memcacheDB connector in node.js

Release Notes

Document Version 1.2

Abstract

This Functional Specification gives detailed information about a customer-specific implementation of a node.js-based connector to a cluster of memcacheDB servers.

Contact Information

If the information you need is not in this document, you can contact the author:

Email: assen.totin@gmail.com

Document Version History

Version	Changed by	Date	Description	Checked by Accepted by
1.0	Assen Totin	03/29/13	Initial Release	
1.1	Assen Totin	04/01/13	Added test cases, ch. 9.	
1.2	Assen Totin	04/02/13	Separated Release Notes from Functional Specification	

Table of Contents

1. About This Document

This document describes the installation and configuration of a node.js-based connector to a cluster of memcacheDB servers.

1.1 Audience

This document is intended for technical personnel. Because the descriptions of the conversions are low level, it is recommended that the reader should be familiar with node.js and memcacheDB which is being extended.

1.2 Typographic conventions

The following text styles identify special information used in the document:

Bold: bold text is used to call attention.

Italics: Italicised text is used to emphasize the specific meaning of the words.

`Fixed-width`: Fixed-width font is used for presenting user input.

Note: Notes are written between two lines to point to important issues.

1.3 Terms and concepts

The following abbreviations, terms and concepts are used in the document:

1.3.1 Abbreviations

IETF	Internet Engineering Task Force. IETF is the main technical authority regarding technical standards on the Internet. See www.ietf.org for details.
RFC	Request for Comments – the formal name for most Internet standards as accepted by IETF.

1.3.2 Terminology

DEFLATE	Losless data compression algorithm as specified in RFC 1951. Implemented in popular utilities like *compess* and *gzip* for Unix, *PKZIP* for MS-DOS, *WinZip* for Micsofot Windows etc.
memcacheDB	An persistent data storage which keeps the data in a hash (key-value pairs), capable of multi-server (clustered) networked operation. See www.memcachedb.org for details.
node.js	An event-driven framework for building scalable networked applications. See www.nodejs.org for details.
tape archive format	A format for archiving multiple files in a single container using concatenation. Implemented in the *tar* Unix utility.

1.4 Related documentation

- node.js Connector Functional Specification

2. Code and Licensing

The connector, developed under this project, is delivered free of any license.

The node.js engine is not be modified within this project and will, therefore, retain its original license.

The memcacheDB engine is not be modified within this project and will, therefore, retain its original license.

3. Deliverables

The following will be delivered under this project:

- The source code, developed under the project, archived using *tape archive format* and further compressed using the *DEFLATE* algorithm.
- Functional Specification.
- Release Notes.

4. System Requirements

The project requires:

- A node.js server.
- A cluster of memcacheDB servers.

5. Installation and Configuration

5.1 Prerequisites

Installing the package may require local administrative privileges.

5.2 Installation procedure for the connector

1. Log on to the server which runs the node.js.

2. Obtain local administrative privileges.

3. Uncompress and unarchive the supplied sources.

4. Copy the `lib` directory to a suitable location where it can be included by the applications using it.

5.3 Configuration of applications

5.3.1. Initializing the connector

```
var memdb = require('/path/to/lib/memdb');
```

Replace the path with the actual path to the `lib` directory of the connctor.

```
MemdbClient = new
memdb.MemdbClient(['127.0.0.1:21201','10.10.10.2:21202']);
```

Replace the IP addresses and ports with the actual ones.

Note: If no IP addresses are given, one default host at `127.0.0.1:21201` is used.

Note: Each memcacheDB server should use an unique IP address.

5.3.2. Sending queries

All queries should be sent to the initialized instance. It will take care to automatically distribute the load among servers.

Writing example:

```
MemdbClient.set('some_key', 'some_value', function(err, data) {

          // Some callback function code

}, 0, 0);
```

Reading example:

```
MemdbClient.get('some_key', function(err, data) {

          // Some callback function code

} );
```

For full API list, see the *Functional Specification, Appendix A*.

5.3.3. Closing the connection

```
MemdbClient.close();
```

6. Test Cases

The following test cases have been run on the project.

6.1 Test Suit A

Start one master server (M) and two slave servers (S1, S2).

No	Test Case	Scenario	Status
1	Master and slaves should be properly recognized. All write requests should go to master, all read requests to slaves in round robin.	With M, S1, S2 working: M should get all write requests; read requests should be sent to S1 and S2 in a round-robin fashion.	PASS
2	Skip slaves that have failed.	Stop S2: now all read requests go to S1.	PASS
3	Failed read requests should automatically be forwarded to another slave.	After stopping the S2, the failed read request goes to S1.	PASS
4	If all slaves are gone, the master gets both read and write requests.	Stop S1: now all read requests go to M.	PASS
5	Periodically check the slaves list and use any new slave that has come online.	Bring back up S1 and S2. After the periodic check, the read requests will be transferred from M to S1 and S2.	PASS

6.1 Test Suit B

Start one master server (M) and two slave servers (S1, S2).

No	Test Case	Scenario	Status
1	Get new master from slaves when old master have failed.	Stop M. S1 and S2 will elect a new master, Mn. Upon next write request, Mn will be detected as master. All write requests should go to Mn and all read requests to the remaining slave.	PASS
2	Failed read requests should automatically be forwarded to the new master.	After stopping the M, the failed read request goes to Mn.	PASS

6.3 Test Suit C

Run one master server (M) and one slave server (S).

No	Test Case	Scenario	Status
1	Spawn new connections on demand (when there are no enough connections to handle requests for a given slave).	Quickly (ina loop) send 10 read requests which all go to S. Since 10 is more that the default number of connections (3), 7 new connections will be spawned.	PASS
2	Clean-up unnecessary connections when they are not in use.	After the check for excessive connections runs, it will detect that: * the number of connections (10) is more than the threshold (2 times the minimum number of connections or 6) , and * the number of idle connections is more that the number of busy connections, so ½ of the unused connections will be shut down.	PASS