University of Trento
Department of Computer Science
Distributed Systems 1

Report — Multi-Level Distributed Cache

Marcel Pascal Stolin
marcelpascal.stolin@studenti.unitn.it

01/05/2023

1 Introduction

The goal of this project is to demonstrate the knowledge and skills acquired during the course *Distributed Systems 1*. This is done by implementing a distributed multi-level cache using the *Akka Classic* toolkit [1]. The purpose of a cache is to prevent high throughout-put at the database level. Therefore, clients are supposed to interact with the database through a network of caches, where caches are organized in two levels (L1 and L2, borrowed from CPU terminology). The system is optimized for *Read* operations whereas *Write* operations always involve the database. A further requirement of this system is to provide *Eventual Consistency*. Therefore, the system ensures that all participants receive the latest update of a *Write* sooner or later.

2 Conceptual Design

This section describes the conceptual design of the system and its participating nodes. Additionally, their purpose is explained further.

2.1 Actors

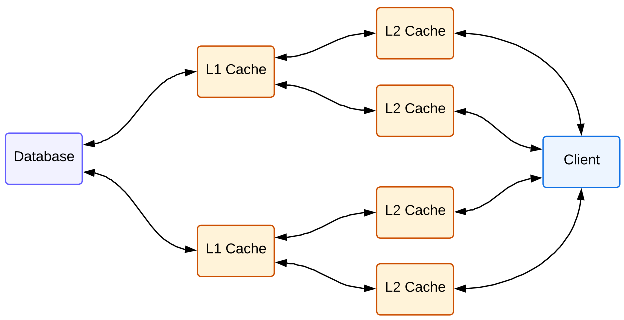

Figure 1: Overall environment architecture

Figure 1 illustrates a possible actor environment of this application. It consists of four different actors, each serving a different purpose and as seen, each actor type represents a different layer. The actors who are involved in the environment are clients, L1 and L2 caches, and a single database. Given by project description, only the caches can crash.

2.1.1 Client

A client represents a user who can instantiate operations like read or write. The client does not know about any L1 cache or the database and can only exchange messages with an L2 cache. Whenever an L2 cache has crashed, an L2 cache needs too much time to process the received message or an `ErrorMessage` is received (explained further in Section 3.1), the client will timeout. A client can perform either multiple *Read* operations or a single *Write* operation at the same time. If a client is waiting for a *Write* response and a new *Read* or *Write* request is supposed to be initiated, it will throw an error.

[1] `https://doc.akka.io/docs/akka/current/index-classic.html` (Accessed: 02-04-2023)

2.1.2 Cache

As illustrated previously in Figure 1, two types of caches exist, an L1 (level 1) cache and an L2 (level 2) cache. A cache is responsible to save frequently requested data. Whenever a request has arrived, the cache needs to check if it can serve that request immediately, otherwise it needs to forward that message to the next level. As given by the project description, caches are the only actors that are allowed to crash. On crash, a cache loses all of its temporary data. Temporary data is the storage of key-value pairs (further details are explained in Section 3.5) and other data needed for message exchange. Knowledge about the environment (references to other actors) is persistent and therefore not lost after a crash. A cache can handle multiple reads and multiple writes at the same time. However, only a single write for a specific key is allowed at the same time.

L2 Cache The L2 cache serves as the endpoint for the client, as it is contacted directly by the client. Then, the L2 cache decides if it needs to forward the received message to the next level (its parent L1 cache) or answers back immediately. As an example, when a client requests to read a value, and the client does know a more recent version of the value than the L2 cache, the L2 cache needs to forward that message to the next level to request the most recent value instead of responding with an older value.
An L2 cache only knows about its parent L1 cache and the database. If the L1 cache is not able to serve the request (e.g. due to a crash), the requesting L2 cache will timeout and forward that message directly to the database instead. Afterward, it serves the response message to the client. However, this error handling is performed for *Read* operations, *Write* operations have to be performed through the parent L1 cache.

L1 Cache An L1 caches behave as an L2 cache. Whenever a request has arrived, it checks if it can serve that request immediately. Otherwise. it forwards that request to the next level (the database) and eventually responds to the requester (a child L2 cache). In addition, an L1 cache is responsible to serve as the coordinator to its child L2 caches. Whenever an L1 cache crashes, it makes sure that all its L2 caches flush. Flushing is explained in more detail in Section 3.3.

