INFO-H-415: Advanced databases
Project Voldemort

Louvet Florian
Gérard François

December 31, 2013

Contents

1 Introduction **3**

2 Voldemort **4**
2.1 How to deal with a big amount of data? . . . 4
2.2 The awareness of network state . . . 5
2.3 Queries used in Voldemort . . . 5
2.4 Versioning . . . 6
2.5 Advantages and disadvantages of Voldemort . . . 7

3 Application **8**
3.1 Server configuration . . . 8
3.1.1 cluster.xml . . . 8
3.1.2 stores.xml . . . 9
3.1.3 server.properties . . . 9
3.2 Code of the application . . . 10
3.2.1 Main . . . 10
3.2.2 Connect . . . 10
3.2.3 Mod_Loging and Mod_request . . . 10
3.2.4 SERVER_CONFIG . . . 10
3.2.5 Test . . . 10
3.2.6 Test_method . . . 11
3.2.7 Cat . . . 11
3.2.8 DAO . . . 11
3.3 Stores . . . 11
3.3.1 Login store . . . 11
3.3.2 Cat store . . . 11
3.3.3 Owner store . . . 11
3.3.4 Comments store . . . 12
3.4 Context-driven testing . . . 12
3.4.1 Obsolete version test (Test 1) . . . 12
3.4.2 Raw versioning and consistency (Test 2) . . . 12
3.4.3 Server availability . . . 13
3.4.4 Other Tests . . . 13
3.4.5 Tests Data . . . 14

4 Conclusion **15**

A Appendix **17**
A.1 cluster.xml . . . 17
A.2 stores.xml . . . 20

1 Introduction

The purpose of this project is to present a database technology and to implement a straightforward application using this technology as a database management system to show its capability in a real-life environment.
We have opted for a NoSQL(Not only SQL) distributed key-value storage system called *Voldemort* mainly know to be used by the social network *LinkedIn* and based on *Amazon's Dynamo* system.
In this report, we will explain how *Voldemort* is working, what are its advantages and disadvantages, what application we decided to implement, how we are using *Voldemort* in the application and present our test results.

2 Voldemort

In this section, the functional operation of *Voldemort* will be explained.
First, we will see how to deal with scaling, querying and versioning problems.
Then we will go quickly through the pros and cons of using this type of database.

2.1 How to deal with a big amount of data?

One of the biggest issues in data management nowadays is to face the huge amount of data that must be stored in every system. For instance, every company is maintaining plenty of information about its customers and employees to make statistics or to have a global view of the market.
Voldemort's solution to the scaling problem is to distribute the data among multiple servers. Each server cannot store all data otherwise it would quickly run out of memory. However, we should still be able to access information from anywhere, therefore data will be split and replicated to multiple nodes.
The data is partitioned with a consistent hashing algorithm. For S servers, a value n (greater than S) is chosen and each server will handle n/S partitions distributed with an arbitrary hash function.
One of the advantages of this method is that the number of cache hit is increased since each server has its own cache and can retain the data which is often locally requested instead of being dependent on the global cache of a centralized database.

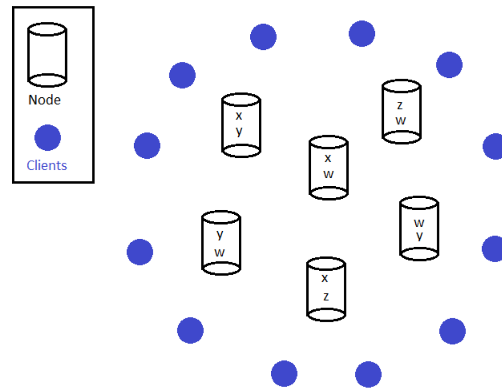

Figure 1: Distributed database

2.2 The awareness of network state

The *Voldemort* cluster does not contain any master node, this means that there is no server managing the network's state and notifying the cluster when something goes down or is running slowly.
Thus, each node is responsible to inform the others if something does not work in its functional operation. Because the standard deviation of the time needed to process a *Voldemort* query (put, get, delete) is very low, it can be used as a "ping request" to provide node status. This means that we can easily calculate a service level requirement in terms of number of queries handled per unit of time.
If a node does not meet the requirement, it is considered to be down and will be temporaly banned so no new access to this node data is available. Because the data is duplicated on multiple nodes, it does not severely affect the end user of the database. After a given period of time, if the node is fixed, it will be active in the cluster.

