Department of Information Engineering and Computer Science
University of Trento, Italy

Distributed Systems 1 - A.Y. 2021/2022

Distributed Multi-level Cache

Authors:
Bortolotti Samuele,
De Menego Luca

July 29, 2022

Contents

1 Introduction

The project entails putting a *distributed multi-level cache protocol* into practice.

The employment of caching has been successfully implemented to enhance the performance of static content on the Web. Caching proxies maintain local copies of frequently requested resources, enabling large organizations to significantly reduce their upstream bandwidth usage and costs while significantly enhancing performance, which reduces the load or traffic on the network. Caches' primary objective is to prevent database congestion since they store frequently requested items and handle the bulk of client requests separately. In order to hold many cache objects and achieve a cache access performance that is nearly as fast as the CPU, *multi-level cache* uses more than one level of cache [1].

We are focusing our attention on two-layer multi-level caches, which indicates that there are two layers of caching, as stated in the project description. Some caches operate at the first level, with a direct connection to the database. These are referred to as L1 caches. L2 caches have L1 caches as their parents.

The goal of the assignment is to guarantee *eventual consistency* in presence of cache crashes. Eventual consistency is a consistency model used in distributed computing to achieve high availability that obliquely ensures that all accesses to a particular data item will eventually return the most recent value if no new modifications are made to that item [2].

The project has been implemented with *Akka*[1], a toolkit for building distributed and concurrent message-driven applications which offers to developers an API with a high level of abstraction. The *Akka* actors involved are:

- `Client`: interacts with L2 caches asking to read or write data;
- `Cache`: is divided internally in L1 and L2. When unable to resolve a client's request, the L2 cache forwards it to L1 caches. In this case the requests are routed to the L1 cache via L2 caches. If it can, the L1 cache responds to L2 caches right away; if not, it sends the queries to the database;
- `Database`: the data source of the system.

1.1 Assumptions

The implemented systems considers several assumptions, specifically:

- links are FIFO and reliable;
- the multi-level cache architecture structure does not change over the execution of the protocol. The multi-level cache architecture is predefined and organized in a tree-like fashion;
- caches and the database have unlimited memory for storing items;
- upon crashing, nodes retain knowledge about the multi-level cache architecture, however all their items are removed;
- neither clients nor the database crash;
- it is possible to detect crashes, indeed caches which have crashed will recover after a certain delay;
- one client can perform one request at a time. However, multiple clients can generate concurrent requests.

[1] `https://akka.io/`

1.2 Operations

The two primary operations that clients can carry out on data stored in the database are *READ* and *WRITE*. If the requested value is present in memory, the cache layers immediately handle *READ* operations, if not, they are sent to the database which replies with the requested data item. Instead, *WRITE* operations always make use of the main database. The database saves the modified value and propagates it to all L1 caches after a *WRITE*, which then propagates it to all L2 caches. Once their write operations have been confirmed, clients engaging with the same cache are guaranteed not to get out-of-date values, and the protocol should provide *eventual consistency*.

Additionally, two "critical" variations of the fundamental operations, namely *CRITREAD* and *CRITWRITE* are implemented, which offer better assurances. A *CRITREAD* always retrieves the most recent value for a specific item, therefore the request is always sent to the database. Hence, even if a cache has a replica of the requested data item it does not answer. As contrast to a regular *WRITE*, a *CRITWRITE* ensures that no client will be able to read the new value of a specific item from any cache, followed by the old value. This implies that before the operation is finished, the database must make sure that no cache has an out-of-date value for the written item.

1.3 Crashes and Recovery

The system employs a simple timeout-based crash detection technique since caches can crash at crucial points during the execution of the algorithm:

- a client selects a different L2 cache and reroute its requests if it finds out that one L2 cache has crashed;
- a L2 cache selects the primary database as a temporary replacement parent when it detects that its L1 parent has crashed.

Assuming that all the caches data items are stored in a volatile memory, cache crashes result in the loss of all cached items. However, they continue to keep track of system data like the database actor and the reference of their tree neighbors. Caches recover and reactivate after a predetermined period of time.

Despite being in a consistent state, the system is stalled when the network is slower than the timeout each cache sets, because the clients continue to perform new requests while the caches continue to register timeout failures, preventing the system from progressing.

2 Project Structure

The project can be built with the employment of *Gradle*[2] or directly run with *Docker*[3]. The project's *README* contains all the instructions needed in order to install and run the program.

The source code can be found in the `src` folder, containing:

- the actual protocol implementation, in the `main` directory;
- a number of well prepared unit tests which are evaluated in Section 4 and are located in the `test` directory.