2.1.3 Database

The database is responsible to save all data and to serve as the coordinator for all L1 caches. It always knows the most recent value for a key, because *Write* operations always involve the database. Whenever a request has arrived, it tries to serve the request immediately, as it can handle multiple reads and writes, but only a single write for the same key at the same time.

2.2 Operations

The operations that can be performed by the actors in this environment are given in the project description. These are *Read*, *Write*, *Critical-Read*, and *Critical-Write*. The system is optimized for read and writes will always involve the database, to guarantee consistency.

2.2.1 Read

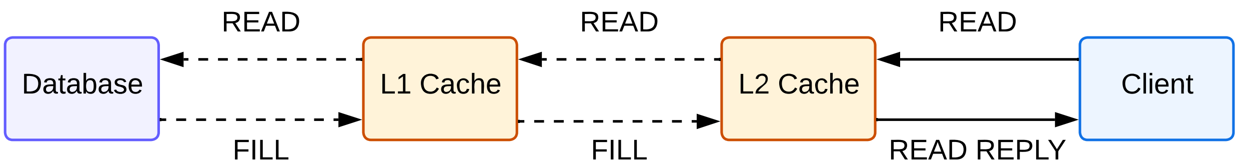

Figure 2: Message exchange during a *Read* operation

Read is a basic operation that is illustrated in Figure 2. A client initiates *Read* operations by sending a `ReadMessage` (explained in Section 3.1) to an L2 cache. The read request is always associated with a specific key that identifies the value. If the L2 cache knows a more recent or the same *Update-Count* (see Section 3.5 for details about the *Update-Count*) for the given key, it responds immediately back to the client. If not, the L2 cache forwards the request to its parent L1 cache. Then, the L1 cache behaves exactly like the L2 cache.

If needed the L1 cache forwards the *Read* request to the database. If the key is known and unlocked, the database immediately responds with a `FillMessage` to the requesting L1 cache. Then, the L1 cache updates its data with the newly received value and forwards the received `FillMessage` to the requesting L2 cache. The L2 cache also updates its data with the newly received value and responds a `ReadReplyMessage` with the requested value to the client. Multiple clients may request the same key at the same time at the same L2 cache, as well as multiple L2 caches request the same key at the same time at the same L1 cache. Therefore, a cache adds a *Read* request, it cannot serve immediately, to a list, and whenever the `FillMessage` has arrived from its parent it multicasts either the `FillMessage` to all requesting L2 caches or a `ReadReplyMessage` to all requesting clients.

The following are error cases that can happen during a *Read* operation:

The key is locked Whenever the key has already been requested for write, it will be locked. This can happen at all caches and at the database. All involved actors will directly respond with an `ErrorMessage` back to the requesting actor. This will eventually be delivered back to the client, given that no actor involved in the exchange crashes. Otherwise, the client will timeout (see Section 3.4 for details about timeouts).

The Key is unknown If the requested key is unknown at the database, it will send an `ErrorMessage` that is delivered back to the client, assuming that neither the L1 nor the L2 cache has crashed, otherwise, the client will timeout.

The L2 cache has crashed The L2 cache can either crash before the client sends a `ReadMessage` or after it has received the message. In both cases, the client will never receive the `ReadReplyMessage` and times-out. This finishes the *Read* operation.

The L1 cache has crashed The L1 cache can also crash before it receives the request or afterwards. In both cases, it will result that the L2 cache will timeout and forwards the request directly to the database since it never receives a response from the L1 cache. After receiving a `FillMessage` from the database, it multicasts a `ReadReplyMessage` to all clients who have requested to read the key.