2.3 Queries used in Voldemort

Since *Voldemort* is a NoSQL key-value storage database, the number of queries and their complexity is really low. To store the data, *Voldemort* uses *Stores* which are a bit the equivalent of tables in relational databases, each store is a hashmap containing some tuples (Key,Value). (*Figure 2*)
Here are the three queries used by *Voldemort*:

- aStore.GET(K) returns the value associated with the key K in the store *aStore*
- aStore.PUT(K, V) inserts the Key/Value tuple (K, V) in the store *aStore*.
- aStore.DELETE(K) deletes the tuple associated with the key K in the store *aStore*.

In order to make more complex queries, like "joins", the operations must be made in the code (client side). However, the value stored in the tuples can be more complex data like a serialized object or a JSON object which allow us to fetch big objects in one GET request, thus, the complexity is on the edge of the network and not inside the database.

Clients	
Key	Value
1	Norris Chuck
2	Jackson Michael
3	Jedusor Tom
4	Chen Peter
5	'); DROP TABLE Clients; --
...	...

Figure 2: A store containing data about Clients

2.4 Versioning

Voldemort works with multiple servers that can be accessed by multiple clients at the same time and potentiqlly accessing the same information. Thus keeping the consistency of the data is a real challenge in a distributed system.

The solution to this challenge utilized by *Voldemort* is to use a well know algorithm in distributed systems called Vector clocks. A vector is associated with each data, it contains tuples (S,V) linking each server S to a value V representing the version of the data from the server S point of view. (*Figure 3*)

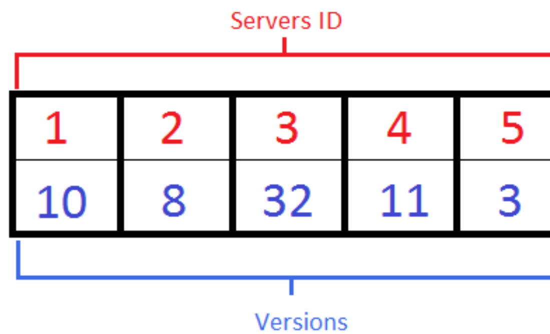

Figure 3: Vector clock

The algorithm is working as follow:

- Set all the clocks to zero in all vector clocks (VC)
- If data is modified on the server S, it increments VC[S] then broadcasts the data and the associated vector.
- If data is received from another server, the current clock vector (VC) is compared with the received clock vector (RVC). If $\forall S, RVC[S] \geq VC[S]$, the data is updated. Otherwise, there is a conflict.

2.5 Advantages and disadvantages of Voldemort

Advantages:

- Key-value storage keeps the system simple.
- The simplicity of the system makes it fast.
- Possible to store complex items (Serialization, JSON).
- Easy to configure (XML files).
- Simple queries (easy to evaluate the time of the requests).
- Good potential of scaling when the amount of data is increasing.
- Can be used in parallel with a SQL database.
- Clients for multiple languages (Java, Python, Ruby, C++)

Disadvantages:

- Impossible to do complex operations in a single request
- In some cases, increases code complexity (e.g., more complex Data Access Objects)
- Impossible to use triggers
- Simulating classic SQL queries may require additional stores.

3 Application

In order to illustrate the use of *Voldemort* in applications, we decided to implement a small social network called Catwalk in Python. Catwalk is supposed to be a network used by "show cat" owners in order to promote them to the "show cat" community. This social network has several features: *General information, Add/Modify/Delete a cat from the social network, Write a comment about a cat.*

Distributed key/value storages are particularly used for unstructured data and data which is already associated to a unique id (e.g., user information, user sessions, user profiles, user preferences, shopping carts, etc.). Considering this, we opted for a basic "social network" application.

3.1 Server configuration

First step in the implementation of this application was the configuration of the server cluster that had to host our distributed key/value storage. *Voldemort* servers are configured via three files:

- cluster.xml
- store.xml
- server.properties

3.1.1 cluster.xml

You may find a copy of our configuration for cluster.xml in appendix. (A.1)
The file cluster.xml defines the configuration of the server cluster on which *Voldemort* will store data. Our cluster is divided in six zones which represents geographical areas (Europe, Asia, North America, South America, Africa and Oceania). Each zone is characterized by a list representing the proximity of the other zones. It represents a "data center" composed of two servers called nodes. Under the circumstances of our tests, all hosts have been set to localhost(127.0.0.1); different http and socket ports have been attributed to each node. Each node is divided into a number of partitions which are referenced within the *Voldemort* documentation as:

> Partitions are not static partitions of servers, but rather they are a mechanism for partitioning the key space in such a way that each key is statically mapped to a particular data partition.