The most relevant files located in `src/main/java/it/unitn/disi/ds1` are:

[2]`https://gradle.org/`
[3]`https://www.docker.com/`

- `Main.java`: the main function of the project, allowing to start a predefined *Akka* architecture implementing the distributed multi-level cache protocol. Additionally, the program has a command line parser which can be used to configure the system parameters, such as how long each iteration should last and how many L1 caches, L2 caches per L1 cache, and clients should be used.
- `Config.java`: class which defines all of the configurations the program uses, along with some helpful enumerations.
- `Utils.java`: class which offers a variety of static functions, it is employed by the majority of the system constituent entities.
- `Logger.java`: class used for formatting logs that exposes two alternative loggers:
    1. *DEBUG*: linked with the standard output, and only used to follow the protocol flow when it is running;
    2. *CHECK*: linked with the file `logs.txt`, it is used to check whether the system was consistent throughout the entire run or not. Such logs are prepared so that they may be read by `Checker.java` when the program is being evaluated;
- `Checker.java`: class which checks the consistency of the logged run using the created `logs.txt` file.

The exposed directories are:

- `actors`, it contains all the implemented *Akka Actors*, presented in Subsection 2.1;
- `messages`, it contains all the messages exchanged by the actors, presented in Subsection 2.3;
- `structures`, it contains the implementation of the tree architecture.

2.1 Actors

The actors engaged in our system are: *Database*, *Cache* (L1 and L2) and *Client.*

Actor has been first defined as a base abstract class that all other actors *extend.* It implements basic functions while providing the actors fundamental functionalities.

The database is represented as a key-value pairs `HashSet` of integers which is stored by the `Database` actor. It also initiates a bespoke protocol based on the *Two-Phase commit* algorithm when critical writes are requested.

There are two main types of *Cache* actors: L1 and L2. The parent of L1 caches is the database itself, whereas L2 caches have other L1 caches as parents. This is the main distinction between L1 and L2 caches. Each cache has a cached database which maps each item to its most recent known value, and it is updated as part of the regular system flow.

The system's ultimate user is represented by the *Client* actor. Each client can communicate with L2 caches to perform read and write operations. If there is a problem with the operation, the result is `null`; otherwise, the requested value is returned in the case of a read, or the updated value in the case of a write. The client also receives a sequence number in addition to the value, which is used in order to ensure all *eventual consistency* properties are fulfilled. The employment of the sequence number is thoroughly discussed in Subsection 3.2

2.2 Reads Flow

The following steps can be used to illustrate how a *READ* operation typically proceeds:

1. the program generates a `ReadMessage` specifying the requested `key` and indicating the read is critical or not critical;

2. the program sends the message to a given client;

3. the client generates a *Universally Unique Identifier (UUID)*[4] linked with the newly created `ReadMessage`, associating the sequence number of the last accessed value for the requested data item, then it sends the request to a random L2 cache and schedules a timer;

4. the client is now waiting for a response: it cannot not perform other requests;

5. the L2 cache receives the `ReadMessage` and checks whether the requested key is in the middle of a critical update. If it is, it sends a `ResponseMessage` to the client with an error;

6. otherwise, the cache behaves differently based on whether the read request is critical or not:

   - *READ*: the cache checks if it already has a value mapped to the requested key in its cached database. If it is the case, and the sequence number associated to the requested data item is greater or equal to the one indicated by the client, it sends a `ResponseMessage` to the client with the requested value, otherwise the `ReadMessage` is forwarded to its L1 cache parent, and a timeout is set;
   - *CRITREAD*: the cache immediately forwards the `ReadMessage` to its L1 cache parent;

7. the L1 cache receives the `ReadMessage`, and performs the exact same steps as the L2 cache. In this case, however, the parent is the database;

8. if the database receives the `ReadMessage`, it gets the value associated with the requested key from the database, and according to the set of *hops* stored inside the message the `ResponseMessage` follows the path backwards;

9. during the backward step, each cache stores the newly updated value, together with its sequence number, inside the cached database;

10. upon reaching the client, the client stores the sequence number associated to the obtained value.

2.3 Writes Flow

The following steps can be used to illustrate how a *WRITE* operation typically proceeds:

1. the program generates a `WriteMessage` defining the key-value pair, and indicating the write is critical or not critical;

2. the program sends the message to a certain client;

3. the client generates a UUID linked with the newly created `WriteMessage`, sends the request to a random L2 cache and schedule a timer for a possible timeout;

4. the client is now waiting for a response: it cannot perform other requests;

[4]UUID, which stands for universally unique identifier, represents a 128-bit value, which are for practical purposes unique [3].

5. the L2 cache receives the `WriteMessage` and checks whether the requested key is in the middle of a critical update. If it is, it sends a `ResponseMessage` to the client with an error;
6. otherwise, the cache forwards the request to its parent: an L1 cache;
7. the L1 cache receives the `WriteMessage`, and performs the exact same steps as the L2 cache. In this case, however, the parent is the database;
8. the database receives the `WriteMessage`, and it behaves differently based on whether the write request is critical or not:
    - *WRITE*: the database updates the value based on the key-value pair in the request and the associated sequence number, then it propagates the updated value together with the new sequence number to all caches, which updates their cached database if it contains a value mapped to the updated key and the sequence number is greater than the one stored. Thanks to the *hops* specified in the `WriteMessage`, the response is also forwarded back to the client.
    - *CRITWRITE*: a customized protocol devised from *Two-Phase Commit* is started, outlined in Subsection 3.7.
9. Upon reaching the client, the client stores the sequence number associated to the obtained value.