2.2.2 Write

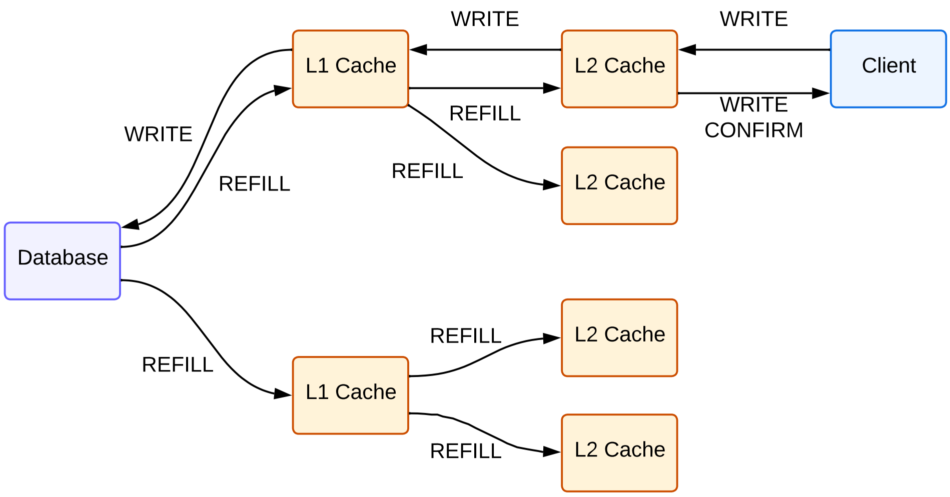

Figure 3: Message exchange during a *Write* operation

Figure 3 shows the message delivery of a successful *Write* operation. The client sends a `WriteMessage` to an L2 cache. Then, if the data is unlocked the L2 cache will forward the message to its parent L1 cache. The L1 cache will also forward the message to the database if the key is unlocked. Lastly, the database also checks if the key is not locked and then writes the new value. On success, the database multicasts a

ReFillMessage to all L1 caches. Upon receiving a ReFillMessage, the L1 caches check if they have already saved that key in an earlier operation. If yes, they will update their value. Additionally, they also multicast the ReFillMessage to all their child L2 caches. The L2 caches will also update their data if needed. However, the caches that have received the WriteMessage, will update their value even if they didn't know the value before. Lastly, the requested L2 cache will send a WriteConfirmMessage to the client, to finish the *Write* operation.
All caches and the database will lock the key upon receiving a WriteMessage, to prevent other actors from reading or overriding the value during an ongoing *Write* operation (see Section 3.5 for details about the locking mechanism). The database unlocks the key after sending a ReFillMessage. The caches either unlock the key when receiving a ReFillMessage or if they timeout whenever the next level is not responding.

The following are possible error cases during a *Write* operation.

The L2 cache has crashed Whenever a L2 cache crashes, the client will never receive a WriteConfirmMessage. This results, that the client will timeout.

The L1 cache has crashed If the L1 cache times-out, a ReFillMessage is never received by the L2 cache. Then, the L2 cache will timeout and responds with an ErrorMessage to the client. This results that the client will cancel the *Write* operation. After the L1 cache recovers, it flushes all of its L2 clients to make sure no child L2 knows more recent data than the parent L1 cache.

The Key is locked If the key is already locked due to an ongoing *Write* operation, all actors will respond with an ErrorMessage. If no actor crashes, the ErrorMessage is eventually received by the client to cancel the *Write* operation. Otherwise, the client will timeout.

2.2.3 Critical-Read

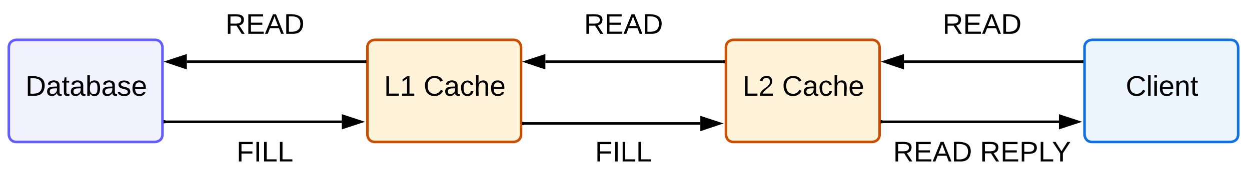

Figure 4: Message exchange during a *Critical-Read* operation

Critical-Read is the critical version of *Read*. The difference is, that a CritRead always involves the database. Therefore, the caches do not check if they can serve the request directly and it is guaranteed, that the operation will always return the latest value from the database. This operation is illustrated in Figure 4. It is shown, that both L2 and L1 caches forward the messages immediately. The database checks if the data is locked, if not, it responds with a FillMessage, like in *Read* (see Section 2.2.1). Additionally, the requested L2 cache replies a ReadReplyMessage to the client after receiving the FillMessage.

Possible error cases are the same as in *Read*, see Section 2.2.1.