The number of partitions defined within a cluster is constant, which means that any addition of a node results in a rebalancing of the already existing partitions. It is important to choose a substantial number of partitions in order to allow a future enlargement of the cluster.

3.1.2 stores.xml

You may find a copy of our configuration for stores.xml in appendix. (A.2)
The file stores.xml defines the configuration of the different stores that will contain data. A store could be considered as the equivalent of a relational table for a distributed key/value storage. It is basically a gigantic Hashtable making value correspond to a key. Mandatory and optional settings are: name, replication-factor, preferred-reads, required-reads, preferred-writes, required-writes, persistence, routing, routing-strategy, key-serializer, value-serializer, retention-days, retention-scan-throttle-rate[1].

- replication-factor (N) - The number of replicas that will be stored through the cluster.
- required-reads(R) - The number of machines to read from in order to judge the query successful.
- required-writes(W) - The number of writes needed in order to judge the query successful.

We decided to use JSON serialization to save "complex" data in the stores for three reasons. First, it is one of the few serializations natively supported by *Voldemort*. Second, it is equivalent to python dictionaries which can be easily manipulated (e.g., python objects attributes are stored into a dictionary). Finally, JSON is a prevalent format used in applications (e.g., used for data transfer in AJAX and other web-services) and as of this date it is supported by 55 programming languages.

3.1.3 server.properties

This configuration file contains some properties that are unique to the server. It contains the "node id", and (if it is the type of persistence chosen) some BDB configuration parameters.

[1]For each of those parameters a definition can be found in *Voldemort*'s documentation.

3.2 Code of the application

In this section we will describe the modules used by the application.
Since the purpose of this project was to illustrate a specific technology of database and not to implement a huge application, the code was kept simple and the features restricted.
Thus, "Catwalk" offers no graphical user interface and the interaction with users is made through keyboard input.

3.2.1 Main

The main module is the entry point of the application, it is responsible to set up the connections with the database, display the main menu to the user and call the methods of the other modules to satisfy the request of the user.

3.2.2 Connect

The connect module contains a single method *connect(REGION)* which makes a connection to the store of the server corresponding to the region given in parameter. It returns a dictionary containing the stores which will be used in the rest of the application to make queries.

3.2.3 Mod_Loging and Mod_request

These two modules contains the main action that a user can perform in the application.
Mod_login handles all the actions related to the login and the creation of a new user.
Mod_request handles the actions related to the modification and the management of the cats, its functions are called by the main module if the user wants to see a cat, modify it, delete it or add a new one.

3.2.4 SERVER_CONFIG

SERVER_CONFIG contains a large number of constants related to the servers. The connect module accesses the SERVER_CONFIG and retrieves information about the regional server given in parameter such as the hostname/ip address and the port number on the server.

3.2.5 Test

In order to test *Voldemort*'s efficiency in our application, we had to make some tests and to simulate multiple client connections to servers at the same time. Hence, this module is running multiple threads which will perform queries on the cluster and is displaying the result of those tests.

3.2.6 Test_method

This module contains the actions that the threads running in Test must perform to modify the state of the database.

3.2.7 Cat

Cat is the class representing the "show cats". It contains simple attributes related to a real cat, a method to display it to the user in the console, a method to retrieve a JSON-like object of the cat and a method to construct the cat from a JSON-like dictionary.

3.2.8 DAO

This module is implementing a Data Access Object (DAO) pattern well known by the developers of large applications using different databases at the same time. Our DAO is containing methods doing the basic actions on the database such as adding a new user, a new cat, etc. All DAO methods are taking basic python types or one of our classes in parameter to perform the queries, thus, if we decide to change the database at some point, only this part of the application must be modified and the structure (of the rest of the code) could remain the same.

3.3 Stores

We used four stores to keep the data in our application.

3.3.1 Login store

This store contains the data about the users that can claim access to the application, the keys are the names (we can use it as a key since a user's name is unique) of the users and the values are the corresponding password of each user's name.

3.3.2 Cat store

The cat store contains the information related to the cat stored in the database, the keys are formatted on basis of the first and last name of the cat. The value is a JSON object containing the value for each attribute of the Cat class except the comments which are stored somewhere else.

3.3.3 Owner store

The owner store is used to keep track of the cats owned by a user, the keys are the user's names and the values are a list of strings representing the key corresponding to the user's cats in the cat store.