2.4 Crashes simulation

The project allows to easily set up crashes simulation in predefined focal points of the implemented protocol. The points are specifically described in the `Config.java` file, and they are primarily:

- before/after reads/writes;
- before/after responses;
- before/after/during multicast;
- before/after/during flushes.

In order to implemented crashes, each cache can enter in a *Crashed* state, defined with a custom *Akka Behaviour* in which the cache can only handle `RecoveryMessages` (used to recover from a crash) and `TokenMessages` (used for performing distributed snapshots). In order to make a certain cache crash during the protocol, a `CrashMessage` must be sent to it, containing the exact code location in which the cache should crash, defined from an enumeration of predefined possible positions, and the amount of time after which the cache should recover.

As explained in the Section 1, on recover the cache only keeps information about the system architecture; everything else will be lost.

2.5 Messages

The following messages, which can be located inside the
`src/main/java/it/unitn/disi/ds1/messages/` folder, have been developed in order to meet all the project's requirements:

- `Message.java`
- `JoinCachesMessage.java`
- `CrashMessage.java`

- `RecoveryMessage.java`
- `FlushMessage.java`
- `TimeoutMessage.java`
- `WriteMessage.java`
- `ReadMessage.java`
- `ResponseMessage.java`
- `StartSnapshotMessage.java`
- `TokenMessage.java`
- `CriticalUpdateMessage.java`
- `CriticalUpdateResponseMessage.java`
- `CriticalUpdateTimeoutMessage.java`
- `CriticalWriteResponseMessage.java`

All of the other messages' classes inherit from the `Message` class, which is found in the `Message.java` file and is a *Serializable* empty class.

The message used to introduce the caches to one another and that enables them to set a reference to the parent and to the children is contained in `JoinCachesMessage.java`.

The messages used to instruct the cache to crash and to recover from the crashes are included in the files `CrashMessage.java` and `RecoveryMessage.java`, respectively.

The `FlushMessage` is used to tell to a cache to delete every copy of the data items.

A special type of message called a `TimeoutMessage` is used to alert a cache that the actor from which it was expecting a response is probably going to crash.

The client and caches utilize the `WriteMessage` and `ReadMessage` messages to request a *WRITE/CRITWRITE* or a *READ/CRITREAD* operation, respectively.

`ResponseMessage` is a text message that provides the client request's answer, as produced by a cache or a database.

In order to take a *distributed snapshot* of the system, `StartSnapshotMessage` and `TokenMessage` are used. They were used during the initial phase of testing to make sure that all updates had spread across the system.

Lastly, the messages `CriticalUpdateMessage`, `CriticalUpdateTimeoutMessage`, `CriticalUpdateResponseMessage` and `CriticalWriteResponseMessage` are utilized to build a protocol similar to *Two-Phase Commit* in order to implement the *CRITWRITE* operation. In Subsection 3.7 they are extensively covered.

Just to offer a quick overview, the `CriticalUpdateMessage` is used to get information from the caches, namely whether or not they can update the data. The message we use to determine if a cache from which we are expecting a `CriticalUpdateResponseMessage` from has crashed is `CriticalUpdateTimeoutMessage`. `CriticalUpdateResponseMessage` is the message that contains votes from the caches that are either *OK* or *NO*. The `CriticalWriteResponseMessage`, is used to inform the caches to either *COMMIT* or *ABORT* the critical write.

2.6 Architecture

In order to facilitate the interaction with the multi-level tree-system which is requested by the assignment, we have developed a simple architecture whose components can be found in the `src/main/java/it/unitn/disi/ds1/structures/` folder.

Such architecture comprises:

- a list of client actors;
- the tree of caches including the database.

The database serves as the root of the cache tree, an *n-ary Tree Data Structure*. The complete system structure can be managed easily since each node is made up of an actor and a list of children.

Each branch of the tree is balanced whether there are L1 cache crashes or not. Each L1 cache in the initial scenario has exactly the same amount of children by design. When an L1 cache fails, the children which are aware of it mark the database as its parent and become *unavailable*, meaning that all client queries are ignored until the original parent is available and the branch is thus restored.

3 Implemented protocol

The adopted protocol complies with all the criteria outlined in the earlier sections. Eventual consistency is ensured by the complete propagation of freshly updated data to L1 and L2 caches and the use of sequence numbers satisfies some crucial properties such as the *Monotonic Read*. A modified *Two-Phase Commit* mechanism is employed so as to satisfy critical write operations. The following subsections provide further information and illustrations.

