Parallel and Distributed Computation - 2nd Semester
Proj. 2: Distributed and Partitioned Key-Value Store

Deadline: 27-05-2022

---

1. Overview

In the second project you will develop a distributed key-value persistent store for a large cluster.

A key-value store is a simple storage system that stores arbitrary data objects, the **values**, each of which is accessed by means of a **key**, very much like in a hash table. To ensure persistency, the data items and their keys must be stored in persistent storage, e.g. a hard disk drive (HDD) or a solid state disk (SSD), rather than in RAM.

By **distributed**, we mean that the data items in the key-value store are **partitioned** among different cluster nodes.

Our design is loosely based on Amazon's Dynamo, in that it uses **consistent-hashing** to partition the key-value pairs among the different nodes. This will be described later, but we recommend that you read the paper, as it may give you ideas to solve some of the challenges you will find.

The service is expected be able to handle concurrent requests and to tolerate:

1. node crashes, and
2. message loss.

In addition to implement the data-store nodes, you are expected to implement a test client, also specified below.

1.1 Document organization

This specification has several sections:

Section 2: Service
: Describes the service and summarizes the interface must provide for a testing client.

Section 3: Membership Service
: Specifies the cluster membership service and respective protocol that must be provided by every cluster node.s

Section 4: Key-Value Store
: Specifies the service for storing key-value pairs in a distributed and partitioned way.

Section 5: Replication
: Discusses the use of replication to increase availability. You may wish to implement replication to increase your project's grade.

Section 6: Fault-Tolerance
: Discusses some failure scenarios that you may wish to address to increase your project's grade.

Section 7: Test-Client
: Specifies a client for testing your code.

Section 8: Implementation Aspects
: Discusses some points that are relevant to the implementation.

Section 9: Final Considerations
: Among other topics, specifies what you must submit and how, and describes the grading criteria.

Appendix A: Generic Char-Based Message Syntax
: Describes a generic char-based message syntax that you can use to specify the syntax of the messages of the different protocols.

2. Service

Each node must provides two interfaces.

2.1 Key-Value Store Interface

A key-value persistent store provides essentially three operations:

`put(key, value)`
: which adds a key-value pair to the store

`get(key)`
: which retrieves the value bound to a key

`delete(key)`
: which deletes a key-value pair

2.2 Cluster Membership Interface

This interface allows to add/remove nodes to the cluster:

`join()`
: which adds a node to the cluster

leave()
: which removes a node from the cluster

2.3 Service Invocation

A service node should be invoked as follows.

```
$ java Store <IP_mcast_addr> <IP_mcast_port> <node_id>  <Store_port>
```

where:

<IP_mcast_addr>
: is the address of the IP multicast group used by the membership service

<IP_mcast_port>
: is the port number of the IP multicast group used by the membership service

<node_id>
: This is the node's id and it must be unique in a cluster. Ideally, it would be the IP address, but we are not sure if you can use multiple loopback IP addresses in Windows. (This would allow you to test your project by running multiple nodes on the same host, each of which with its own loopback IP address.)

<Store_port>
: is the port number used by the storage service

3. Membership Service

One of the main issues in a key-value distributed store is to find the node responsible for a key, i.e. where the key-value pair is or should be stored.

The algorithm that your storage system shall implement requires every node to know every other node in the cluster. To keep this information, the cluster nodes run a distributed membership service and protocol.

The operations supported by the membership service are described in Section 2.2. These operations are triggered by the cluster administrator.

The membership protocol strives to keep the membership information at the nodes up-to-date, even in the presence of node crashes.

3.1 Membership Protocol

Every time a node joins or leaves the cluster, the node must send via **IP multicast** a `JOIN/LEAVE` message, respectively. These messages must include the value of a **membership counter**, which is initially 0, when the node joins for the first time, and that is increased by one every time the node leaves or joins the cluster. Thus, an even value counter means that the node is joining the cluster, whereas an odd value counter means that the node is leaving the cluster. The membership counter must survive node crashes and therefore should be stored in non-volatile memory.

Upon receiving a `JOIN/LEAVE` message, a cluster node updates its view of the cluster membership, and adds the respective event (either join or leave) to a **membership log**. Each record in this log includes only the node's id and the value of the membership counter.

