memcacheDB connector in node.js

Function Specification

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 5
1.1 Audience 5
1.2 Typographic conventions 5
1.3 Terms and concepts 5
1.3.1 Abbreviations 5
1.3.2 Terminology 5
1.4 Related documentation 6
2. Scope of Work 7
3. Main Functions 8
3.1 Write requests 8
3.2 Read requests 8
3.3 Concurrency 9
4. Effort estimate 10
Appendix A: API 11

1. About This Document

This document describes the functionality 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

None used.

1.3.2 Terminology

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.

1.4 Related documentation

No related documentation is required.

2. Scope of Work

The project consist of:

- Creating a connector (library) which runs on top of a node.js engine and helps connects to a cluster of memcacheDB servers.

3. Main Functions

The main function of the memcacheDB connector is to help a node.js application exchange data with a cluster of memcacheDB servers.

3.1 Write requests

The memcacheDB connector will automatically determine which member of the cluster is a master server and will send all write requests only to it.

If the current master changes or becomes unavailable, the connector will query who is the new master (as elected by memcacheDB cluster) and will begin using it for write queries. A failed write request will be resent to the new master.

3.2 Read requests

If more than one slave member is available, requests will be distributed between them in a round-robin fashion.

If there is only one slave, it will receive all read requests.

If there is only one member left in the cluster (stand-alone master), it will also receive the read requests (along to write requests).

If a slave member becomes master, it will not be used for read queries (as long as there are other slave members).

If a slave member becomes unavailable, the connector will stop sending requests to it. A failed read request will be resent to another available slave (or to the stand-alone master).

The connector will check periodically the slave list and if a slave has become available

again and after such check succeeds, requests will once again be sent to this member.

> Note: This allows the administrator to add slaves which have not been provisioned to the connector when it started. However, because the connector will not know the TCP port of the new slave, it must operate on the default memcaheDB port of 21201.

3.3 Concurrency

When initialized, the connector will launch several concurrent worker connections to each server. This connections will be reused when communicating with the servers. If there is pending data, but all connections to the given server are used, the connector will spawn a new worker for this server.

The connector will periodically monitor the number of workers for each server and will reduce them if there are too many unused workers. The reduction rule is as follows:

- If the total number of workers is more than twice the minimum number of workers, and
- If there are more idle workers than busy workers,
- Then remove ½ of the idle workers.

4. Effort estimate

Implementation of the project will require the following resources:

Task	**Estimate, man-hours**
General	
Documentation: Functional specification	4
Documentation: Release notes	4
Programming	
Generic memcacheDB client	8
Master/slaves detection	4
Slaves round-robin load balancing	4
Per-server connection pool management	4
Failed slave detection	2
Failed slave periodical check	4
Testing and QA	
Generic memcacheDB client	4
Master/slaves detection	2
Slaves round-robin load balancing	2
Per-server connection pool management	1
Failed slave detection	1
Failed slave periodical check	2
Packaging and Delivery	
Packaging	1
Delivery	1
TOTAL:	**48**

Appendix A: API

The following public methods are described below. For limitations regarding allowed values and sized of memcacheDB-related parameters, refer to memcacheDB documentation.

```
new(['A.B.C.D:E', …], {'param':value})
```

Description: Creates a new connector.
Arguments:

- Array of strings. Each member is a server IP address and port, e.g. 192.168.0.1:21201. Mandatory.
- Object where keys are internal parameters and values are their desired values:
    - default_host: host to use if first argument is empty (default: localhost)
    - default_port: port if first argument is empty (default: 21201)
    - default_workers: how many workers to spawn by default for each server (default: 3)
    - default_interval_cleanup_workers: how often to clean up unneeded workers (time in ms, default is 3600000, i.e.1 hour)
    - default_interval_check_slaves: how often to check for changes in slaves (time in ms, default is 120000, i.e. 10 minutes)
    - debug: whether to display debugging information (default is 0)

```
.get(key, callback(err, data))
```

Description: Fetches data by the specified key. The data is sent to the provided callback function.
Arguments:

- key: ASCII string to be used as memcacheDB lookup key. Mandatory.

- callback: function to be called when the data is available. Mandatory. Arguments:
  - err: null if no error occurred, error identity otherwise
  - data: the data, received from the memcacheDB

```
.set(key, value, callback(err, data), lifetime, flags)
```

Description: Sets the specified key ro the specified value. The data is sent to the provided callback function.

Arguments:

- key: ASCII string to be used as memcacheDB lookup key. Mandatory.
- value: the value to be stored in memcacheDB. Mandatory.
- callback: function to be called when the data is available. Optional. Arguments:
  - err: null if no error occurred, error identity otherwise
  - data: the data, received from the memcacheDB
- lifetime: the lifetime of the record. Will be ignored by the memcacheDB. Optional.
- flasg: flags for record. Optional.

```
.add(key, value, callback(err, data), lifetime, flags)
```

Description: Sets the specified key to the specified value if it does not already exist.

Arguments: same as for .set()

```
.replace(key, value, callback(err, data), lifetime, flags)
```

Description: Sets the specified key to the specified value if it already exists.

Arguments: same as for .set()

```
.append(key, value, callback(err, data), lifetime, flags)
```

Description: Appends the specified value to the value of specified key.

Arguments: same as for .set()

```
.prepend(key, value, callback(err, data), lifetime, flags)
```

Description: Prepends the specified value to the value of specified key.

Arguments: same as for .set()

```
.cas(key, value, unique, callback(err, data), lifetime, flags)
```

Description: Checks and sets the specified key to the specified value.

Arguments: same as for .set(), plus:

- unique: unique key to use for checking.

```
.delete(key, callback(err, data))
```

Description: Deletes the key and its value.

Arguments:

- key: ASCII string to be used as memcacheDB lookup key. Mandatory.
- callback: function to be called when the operation completes. Optional. Arguments:
    - err: null if no error occurred, error identity otherwise
    - data: the data, received from the memcacheDB

```
.version(callback(err, data))
```

Description: returns the version of the memcacheDB server.

Arguments:

- callback: function to be called when the operation completes. Mandatory. Arguments:
    - err: null if no error occurred, error identity otherwise
    - data: the data, received from the memcacheDB

```
.increment(key, value, callback(err, data))
```

Description: Increments the value, specified by the key, with the provided value.

Arguments:

- key: ASCII string to be used as memcacheDB lookup key. Mandatory.
- Value: the value to use for increment.
- callback: function to be called when the operation completes. Optional. Arguments:
  - err: null if no error occurred, error identity otherwise
  - data: the data, received from the memcacheDB

```
.decrement(key, value, callback(err, data))
```

Description: Decrements the value, specified by the key, with the provided value.

Arguments: same as for .increment()

```
.stats(type, callback(err, data))
```

Description: Increments the value, specified by the key, with the provided value.

Arguments:

- type: ASCII string, the name o fthe type of statistics to be retrieved from the memcacheDB lookup key. Mandatory.
- callback: function to be called when the operation completes. Mandatory. Arguments:
  - err: null if no error occurred, error identity otherwise
  - data: the data, received from the memcacheDB

```
.close()
```

Description: Closes the connector.

Arguments: none