3.1 Eventual Consistency

Eventual Consistency is a consistency model which ensures that any modifications to a specific data item will *eventually* reach all of the replicas involved in the system. Since the database is the sole entity in our particular situation which can do writes, we would like the caches to get updated data in a finite amount of time. We can easily achieve this property in a situation without crashes by only propagating modified values because we can assume that links are reliable:

- the *database* receives a write request and updates the corresponding value;
- the *database* multicast the data to all L1 caches;
- every *L1 cache* multicast the data to all L2 caches.

Although the propagation of the information may take some time, it is fair to presume that all caches will ultimately receive the updated value. Additionally, because there is only one source of data and the links are FIFO, each replica will experience the same global interleaving, which is based on the sequence in which queries are processed by the database.

This straightforward scenario, is, however, ineffective if caches crash. Two aspects of the system have been used to address this issue:

- when a cache fails, all objects which were saved are lost;
- crashed caches will eventually recover after some configurable time, so recovery actions are always performed.

We can confidently assert that an L2 cache crash does not cause the system to enter a non-consistent state as a result of the first property. Actually, the cache would just lose all data it has saved, hence any requests would have to be sent back to the cache's parent. When an L1 cache fails, instead, issues occur since some of its L2 children do not receive the updated information. Our protocol establishes a specific recovery operation that each cache must carry out in this unique corner case, that consists in the multicast of a `FlushMessage` to all children. When a cache gets a `FlushMessage`, it clears its memory and deletes any items that were previously cached. As a result, all entities in a sub-tree will ultimately reset when an L1 cache crashes.

The system may have some inconsistencies when many replicas are accessed by the same client over a short period of time using various caches since *eventual consistency* only guarantees consistency as long as clients always access the same replica. Due to the fact that clients in the established architecture are free to contact whichever cache they want, such issue with *eventual consistency* is likely to frequently occur. Taking into account that *eventual consistency* may be thought of as a superclass of *client-centric consistency*, we have devised a *client-centric consistency* for the proposed project since it provides significantly higher guarantees.

As an important notice, as it may be clear from the following sections, to comply with *eventual consistency* alone it is sufficient to remove the *sequence number* from all actors.

In the following paragraphs, we discuss how *sequence numbers* allow us to comply with the *client-centric consistency* model.

3.2 Monotonic Read Property

If a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more recent value.

– Monotonic reads

In order to design a protocol that is effective and able to handle the *Monotonic reads* feature, we added *sequence numbers* associated with each version of the data recorded in the database and subsequently replicated in the caches. In practice, each *Akka* actor inherits the `seqnoCache HashMap` from the `Actor` class, which is used to store the sequence numbers. Therefore, it is feasible to get the requested item's sequence number by querying the map with the data item's key. Every time a write request is received, the database generates a sequence number, which is then propagated to all caches that have previously stored the value.

As an effective representation of how the sequence numbers allow our multi-level distributed cache to ensure the *Monotonic reads* property, let us consider the following scenario.

Suppose client `C1` requests to get the value of a data item x. Let us assume that `C1` has read the most current value of this recently updated data item; under these circumstances, it is logical to presume that the update may not have propagated to all caches.

The client `C1` selects an L2 cache at random, which we will refer to as `L2-01` for the sake of this exposition, as stated in the Reads Flow Subsection 2.2.

Let us assume that `L2-01` already owns a copy of the data item x, but that cache has not yet received the update. Therefore, it would unavoidably violate the *Monotonic reads* property if `L2-01` replies with the old value it now possess.

With the protocol we have developed, we can get around this problem and ensure the *Monotonic reads* property. In order for the cache to accurately determine which version of the data the client knows, the client first includes the most current sequence number it possesses inside of the read request. Additionally, because the client cannot crash by assumption and cannot handle more requests at once, the solution has been greatly simpler. When the request is received, the `L2-01` realizes it has a lower sequence number than the one `C1` has accessed, so it responds with an error message.

The update of x will ultimately reach the cache `L2-01`, and the client is free to query another L2 cache since eventual consistency holds for our multi-level cache. Instead, if `L2-01` had the identical or a more current version of x, it would have responded with the information in x.

As a remarkable consideration, some alternative solutions could have been:

- `L2-01` waits for the reception of a new update: this is a blocking approach since the client must wait for the new update, which may take some time to arrive;
- `L2-01` requests the database or the parent cache for the data item x. This approach is not compliant with the instructions we were given since a cache may only ask for an update explicitly if it already possesses the data item, which is not the case here. Additionally, since the update would ultimately reach the cache `L2-01` anyhow, therefore such approach increases network traffic and results in duplicate messages across the system.

3.3 Monotonic Writes Property

A write operation by a process on a data item x is completed before any successive write operation on x by the same process

– Monotonic writes

The *Monotonic writes* property is trivially satisfied thanks to the following assumption, specified in the assignment Subsection 1.1: "One client can handle just one request at once".

As a result, anytime a client requests a write operation, it must wait until that operation is completed before asking for another write request.

The outcome of write request can either be:

- a confirmation that the data has been written successfully;
- an error message.

In both situations, the client is fully aware of whether or not their request has been granted.