A node joining the cluster must initialize the cluster membership. To perform this initialization, some of the cluster members will send the new member a `MEMBERSHIP` message, including its view of the membership, i.e. a list of the current cluster members, as well as the most recent 32 membership events in its log. This transfer of membership information must be done using TCP. Before sending the `JOIN` message, the new member starts accepting connections on a port whose number it sends in its `JOIN` message. Upon receiving the `JOIN` message a cluster member waits for a random time length, after which it sends the membership information. To prevent the new member from being flooded with `MEMBERSHIP` messages, it stops accepting connections after receiving 3 of them.

In order to avoid the propagation of stale information, these `MEMBERSHIP` messages should be sent by nodes whose membership information is up-to-date. (It is up to you to determine how this can be done with high probability.)

If a node joining the cluster does not receive the `MEMBERSHIP` message from 3 other nodes, it should retransmit the `JOIN` message, up to a total, i.e. including the initial message, of 3 times. Nodes that have already replied wiht a `MEMBERSHIP` message successfully, should not resend it, unless there was another membership change meanwhile.

Because of failures, nodes may miss membership change messages. Therefore, every so often, say 1 second, one of the cluster nodes must multicast a `MEMBERSHIP` message with the most recent 32 membership events in its log. Upon receiving such a message, a node updates its membership event log as well as its cluster membership.

Again, the membership protocol should strive to prevent nodes with stale membership information from multicasting these `MEMBERSHIP` (with high probability.)

The membership log does not need to keep more than one event per node: that node's event with the largest membership counter. Compaction of the log, i.e. removal of redundant events, is useful to reduce the amount of disk storage. But even if nodes do not compact the log in non-volatile storage, they should not include redundant events in the messages of the membership protocol.

Note that above we did not describe how a node determines the cluster membership from the protocol messages. It is up to you to decide. We expect you to document this aspect of your design in the report.

A simple multi-threaded implementation of the membership protocol is worth up to 30%.

Message format

You can specify the message format that you see fit. However, all messages should be char-based, i.e. they should be human readable without further processing. The Appendix specifies a generic message format that we used in one project in another course. You can use it, or adapt it, if you wish.

4. Key-value Store

The key-value store is implemented as a distributed partitioned hash table in which each cluster node stores the key-value pairs in a bucket.

The key-value store shall use SHA-256 to generate keys. Therefore, you can assume that there are no hash collisions.

To partition the key-value pairs among the nodes in the cluster, the key-store uses **consistent hashing**. Consistent hashing is a hashing technique that allows to resize a hash table, i.e. change the number of buckets, without remapping all the keys in the table. This is important to the efficiency of the cluster reorganization upon node joining/leaving.

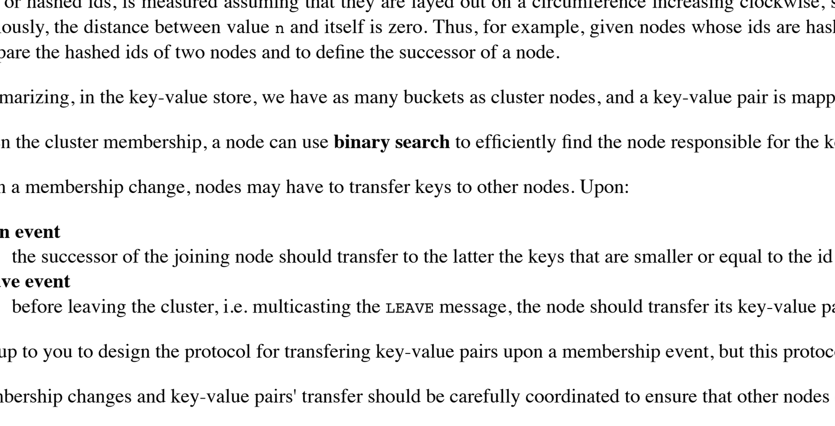

Fig. 1- Consistent hashing.