2.2.4 Critical-Write

Critical-Write is the critical version of *Write*. It guarantees, that no client can see an older version during and after the operation. In addition, the database has to ensure this before it propagates the *Critical-Write*. Therefore, all caches and the database, have to agree on whether to commit or abort and to lock the value during the operation. This is called *Atomic Commitment (AC)*. The two AC algorithms introduced in the lectures are *Two-phase Commit* and *Three-phase Commit*. For *Critical-Write* a *Two-phase Commit* inspired algorithm is sufficient, because the database (coordinator) cannot crash in this scenario/environment. Therefore, an uncertain state of the actors is not expected. A special case in this scenario is, that the database takes the role of the coordinator for all L1 caches, and in addition, the L1 caches take the role of coordinator for all their child L2 caches. Therefore, the L1 cache is a coordinator and a coordinated actor at the same time.

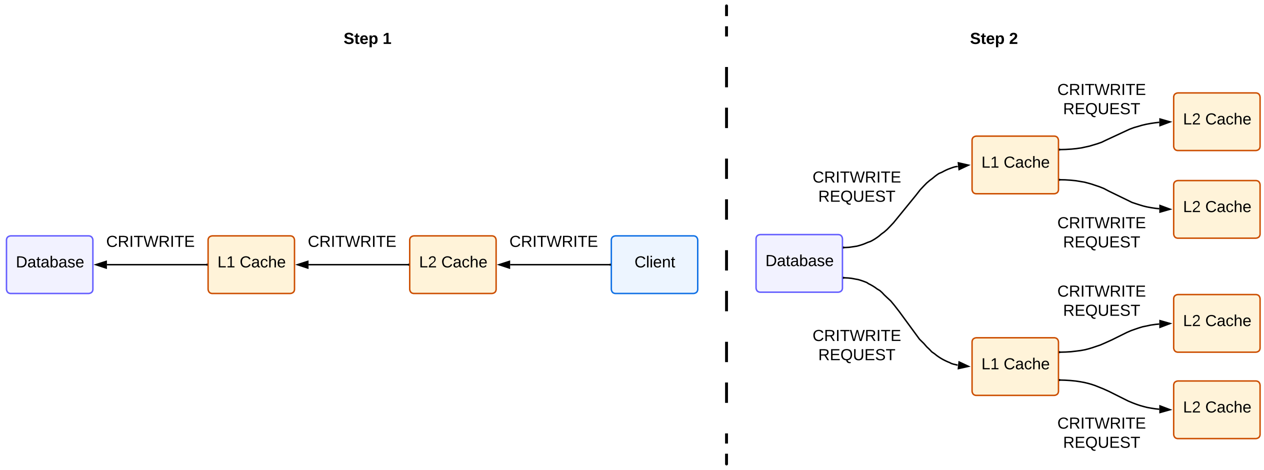

Figure 5: First phase of the *Critical-Write* operation containing step 1 and 2

Figure 5 illustrates the message exchange during the first phase of the *Critical-Write* operation. In the first step, the client instantiates a *Critical-Write* by sending a `CritWriteMessage` to an L2 cache. The L2 forwards that message to its parent L1 cache and the L1 cache forwards it to the database. Next in step 2, the database instantiates a voting by sending a `CritWriteRequestMessage` to all L1 caches. Then, the L1 caches additionally multicast the `CritWriteRequestMessage` to their child L2 caches. Upon receiving a `CritWriteMessage`, the database locks the requested key. If it is already locked, it responds with an `ErrorMessage` accordingly. L1 caches lock the key when receiving a `CritWriteRequestMessage`. If it is already locked, the L1 cache aborts locally and responds to the database by sending a `CritWriteVoteMessage` that indicates the L1 cache has voted to abort. The same is performed by the L2 caches, if the value is already locked, it will vote to abort.