3.4 Read Your Write Property

The effect of a write operation by a process on a data item x will always be seen by a successive read operation on x by the same process

– Read Your write

In order to satisfy the *Read your write property* we have relied on sequence numbers. Similar to the *Monotonic read* property, the client always provides the sequence number associated with the most recent value of the data item it has read in the request whenever it asks for a read operation on a specific data item.

In such circumstances, three results are conceivable following a client's write request on a specific data item:

- if the L2 cache that the client has reached does not have a copy of the requested data item, it will request it from the parent cache, eventually reaching the database;
- the L2 cache which has been contacted by the client owns a replica of the requested data item, however the sequence number associated is lower than the one specified by the client. Since the L2 cache is fully aware that the modified value will ultimately be propagated as eventual consistency holds, it responds with an error message in this case;
- the L2 cache which has been contacted by the client owns a replica of the requested data item whose associated sequence number is equal or greater than the one specified by the client. The L2 cache can correctly answer in this situation.

3.5 Writes Follow Reads Property

A write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or more recent value of x that was read

– Writes follow reads

Likewise *Monotonic writes* property, the *Writes follow reads property* is trivially satisfied thanks to the assumption specified in assignment Subsection 1.1: "One client can perform one request at a time".

As a result, a client must wait for the results of the first action before requesting a write operation on a data item after completing a read operation on the same item. The order of the operation is also consistent across all caches because to the use of sequence numbers, which can only be set by the database.

3.6 Critical Reads

As stated in Section 1, when performing a critical read a client is guaranteed to receive the most recent value for a certain data item. We cannot presume caches will have enough information to discern whether or not they contain the latest recent data since the consistency model our protocol uses as its foundation is *eventual consistency.* as a result, in the proposed solution when a cache receives a citical `ReadMessage`, it simply forwards it to its parent. In this way, the message will reach the database, the only entity whose data items are up to date. The `ResponseMessage` continues the path backwards, eventually reaching the initial client, as described in Subsection 2.2.

A schematic representation of the entire flow is depicted here:

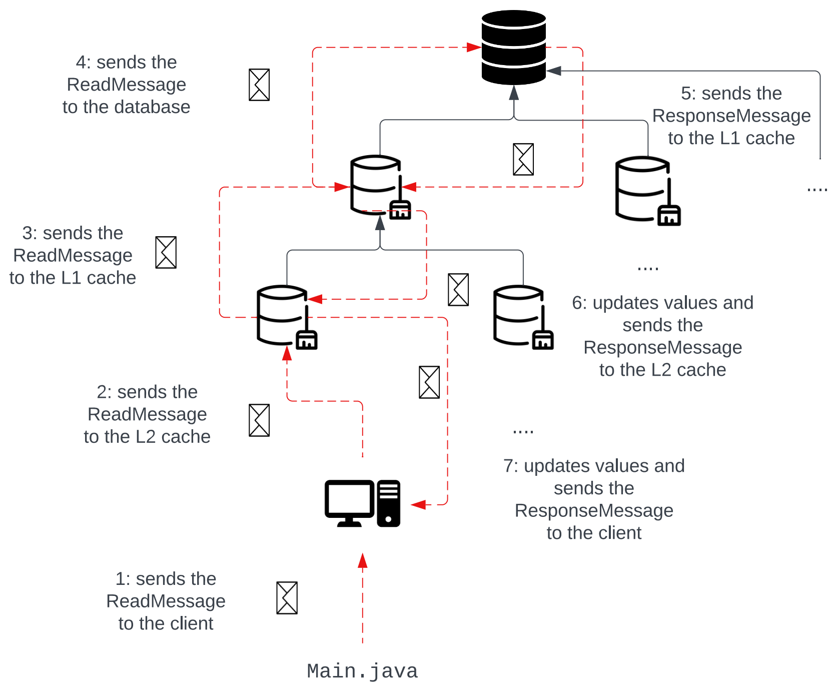
4: sends the ReadMessage to the database
5: sends the ResponseMessage to the L1 cache
....
3: sends the ReadMessage to the L1 cache
....
6: updates values and sends the ResponseMessage to the L2 cache
2: sends the ReadMessage to the L2 cache
....
7: updates values and sends the ResponseMessage to the client
1: sends the ReadMessage to the client
Main.java

Figure 1: Critical Read Flow

3.7 Critical Writes

The most arduous functionality that must be completed in order to effectively finish the project is presumably the critical write. According to the specified requirement, the database must make sure that no cache contains an outdated value for the written item before the update is performed. Furthermore, no client should be able to read the new value from any cache, followed by the old value. The database propagates the change as for a write operation once it has confirmed that the cached objects have been deleted.

We extensively drew inspiration from the *Two Phase Commit* protocol, particularly the *COMMIT-ABORT* procedure, to satisfy these requirements.