3.3.4 Comments store

The comments store contains the comments associated to the cats. The keys are the same as in the cat store and the value is a JSON object containing the text of the message and the author of the message. When a cat is displayed, a request is made to the comments and cat store with the same key to get all information on the cat.

3.4 Context-driven testing

In addition to the Client application, we decided to implement context-driven tests in order to demonstrate the proper functioning of our program. Those tests consist of simultaneous and various client-requests to the database. To simulate concurrent access we created a module launching multiple threads that run a similar sequence of queries. Each sequence being measured by time and other metrics, if relevant.

3.4.1 Obsolete version test (Test 1)

When simultaneously modifying (put request) data of a similar key within the same store, a VoldemortException of type ObsoleteVersionException might be thrown. This occurs when two different modifications are being applied "at the same time" and would produce a consistency error in the vector clocks.

When it occurs, the put query that would have inserted a consistency error is abandoned and the exception is thrown. In order to prevent loss of data, the *Voldemort* Java client implements a method named applyUpdate which will apply the given action repeatedly until no ObsoleteVersionException is thrown. The python client which is not the native client does not implement it, so we implemented it ourselves based on the Java source code.

3.4.2 Raw versioning and consistency (Test 2)

When clients are simultaneously applying a modification to existing data (e.g., two users adding a comment at the same time to a cat's comment list), data-loss is possible. As explained in the following diagram, if no check is made, the data inserted from the first client (c1) will not contain the comment previously added by the second client (c2). (*Figure 4*)

To prevent those errors, *Voldemort*'s client implements a method called maybe_put(key, data, version), which will only update the data related to a key if its current version is equal to the one given. If vector clocks are not similar, it means that another modification has been made in between the last retrieval and now. If that is the case, the maybe_put method will then return "None", letting the client application decide what to do (e.g., retrieve and insert again with up to date data)

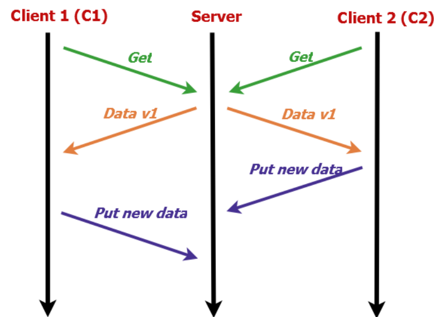

Figure 4: Versioning problem

3.4.3 Server availability

Shutting down a server (Test 3 and 4) Two tests have been made simulating a cluster with an unavailable node. The first one tries to put data in the database, the other one tries to get it. Both of these queries encountered no errors, since required_write and required_read number is reached (R<N-1 and W<N-1). (3.1.2)

Shutting down multiple servers Similar tests to the ones explained in the previous paragraph have been made in this case. If the number of servers brought down is superior to N-R or N-W, the probability of a connection error is high (depending whether or not the data queried is in partitions related to the nodes that are brought down). This event throws a VoldemortException that can be handled by the client application.

3.4.4 Other Tests

Here is a list of other tests that introduce no errors (efficiency tests):

- Simultaneous put on different keys (Test 5)
- Simultaneous get on a similar key (Test 6)
- Simultaneous get on different keys (Test 7)
- Simultaneous deletes on a similar key (Test 8)[2]
- Simultaneous deletes on different similar key (Test 9)

[2]delete query on a non-existing key does not produce any error

3.4.5 Tests Data

Nb.	Operations	Nb. ObsException	Nb. Raw vers	Time
1	2 put -o	80 - 125	-	0.6 seconds
2	1 put -o ; 1 get ; 1 m_put	±55	±14	0.3 seconds
3	2 put	-	-	0.16 seconds
4	1 get ; check	-	-	0.03 seconds
5	2 put	-	-	0.15 seconds
6	1 get ; check	-	-	0.03 seconds
7	1 get ; check	-	-	0.05 seconds
8	1 get ; 1 del	-	-	0.18 seconds
9	1 get ; 1 del	-	-	0.22 seconds

-o : inducing ObsoleteVersionException
ObsException : ObsoleteVersionException
Nb. Raw vers : Nb. of elements not introduced to keep consistency (3.4.2)
check : function checking data
del : delete query

4 Conclusion

We eventually managed to set up a cluster of 12 nodes and to run up to 18 clients using 72 connections sending requests at the same time. The conclusion we have made about this project is that as we expected, the usage of the database after its configuration is very easy since there are only three possible requests. This is really pleasant to avoid building big and complex queries but it requires attention when structuring the stores because *Voldemort* database does not allow much flexibility to retrieve specific data.
The fact that the tuples in the stores can be JSON objects is really convenient because it is possible at some point to simulate a little table or a basic object if needed.
The complexity is really managed in the code, client-side, because of the low amount of request types but the structure of the database is really important and difficult to modify along the way.
While the flexibility of the database is not outstanding, the speed of the request and the large scaling capability are a real advantage in today's databases dealing with a huge amount of data.