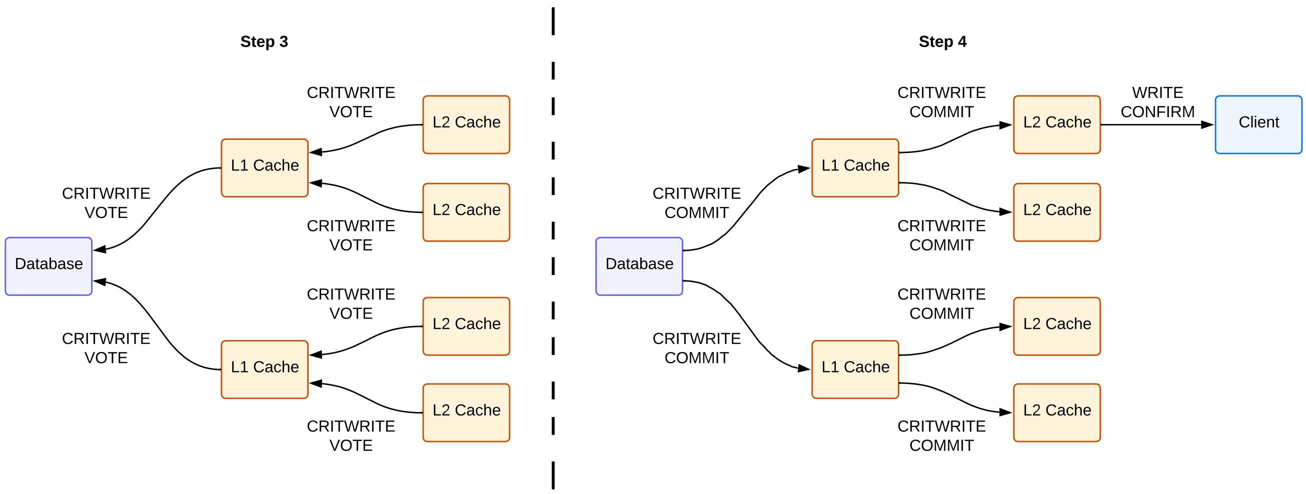

Figure 6: Second phase of the *Critical-Write* operation containing step 3 and 4

Secondly, Figure 6 illustrates the second phase of *Critical-Write*. In step 3, the L2 caches send a `CritWriteVoteMessage` to their parent L1 caches, which contains their decision. If the L2 cache can commit, it will lock the key. The L1 cache waits until it receives a `CritWriteVoteMessage` from all its child L2 caches. If at least one L2 cache has voted to abort, or if at least one L2 cache has timed-out, the L1 cache will vote to abort as well. Then, the database multicasts a `CritWriteAbortMessage`, and an `ErrorMessage` is sent to the client. The `CritWriteAbortMessage` results, that all caches abort locally. Otherwise, if all L1 caches voted to commit (then, also all L2 caches voted to commit) the database multicasts a `CritWriteCommitMessage` to all L1 caches in step 4. The L1 caches multicast the `CritWriteCommitMessage` to their child L2 caches. Upon receiving a `CritWriteCommitMessage`, a cache will update the value and unlock the key. Lastly, the requested L2 cache responds with a `WriteConfirmMessage` to the client. This finishes a successful *Critical-Write* operation.

The possible error cases during a *Critical-Write* operation are the following:

The L2 cache crashes before the voting Whenever an L2 cache crashes before it can vote, the L1 cache will never collect all votes and then times-out. This results, as stated before, that the L1 caches votes to abort and the database decides to multicast a `CritWriteAbortMessage`.

The L2 cache crashes after the voting It is not critical to the protocol if a L2 cache crashes after it has voted to commit. However, it is critical if the L2 cache crashes that is responsible to send the `WriteConfirmMessage` to the client. Then the client will never know that the operation was successful. It will timeout eventually and may retry the operation in the future.

The L1 cache crashes before the voting When an L1 cache crashes before it has voted, the database will timeout and multicasts a `CritWriteAbortMessage` to all remaining L1 caches. After that, the L2 caches belonging to the crashed L1 cache are still waiting. After the L1 cache has recovered, it will multicast a `FlushMessage` to all child L2 caches. This will delete any temporary data held by the L2 cache and also results that the L2 cache aborting the *Critical-Write* operation as well.

The L1 cache crashes after the voting If an L1 cache crashes after it has voted, it will never receive the result from the database (commit or abort). Furthermore, the L2 caches, belonging to the crashed L1, stay in the waiting state. After the L1 cache recovers, it flushes all its L2 caches (as mentioned before) to abort all ongoing operations. In addition, the L2 cache requested by the client will timeout, then abort locally, and send an `ErrorMessage` to the client.

