previous packet. I think by now you've got the basic idea of the wire protocol.