References

[1] Kreps, J. (2013)., *Project-Voldemort.* [online] Available at: http://www.project-voldemort.com/voldemort/, [Accessed: 31 Dec 2013].

[2] Kreps, J. (2009)., *Project voldemort (part ii): how it works.* [online] Retrieved from: http://blog.linkedin.com/2009/04/01/project-voldemort-part-ii-how-it-works/, [Accessed: 31 Dec 2013].

[3] Lamport, L. (1978 July)., *Time, clocks, and the ordering of events in a distributed system.* [online] July. Retrieved from: http://research.microsoft.com/en-us/um/people/lamport/pubs/time-clocks.pdf, [Accessed: 31 Dec 2013]

[4] Sumbaly, R., Kreps, J., Gao, L., Feinberg, A., Soman, C. Shah, S. (n.d.)., *Serving large-scale batch computed data with project voldemort.* [online] Retrieved from: https://www.usenix.org/legacy/events/fast12/tech/full$_p apers/Sumbaly.pdf$, $[Accessed: 31 Dec 2013]$

A Appendix

A.1 cluster.xml

```
<cluster>
        <name>Catwalk</name>
        <zone>
              <zone-id>0</zone-id> <!-- Europe -->
              <proximity-list>1, 5, 3, 2, 4</proximity-list>
            </zone>
            <zone>
              <zone-id>1</zone-id> <!-- Afrique -->
              <proximity-list>0, 5, 2, 4, 3</proximity-list>
            </zone>
            <zone>
              <zone-id>2</zone-id> <!-- Am Sud -->
              <proximity-list>3, 5, 0, 4, 1</proximity-list>
            </zone>
            <zone>
              <zone-id>3</zone-id> <!-- Am Nord -->
              <proximity-list>2, 5, 0, 4, 1</proximity-list>
            </zone>
            <zone>
              <zone-id>4</zone-id> <!-- Oceanie -->
              <proximity-list>5, 1, 2, 3, 0</proximity-list>
            </zone>
            <zone>
              <zone-id>5</zone-id> <!-- Asie -->
              <proximity-list>4, 3, 2, 0, 1</proximity-list>
            </zone>
        <server>
            <id>0</id>
            <host>localhost</host>
            <http-port>8081</http-port>
            <socket-port>6666</socket-port>
            <admin-port>6667</admin-port>
            <partitions>0, 1, 2</partitions>
            <zone-id>0</zone-id>
        </server>
        <server>
            <id>1</id>
            <host>localhost</host>
            <http-port>8082</http-port>
            <socket-port>6668</socket-port>
            <admin-port>6669</admin-port>
            <partitions>3, 4, 5</partitions>
```

```
        <zone-id>0</zone-id>
    </server>
    <server>
        <id>2</id>
        <host>localhost</host>
        <http-port>8083</http-port>
        <socket-port>6670</socket-port>
        <admin-port>6671</admin-port>
        <partitions>6, 7, 8</partitions>
        <zone-id>1</zone-id>
    </server>
    <server>
        <id>3</id>
        <host>localhost</host>
        <http-port>8084</http-port>
        <socket-port>6672</socket-port>
        <admin-port>6673</admin-port>
        <partitions>9, 10, 11</partitions>
        <zone-id>1</zone-id>
    </server>
    <server>
        <id>4</id>
        <host>localhost</host>
        <http-port>8085</http-port>
        <socket-port>6674</socket-port>
        <admin-port>6675</admin-port>
        <partitions>12, 13, 14</partitions>
        <zone-id>2</zone-id>
    </server>
    <server>
        <id>5</id>
        <host>localhost</host>
        <http-port>8086</http-port>
        <socket-port>6676</socket-port>
        <admin-port>6677</admin-port>
        <partitions>15, 16, 17</partitions>
        <zone-id>2</zone-id>
    </server>
    <server>
        <id>6</id>
        <host>localhost</host>
        <http-port>8087</http-port>
        <socket-port>6678</socket-port>
        <admin-port>6679</admin-port>
        <partitions>18, 19, 20</partitions>
        <zone-id>3</zone-id>
```

```
</server>
<server>
    <id>7</id>
    <host>localhost</host>
    <http-port>8088</http-port>
    <socket-port>6680</socket-port>
    <admin-port>6681</admin-port>
    <partitions>21, 22, 23</partitions>
    <zone-id>3</zone-id>
</server>
<server>
    <id>8</id>
    <host>localhost</host>
    <http-port>8089</http-port>
    <socket-port>6682</socket-port>
    <admin-port>6683</admin-port>
    <partitions>24, 25, 26</partitions>
    <zone-id>4</zone-id>
</server>
<server>
    <id>9</id>
    <host>localhost</host>
    <http-port>8090</http-port>
    <socket-port>6684</socket-port>
    <admin-port>6685</admin-port>
    <partitions>27, 28, 29</partitions>
    <zone-id>4</zone-id>
</server>
<server>
    <id>10</id>
    <host>localhost</host>
    <http-port>8091</http-port>
    <socket-port>6686</socket-port>
    <admin-port>6687</admin-port>
    <partitions>30, 31, 32</partitions>
    <zone-id>5</zone-id>
</server>
<server>
    <id>11</id>
    <host>localhost</host>
    <http-port>8092</http-port>
    <socket-port>6688</socket-port>
    <admin-port>6689</admin-port>
    <partitions>33, 34, 35</partitions>
    <zone-id>5</zone-id>
</server>
```

```
</cluster>
```