The *Critical Writes* flow behaves as follows:

1. the program generates a `WriteMessage` defining the key-value pair, and indicating that the write is critical;
2. the program sends the message to a certain client;
3. the client generates a *UUID* linked with the newly created critical `WriteMessage`, sends the request to a random L2 cache and schedules a timer for a possible timeout;
4. the client is now waiting for a response: it cannot perform other requests;
5. the L2 cache receives the `WriteMessage` and checks whether the requested key is in the middle of a critical update. If it is, it sends a `ResponseMessage` to the client with an error;
6. otherwise, the critical `WriteMessage` is forwarded to its parent: an L1 cache;
7. the L1 cache receives the critical `WriteMessage`, and sends the request to its parent, the database;
8. the database stores the request and locks the requested item, in this way every actor who wants to either read or update that data item gets an error message as a reply. The update cannot be performed yet, as we need to make sure that *no cache holds an old value for the written item*;
9. the database creates a `CriticalUpdateMessage`, then it generates a *UUID* to identify the message;
10. the database multicast the `CriticalUpdateMessage` to its children L1 caches, scheduling a timer for the answer;
11. upon the receipt of the `CriticalUpdateMessage`, the L1 cache locks the data item and sends it to its children L2 caches, scheduling a timer for the answer. In this way, any request to access in read or write to that data item produces an error message;
12. the L2 cache then locks the data item as well and answers the L1 cache with `CriticalUpdateResponseMessage` confirming that it is *OK* with the update. From now on, any request to access in read or write to that data item produces an error message;
13. the L1 cache at this point will collect their children answers, which can lead to the following outcomes:
    - the L1 cache got either a *NO* from a child L2 cache or it has registered a timeout for the `CriticalUpdateMessage` response before taking a decision, in this case it takes as final decision to *NO*. The *NO* decision in practice is taken only in a the case a timeout expires, however nothing prevents us from introducing some business logic and devise some cases in which a *NO* vote is necessary.
    - the L1 cache got a *OK* from all its children L2 caches, in this case its final decision is *OK*;

14. after the decision has been taken, the `CriticalUpdateResponseMessage` is sent by the L1 cache to the database;

15. the database now has the following options:

    - it has registered a time-out or at least one L1 cache has voted *NO*, in this case the *Critical Write* is unsuccessful, therefore it removes the lock from the data item and multicast a `CriticalWriteResponseMessage` to all L1 caches, telling them to *ABORT* the *Critical Write*;
    - all the L1 caches have agreed (voted *OK*) to complete the *Critical Write*, therefore the database is sure all the values corresponding to the key of the requested item are locked, hence it can perform the update it has memorized at the beginning of the procedure assigning the newly generated sequence number, then it multicast a `CriticalWriteResponseMessage` to all L1 caches, telling them to *COMMIT* the *Critical Write*;

16. the database has decided to either *ABORT* or *COMMIT*. In both cases, the L1 cache gets a `CriticalWriteResponseMessage` message. The cache unlocks the data item and, if it is an *ABORT* message, it discards the update, otherwise the updates are saved and the `CriticalWriteResponseMessage` message is forwarded to its children L2 caches;

17. the L2 cache will perform exactly the same operations as the L1 cache, except sending the message to its children.

As an important aside, any crash will result in a timeout failure, which results in a *NO* decision by either the database or the L1 caches or an *ABORT* by the L1 caches.

The primary goal of the locks and the of the "vote" procedure is to satisfy the condition that the database must only write new data items if no cache contains an older value for the written item. In fact, we are assuring the following statements by locking the data item prior to changing the final vote:

- no client can access that data item during the update;

- if a cache has agreed but crashes afterwards the protocol still works, since it cannot ask for an old value to its parent cache (the data item is still locked).

The database is also the first and only source of data following the update, which will eventually release all of the locks that have been put up across the system. In the case in which one cache crashes before receiving the *COMMIT* or *ABORT* decision, we know, based on the assumption that one cache will resume after a predetermined amount of time, that it will eventually clear its memory and, if it is an L1 cache, that it will send a flush message to all of its children. As a result, all caches are still consistent since their cache storage has been cleared. Any received request will need to be forwarded to the database, regardless of which decision has been taken.

As it may now be clear, either the caches crash or the procedure completes. We have shown that in both cases either the data item we are required to update is not present or it is updated.

In reality, the data item can still contain an old value, however it is not accessible by anyone, especially by the client. Indeed, we could have deleted the data item instead of locking it, however this choice would have been inconvenient in case of *ABORT* decisions.

There may be a few drawbacks to the strategy we have used, including:

- several messages are needed in order to complete the *Critical write*. However they are needed for the above stated reasons;

- the requested data item is not accessible by other clients for almost the entire procedure. However, all the other data items remain available.