3 Implementation

This section explains important details about the project implementation, the documentation, as well as the project structure.

3.1 Messages

Messages are an essential part of this implementation. In an Actor based environment, they are used to establish communication between actors. The following messages are used in this project:

- `CrashMessage`: Whenever an actor receives a `CrashMessage`, it simulates a crash. Afterward, the actor cannot process any further messages except a `RecoveryMessage`. This results, that all other actors, which trying to communicate with a crashed actor, will eventually timeout.
- `CritReadMessage`: This message is sent by the client to perform a *Critical-Read* (see Section 2.2.3) operation.
- `CritWriteAbortMessage`: The `CritWriteAbortMessage` is multicast by the database during a *Critical-Write* (see Section 2.2.4) if a single cache has either voted to abort or the database has timed-out.
- `CritWriteCommitMessage`: If all caches voted to commit during a *Critical-Write*, the database multicasts a `CritWriteCommitMessage` to finish the operation and to tell all caches to update.
- `CritWriteMessage`: Send by the client to start a *Critical-Write* operation.
- `CritWriteRequestMessage`: After a `CritWriteMessage` has been received by the database, it will multicast a `CritWriteRequestMessage` to instantiate the voting during a *Critical-Write*.
- `CritWriteVoteMessage`: The caches send a `CritWriteVoteMessage` to their next level which determins if the cache has voted to commit or to abort.
- `ErrorMessage`: Whenever an error has occurred (e.g. requesting an unknown key), an `ErrorMessage` is sent to the requesting actor. If no actor during the message exchange has crashed, an `ErrorMessage` is eventually delivered back to the client.
- `FillMessage`: This message is either sent by the database or by an L1 cache (only if an equal or more recent value for the key is known) upon receiving a `ReadMessage`. When a cache receives a `FillMessage`, it will update its value and the *Update-Count* for the key.
- `FlushMessage`: When an L1 cache has recovered after a crash, it multicasts a `FlushMessage` to all its child L2 caches. Upon receiving a `FlushMessage`, a cache resets all its temporary data (see Section 3.3 for details).
- `InstantiateReadMessage`: To make a client start a *Read* operation (see Section **??**), the `ActorEnvironment` (described in Section 3.2) sends an `InstantiateReadMessage` to the client. This message contains a reference to the targeted L2 cache, the key to be read, and a flag determining if this is a *Critical-Read* operation.
- `InstantiateWriteMessage`: This message has the same purpose as the `InstantiateReadMessage` but to instantiate a *Write* operation. Additionally, the message also contains the value to be written.
- `JoinDatabaseMessage`: Needed to give an actor in the environment a reference to the database actor.
- `JoinL1CachesMessage`: Send to the database to make it aware of all L1 caches.
- `JoinL2CachesReadMessage`: This message is multicast to all L1 caches to receive a reference of each of its child L2 caches.
- `JoinMainL1CacheMessage`: Used to make all L2 caches aware of its parent L1 cache.
- `ReadMessage`: A `ReadMessage` is sent by a client to start a *Read* operation.

- `ReadReplyMessage`: After an L2 cache has received a `FillMessage`, it will send a `ReadReplyMessage` to the requesting client (if this was the requested L2 cache for the *Read*) to finish the *Read* operation.

- `RecoveryMessage`: Whenever a cache crashes, it will schedule to send a `RecoveryMessage` to itself after n milliseconds (described in Section 3.3). Upon receiving this message it recovers and can participate in the environment again.

- `RefillMessage`: The `RefillMessage` is multicast by the database to make all caches update their value (if needed) after a successful *Write* operation (see Section 2.2.2).

- `TimeoutMessage`: If an actor does not receive any answer for a message it will timeout by receiving a `TimeoutMessage` (timeouts are described in Section 3.4 in detail). This message contains all the important information to cancel the unconfirmed operation if needed.

- `WriteConfirmMessage`: A `WriteConfirmMessage` is sent by an L2 cache to the requesting client to finish a *Write* or a *Critical-Write* operation.

- `WriteMessage`: To start a *Write* operation, a `WriteMessage` is sent by a client to an L2 cache.