A.2 stores.xml

```
<stores>
  <store>
    <name>test</name>
    <persistence>bdb</persistence>
    <routing>client</routing>
    <replication-factor>1</replication-factor>
    <required-reads>1</required-reads>
    <required-writes>1</required-writes>
    <key-serializer>
      <type>string</type>
    </key-serializer>
    <value-serializer>
      <type>string</type>
    </value-serializer>
  </store>
  <store>
        <name>login</name>
        <persistence>bdb</persistence>
        <routing>client</routing>
        <replication-factor>5</replication-factor>
        <preferred-writes>3</preferred-writes>
        <required-writes>2</required-writes>
        <preferred-reads>3</preferred-reads>
        <required-reads>2</required-reads>
        <key-serializer>
                <type>string</type>
                <schema-info>utf8</schema-info>
        </key-serializer>
        <value-serializer>
                <type>string</type>
                <schema-info>utf8</schema-info>
        </value-serializer>
  </store>
  <store>
        <name>cat_data</name>
        <persistence>bdb</persistence>
        <routing>client</routing>
        <replication-factor>5</replication-factor>
        <preferred-writes>3</preferred-writes>
        <required-writes>2</required-writes>
        <preferred-reads>3</preferred-reads>
        <required-reads>2</required-reads>
```

```
        <key-serializer>
                <type>string</type>
                <schema-info>utf8</schema-info>
        </key-serializer>
        <value-serializer>
                <type>json</type>
                <schema-info version="1">{"owner":"string",
                    "lastname":"string", "firstname":"string",
                    "color":"string", "breed":"string",
                    "weight":"string", "size":"string",
                    "distraction":["string"]}</schema-info>
        </value-serializer>
</store>
<store>
        <name>comments</name>
        <persistence>bdb</persistence>
        <routing>client</routing>
        <replication-factor>5</replication-factor>
        <preferred-writes>3</preferred-writes>
        <required-writes>2</required-writes>
        <preferred-reads>3</preferred-reads>
        <required-reads>2</required-reads>
        <key-serializer>
                <type>string</type>
                <schema-info>utf8</schema-info>
        </key-serializer>
        <value-serializer>
                <type>json</type>
                <schema-info version="1">[{"text":"string",
                    "author":"string"}]</schema-info>
        </value-serializer>
</store>
<store>
    <name>owners</name>
    <persistence>bdb</persistence>
    <routing>client</routing>
    <replication-factor>5</replication-factor>
    <preferred-writes>3</preferred-writes>
    <required-writes>2</required-writes>
    <preferred-reads>3</preferred-reads>
    <required-reads>2</required-reads>
    <key-serializer>
      <type>string</type>
      <schema-info>utf8</schema-info>
    </key-serializer>
    <value-serializer>
```

```
      <type>json</type>
      <schema-info version="1">["string"]</schema-info>
    </value-serializer>
  </store>

</stores>
```