To wrap up, a schematic representation of the entire pipeline is depicted here:

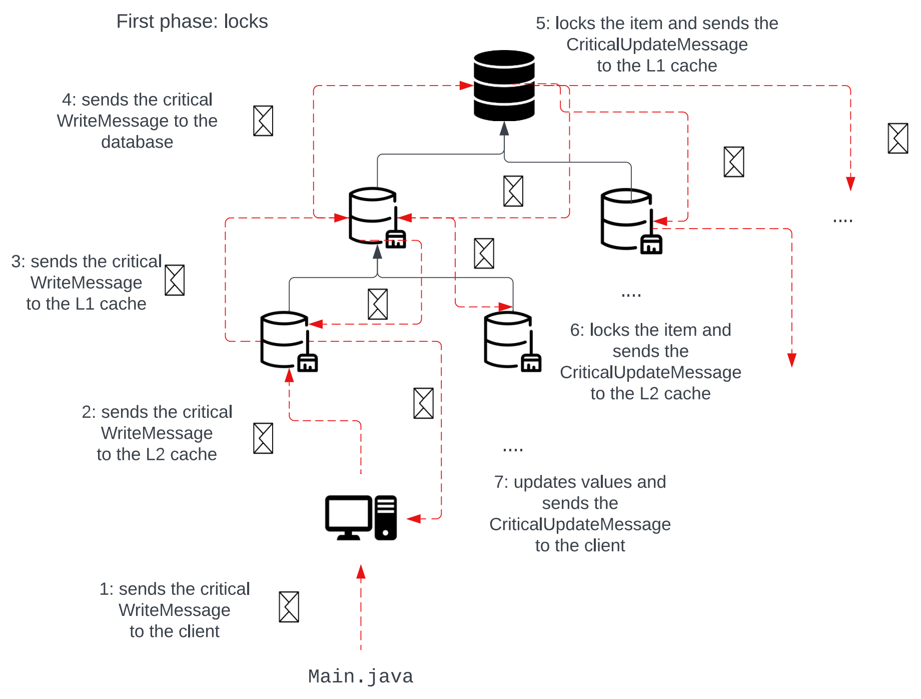
First phase: locks
5: locks the item and sends the CriticalUpdateMessage to the L1 cache
4: sends the critical WriteMessage to the database
3: sends the critical WriteMessage to the L1 cache
....
6: locks the item and sends the CriticalUpdateMessage to the L2 cache
2: sends the critical WriteMessage to the L2 cache
....
7: updates values and sends the CriticalUpdateMessage to the client
1: sends the critical WriteMessage to the client
Main.java

Figure 2: Critical Write Flow Part 1

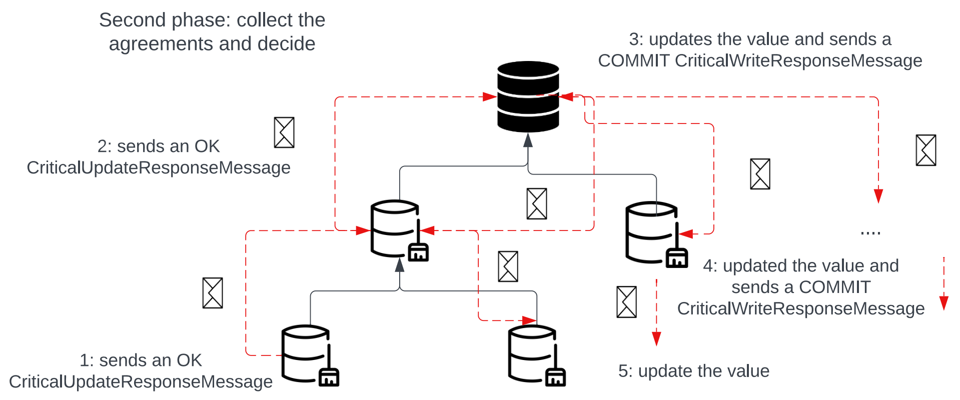
Second phase: collect the agreements and decide
3: updates the value and sends a COMMIT CriticalWriteResponseMessage
2: sends an OK CriticalUpdateResponseMessage
....
4: updated the value and sends a COMMIT CriticalWriteResponseMessage
1: sends an OK CriticalUpdateResponseMessage
5: update the value

Figure 3: Critical Write Flow Part 2

4 Tests

In order to assess the correctness of our solution, specifically whether the protocol we have set up fulfills all the *eventual consistency* requirements, we have employed the `JUnit Java Test Framework`[5], which is a popular framework used by developers to implement unit testing in Java, accelerate programming speed and increase the quality of code.

Each test follows the same structure:

- prior to running the tests, the method initializes the architecture, database, and log file with a predetermined number of caches and clients;
- the architecture and database states are reset and the log file is cleaned before each test;

[5]`https://junit.org/junit5/`

- either a random or predefined exchange of messages takes place;
- a timeout is included in order to ensure all the messages got delivered;
- an assertion employing the `Checker` which checks whether the run has been consistent by fulfilling the *eventual consistency* criterion.