The idea is illustrated in Fig. 1. The hash function used to compute the keys is also used to hash the id of each node in the cluster. A key-value pair is assigned to the node whose **hashed id** is **closest** to the pair's key. The distance between hash values, either keys or hashed ids, is measured assuming that they are layed out on a circumference increasing clockwise, so that hash value `0` follows hash value $2^H - 1$, where `H` is the number of bits of the hash value. Furthermore, the distance is measured clockwise. Obviously, the distance between value `n` and itself is zero. Thus, for example, given nodes whose ids are hashed to `n-1` and `n+2`, modulo 2^H, the node closest to key `n`, also known as `n`'s **successor**, is the node whose hashed id is `n+2`. We use the same rule to compare the hashed ids of two nodes and to define the successor of a node.

Summarizing, in the key-value store, we have as many buckets as cluster nodes, and a key-value pair is mapped to the bucket of the node whose hashed id is the successor of the pair's key.

Given the cluster membership, a node can use **binary search** to efficiently find the node responsible for the key. It should then send the operation request (`put`, `get` or `delete`) to that node, using TCP.

Upon a membership change, nodes may have to transfer keys to other nodes. Upon:

a join event
: the successor of the joining node should transfer to the latter the keys that are smaller or equal to the id of the joining node;

a leave event
: before leaving the cluster, i.e. multicasting the `LEAVE` message, the node should transfer its key-value pairs to its successor.

It is up to you to design the protocol for transfering key-value pairs upon a membership event, but this protocol should use TCP. We expect you to document this protocol in the project's report.

Membership changes and key-value pairs' transfer should be carefully coordinated to ensure that other nodes can find the key-value pairs. (See the discussion below under fault-tolerance.)

A simple multithreaded implementation of the storage service described in this section up to this point is worth up to 30%.

Enhancement

For key-value stores with many keys or large objects, the transfer of the key-value pairs from one node to another may take a lot of time. Delaying the handling of requests on the keys of these pairs while the transfer is ongoing, may lead to extended periods of unavailability.

Addressing this issue will give you extra credit (up to 5%). Note however, that this enhancement makes sense only if you do not implement replication. You are expected to document your implementation in the project's report.

Warning: Your solution must avoid race-conditions.

5. Replication

The above description does not consider replication. Thus, if the node that stores a key-value pair goes down, that key-value pair becomes unavailable.

To increase availability, key-value pairs should be replicated with a replication factor of 3, i.e. each key-value pair should be stored in 3 different cluster nodes.

The actual number of copies created upon a `put` can be smaller than 3, e.g. because a node is down. Your implementation should strive to ensure that the number of copies is equal to the replication factor as soon as possible after healing of the fault.

Replication may lead to the execution of operations in different order in different replicas. E.g. a replica may process a `delete` operation on a key before the respective `put`. Another issue is the possibility of a node missing a `delete` operation and later try to replicate the deleted key-pair on the other replicas, upon realizing that the number of copies of the pair is lower than the replication factor. A common solution to handle this issue is the use of **"tombstones"** for deletion. A "tombstone" is a special key-pair, or marker, indicating that a pair with the corresponding key has been deleted. Thus, instead of removing the key-value pair upon its deletion, you should replace it with its "tombstone".

Handling replication, is worth up to 20% of your final grade.

6. Fault-Tolerance

In the specification of the protocols we have strived to cover many common failure scenarios, given our failure model. However, we do not cover all the scenarios.

For example, if a node is down for a long time and it misses many membership events, the periodic `MEMBERSHIP` messages may not be enough for the node to learn the current membership of the cluster.

Another scenario is the possibility of the membership view of the node to which the client sent a request not being up-to-date and, therefore, the operation request will be sent to the wrong node, i.e. a node that is not responsible for the key.

Making your store tolerant to these failure scenarios or others that you may find, will give you additional credit up to 10%.

7. Test Client

To test your key-value store you shall develop a test client. Essentially, this test client will let you invoke any of the membership events (`join` or `leave`) as well as to invoke any of the operations on key-value pairs (`put`, `get` and `delete`).

As mentioned above, every node should provide these two interfaces. Any node may process a request for any key. Thus a client needs only know one of the cluster nodes to be able to access the key-value store.

The arguments of the key-value operations are as follows. In the case of a `put` it is the file pathname of the file with the value. The key is computed by the client, from the value, and should be printed to the standard output by the test client. For the other two operations, i.e. `get` and `delete` the argument is the key returned by `put`.

The key-value operations should use TCP as the transport protocol.

Membership change requests must be sent directly to the node that should join/leave the cluster.

The membership operations can use any transport protocol. However, you will get up to 5%, if you use RMI.

The maximum scores for the membership service and the key-value store service already include the test client, without which you cannot demonstrate your key-store service.

Invocation of the Test Client

The test client should be invoked as follows.

```
$ java TestClient <node_ap> <operation> [<opnd>]
```

where:

<node_ap>
: is the node's access point. This depends on the implementation. If the service uses UDP or TCP, the format of the access point must be <IP address>:<port number>, where <IP address> and <port number> are respectively the IP address and the port number being used by the node. If the service uses RMI, this must be the IP address and the name of the remote object providing the service.

<operation>
: is the string specifying the operation the node must execute. It can be either a key-value operation, i.e. `"put"`, `"get"` or `"delete"`, or a membership operation, i.e. `"join"` or `"leave`

<opnd>
: is the argument of the operation. It is used only for key-value operations. In the case of:

  `put`
  : is the file pathname of the file with the value to add

  otherwise (`get` or `delete`)
  : is the string of hexadecimal symbols encoding the `sha-256` key returned by `put`, as described in the next section.

8. Implementation Aspects

You must implement this project using either the most recent version of Java SE, 18, or one of the still maintained LTS versions of Java, i.e. 8, 11 or 17.

Furthermore, you can use only Java SE packages. No other Java packages are allowed without our explicit permission. To get permission to use a package that does not belong to Java SE, you must submit a request via the project's Moodle forum, and explain why you want to use that package. Our response to that request applies to all groups. To make it easier to track these requests, please use one message per package.

Regardless, the values of every key-value pair must be stored in a file system file whose name is the respective key. This will allow you to more easily show that your implementation works as required.

Message syntax

You can use the message syntax that you see fit. However, all messages should be char-based, i.e. they should be human readable without further processing. The Appendix specifies a generic char-based message syntax that we used in one project in another course. You can use it, or adapt it, if you wish.

Encoding of Hash Values in Frames and Print Statements

As mentioned above, keys, and hashed node ids used for consistent hashing are values obtained using the `SHA-256` cryptographic hash function. Their length is 256 bit, i.e. 32 bytes, and must be encoded as 64 ASCII character sequences, as follows: each byte of the hash value is encoded by the two ASCII characters corresponding to the hexadecimal representation of that byte. E.g., a byte with value `0xB2` should be represented by the two char sequence `'B''2'` (or `'b''2'`, it does not matter). The entire hash is represented in big-endian order, i.e. from the MSB (byte 31) to the LSB (byte 0).

Filesystem Structure

In order to allow testing the several nodes on a single computer, each node should use its own filesystem folder to keep the values of the items it is responsible for. If all nodes share the same folder, we will not be able to check, by looking only at the filesystem, that your store behaves as expected.

Concurrency

Your implementation must be such that a node must be able to process several requests at the same time. An implementation based on **thread-pools** can increase your grade by up to 10%. You can also get an increase of 5% if you use **asynchronous I/O**. These two increases are additive.

9. Final Considerations

9.1 Development Strategy

Follow an incremental development approach: before starting the development of the functionality required by one operation, complete the implementation, of both the node and the client, of functionality (excluding enhancements) required by the previous operation.

Try to implement operations that depend on others only after the latter. For example, you should implement `LEAVE` messages only after implementing `JOIN` messages.

Implement the enhancements, i.e. the features that give you extra credit, only after completing all the protocols without enhancements.

Nevertheless, we suggest that you partition the project as follows:

1. Membership protocol, both node and client
2. Key-value operations, both node and client
3. Key-value transfer upon membership change

and that you assign each of these parts to a different group member.

Note that you can develop version of the key-value operations without previously implementing a membership protocol. E.g. the group membership can be static and your code can read it from a file.