All the tests we have devised can be found under the folder named `src/test/java/it/unitn/disi/ds1`, which includes the following files:

- `ECAutonomousExecution.java`
- `ECCrashBasicTest.java`
- `ECNoCrashBasicTest.java`
- `EcNoCrashTest.java`
- `Helper.java`
- `MainTest.java`

The `ECAutonomousExecution` tests are used to determine if a system performing random actions satisfies the *eventual consistency* criteria. To achieve this, one can use either the `randomMessage` or the `randomAction` methods from the `Utils.java` file. We specifically want the software to guarantee the consistency of:

- a 20 seconds run in which random requests are performed by the clients;
- a 20 seconds run in which random requests are performed and random cache crashes occur;
- a 50 requests run in which random requests are performed and random cache crashes occur;
- a 50 requests run in which random requests are performed and random cache crashes occur.

the time per run and the number of requests are both configurable inside the `ECAutonomousExecution.java` file.

The `ECCrashBasicTest.java` tests are used to verify whether the system satisfies the *eventual consistency* requirements when asked to answer to some specific requests in case of crashes. The file has "basic" in its name because it considers a simple system configuration in which there is only one L1 cache, one L2 cache and two clients.

Although it is similar to the `ECCrashBasicTest.java` in terms of types of tests, the `ECNoCrashBasicTest.java` file does not include crashes.

The `EcNoCrashTest.java` file contains tests involving a more complex architecture (more than one L1 cache and more than one L2 cache) and checks whether the consistency of the execution of several operations in cascade lead the system to a consistent state. In those tests, no crashes are involved.

The `Helper.java` file includes some useful function which are required by all the tests to work properly.

Finally, the `MainTest.java` is used in order to check that the program does not throw exceptions during its execution.

4.1 Checker

In order to automatically check whether a given run is consistent, a *checker* function has been implemented. The checker utilizes a specially built `logs.txt` file created using the *CHECK* logger throughout the protocol operation, which contains all crucial information about the occurring events in *Tab-separated values (TSV)* format, as described in Section 2.

In particular, it includes:

- the number of L1 caches, L2 caches and clients;
- the database;
- the messages exchanged by all entities;
- cache flushes.

An effective representation of a log line is depicted as follows:

```
CONFIG:        5        5        3
CONFIG:        0-52     ....     99-5
FINE:FINE        20220729_182020        6        2        CRITWRITE        ↵
↪  false        15        60        null        2d9d7c98-...     Request
↪  write for key [CRIT: true]
```

where the first two lines are employed in order to specify the configuration parameters in the system, respectively:

1. the first line depicts the number of L1 caches, the number of L2 caches associated to each L1 cache and the number of clients in the system. Therefore, in the depicted scenario the total number of caches in the system is $5 + 5 * 5 = 30$;
2. the second line displays the contents of the database, which is randomly populated at the start of the program. Each item is represented by a key-value pair, with the remaining items being represented by "…";
3. the log output from one system iteration is displayed on the third line. In this instance, we can observe the log level, which is used to filter the log messages, the timestamp associated with the operation, the identity of the actor who made the request, the identity of the actor who received the message, the name of the request, the key of the requested data item, the known value of the requested data item, the sequence number associated with the data item (in the example shown, the sequence number is `null` since it is a write operation, thus the sequence number will be generated and propagated by the database according to Subsection 3.7), the UUID associated to the request and a human readable message.

Using this information, the checker simulates the run by keeping knowledge about the current state of all entities and by verifying that each step is consistent with respect to the project requirements. Since all events are logged, keeping the state of database and caches is simply a matter of re-implementing their behaviour when certain messages are received or sent. The consistency check verifies that when a *READ* request is received:

1. if it is critical, it should get the latest value based on the current state of the database;
2. if it is not critical, it should be answered with the value contained in the cache state, if they have the requested item in memory, otherwise with the value given by the database.

5 Conclusion

The implemented system has shown to be resilient in all the test cases we have tried, fulfilling all the requirement we were requested to develop. As shown in Subsection 3.2, our system actually provides stronger guarantees with respect to the initial requirements. Eventual consistency, in fact, only works as long as clients always access the same replica. When multiple replicas are accessed by the same client over a short period of time, there may be inconsistencies. Since in the developed architecture clients are free to contact any cache they want, the usual *eventual consistency* problems may arise. The proposed solution instead achieves *client-centric consistency*, solving the problem in a relatively cheap way.

In this project, we have shown one possible implementation of the *Critical Write* functionality which ensures the required consistency criteria in spite of blocking the data item for all other clients' operations.

6 References

[1] Wikiversity, Multilevel caching — Wikiversity, `https://en.wikiversity.org/w/index.php?title=Multilevel_caching&oldid=1855180`, [Online; accessed 1-August-2022], **2018**.

[2] Wikipedia contributors, Eventual consistency — Wikipedia, The Free Encyclopedia, `https://en.wikipedia.org/w/index.php?title=Eventual_consistency&oldid=1092897334`, [Online; accessed 1-August-2022], **2022**.

[3] Wikipedia contributors, Universally unique identifier — Wikipedia, The Free Encyclopedia, `https://en.wikipedia.org/w/index.php?title=Universally_unique_identifier&oldid=1093728359`, [Online; accessed 1-August-2022], **2022**.