Likewise, you can develop a first version of the key-value transfer protocol without the membership protocol. Of course, in the final version, the key-value transfer protocol execution should be triggered by a membership event.

9.2 What and how to submit?

You must submit your project via the Gitlab service at FEUP, using the project that was already assigned to you. If, in the second project, your group comprises students that were in different groups in the first project, please ask your lab instructor to create a new group for you

For the second project you must use the directory named `proj2`. This directory has two subdirectories: `src` for Java source files and `doc` for documentation files.

In the `doc` directory you must include a `README.txt` file with instructions for compiling and running your code.

Under the same directory, you should also submit a report, a PDF file named **`report.pdf`**, with up to 8 pages. The report should focus on the features that increase the ceiling of your project (see the grading criteria). Note that the report must include references to the code. If you just implemented the basic functionality using plain threads, all you need is to document the missing details, including the format of the messages that you have implemented, of 1) the membership protocol, and 2) the key-value storage service.

You will get no credit for a feature that you have implemented but that you have not described in the report. Likewise, a poorly described feature will most likely get you a low score in that feature.

9.3 Demo

You will have to demo your work in the lab classes after the submission deadline.

To streamline the demo, **you will be required to start both the nodes and the testing client from the command line.** We recommend that you write a simple script for that. The earlier you do it, the more time you will save invoking your programs during development.

9.4 Grading Criteria and Weights

A concurrent implementation of the basic protocols using plain threads is worth a maximum project grade of 60%, as shown by the following table:

Criteria	Implementation	Report		Weight
	Features/Comments	Pages	What?	
Membership Service	With avoidance of stale info	2-2.5	Must include message format and description (how and why) of missing details (including fault-tolerance).	30%
Storage Service	Including pair transfer on membership changes	1.5-2	Must include message format and description (how and why) of missing details (including fault-tolerance).	30%
	Pairs transfer **enhancement** Only without replication	0.5	How and why?	5%
Replication		1-1.5	How and why? Implications on membership and storage services	20%
Fault-tolerance	Other failure scenarios.	0.5-1	Only failures scenarios not addressed in Sections 3 or 4. The scenarios described in these sections should be covered under the respective service.	10%
Concurrency	Thread-pools	0.75-1	How and why?	10%
	Asynchronous I/O	0.75-1	How and why "minimizes" number of threads.	5%
RMI		0	Reference to Java source file with definition of the remote interface. Should be included as a subsection of the section on the membership service.	5%

The two columns under "Report" are the number of estimated pages and the contents of report for each of the criteria. Note that the total size of the report exceeds the maximum 8 pages. This is because the total weight exceeds 100%, therefore you need not to implement all the features that give you a maximum score on each criteria. Actually, you may get 100% even if your score on some criteria, for example Asynchronous I/O, is 0. In addition, the "pairs transfer enhancement" and replication are mutually exclusive.

Appendix A: Generic Char-Based Message Syntax

This appendix specifies a generic syntax for char-based messages that was used in a similar project of another course. This generic syntax was then used to instantiate the syntax of different messages of different protocols.

The generic message is composed by two parts: a header and the body. The header contains essentially control information, whereas the body is used for the data and may be optional, or even be omitted, for some messages.

Header

The header consists of a sequence of ASCII lines and terminates with an empty header line. Each ASCII line is a sequence of printable ASCII codes **terminated with the sequence `'0xD''0xA'`**, the ASCII codes of the CR and LF chars respectively, which we denote by <CRLF>. Each header line is a sequence of fields, sequences of printable ASCII codes, separated by spaces, the ASCII char `' '`. **Note that:**

1. there may be more than one space between fields;
2. there may be zero or more spaces after the last field in a line;
3. the header always terminates with an empty header line. I.e. the <CRLF> of the last header line is followed **immediately by another <CRLF> without any character, white spaces included, in between.**

Using this generic header syntax you can define the syntax of different messages of possibly different protocols, by specifying for each message:

- the number of non-empty header lines
- the fields (both their semantics and their syntax) of each of these header lines

Body

When present, the body contains data. The protocols must not interpret the contents of the `Body`. For the protocols its value is just a byte sequence. You **must not** encode it.