3.2 Initiating an Environment

To initiate an environment of actors (clients, caches, and a database), the `ActorEnvironment` class exists. When an `ActorEnvironment` instance is constructed, it creates an `ActorSystem` from the Akka framework. In addition, it also instantiates all actors and makes them aware of each other. Listing 1 shows how an `ActorEnvironment` is being constructed, and how a specific client and L1 cache can be selected.

```
// Set properties
Integer numberOfL1Caches = 2;
Integer numberOfL2Caches = 4;
Integer numberOfClients = 2;

// Init ActorEnvironment
ActorEnvironment actorEnvironment = new ActorEnvironment(
    "Multi-Level-Cache",
    numberOfL1Caches,
    numberOfL2Caches,
    numberOfClients
);

// Get Client-1
ActorRef firstClient = actorEnvironment.getClient(0).orElseThrow();
// Get L1 cache (id: L1-1)
ActorRef l11 = actorEnvironment.getL1Cache(0).orElseThrow();
```

Listing 1: Instantiating an `ActorEnvironment`

Furthermore, the `ActorEnvironment` is being used to start operations (see Section 2.2 for available operations). An example of how to start a *Read* operation is shown in Listing 2. There, the selected client starts a *Read* operation for the key *9*, by sending a `ReadMessage` to the selected client. This is done using the `makeClientRead` method, that will send an `InstantiateReadMessage` to the client. Then upon receiving the message, the client sends a `ReadMessage` to the L2 cache defined in the `InstantiateReadMessage`. Additionally, methods for *Critical-Read* (`makeClientCritRead`), *Write* (`makeClientWrite`), and *Critical-Write* (`makeClientCritWrite`) also exist.

```
// Instantiate environment
ActorEnvironment actorEnvironment = new ActorEnvironment("Multi-Level-Cache", 2, 4, 1);
// Get actors and start read conversation
ActorRef firstClient = actorEnvironment.getClient(0).orElseThrow();
// Get L2 cache with the id L2-1-1
ActorRef l211 = actorEnvironment.getL2Cache(0).orElseThrow();
// Start read operation for key 9
actorEnvironment.makeClientRead(firstClient, l211, 9);
```

Listing 2: Starting a *Read* operation using the `ActorEnvironment`

3.3 Forcing Crashes and Delays

One requirement of this project is, to implement a functionality to make caches crash and to delay a message. Caches (see Section 2.1.2), are the only actors required to crash. A cache will crash upon receiving a `CrashMessage`.

```
// Make L2 cache (L2-1-1) crash after 10.000 ms
actorEnvironment.makeCacheCrash(l211, 10000);

// Make L1 cache crash after receiving a Read message and L2 wait for processing
actorEnvironment.makeClientRead(
    firstClient,
    l211,
    9,
    MessageConfig.of(
        CacheBehaviourConfig.crashAndRecoverAfter(10000), // L1 cache
        CacheBehaviourConfig.delayMessage(2000) // L2 cache
    )
);
```

Listing 3: Example on how to make caches crash using the `ActorEnvironment`

Listing 3 shows an example of, how to make a specific L2 cache crash (in the example the L2-1-1 cache). A `CrashMessage` can either be sent directly to a cache by using the `makeCacheCrash` (first example) method of the `ActorEnvironment`, or by defining a `MessageConfig` as an argument for the *make* methods (second example) introduced in Section 3.2.
In the first example, using the `makeCacheCrash` method, the L2 cache crashes immediately, and recovers after the given delay in milliseconds.
The second example introduces the `MessageConfig` class. It takes two arguments, both a `CacheBehaviourConfig`, the first representing the L1 cache and the second representing the L2 cache (caches the message is passing until it reaches the database). A `CacheBehaviourConfig` can be used to make a cache crash after it has processed/forwarded the message or wait to process/forward the received message. As shown in the example, using `crashAndRecoverAfter` the cache crashes and recovers after n milliseconds. Additionally, using `delayMessage` the cache will handle the received message after waiting for n milliseconds. Finally, using `none`, no delay or crash is performed.

In this project, crashing is simulated. This means, that caches will ignore any incoming messages after they have crashed. However, it is also required for caches to recover after a certain interval n. To achieve this, a cache schedules to send a `RecoverMessage` to itself after n milliseconds, whenever it receives a `CrashMessage`.

Another important aspect of crashing is flushing. As mentioned in Section 2.1.2 L1 caches make sure, that no child L2 cache knows any more recent data than the L1 cache. To achieve this, the crashed L1 cache will multicast a `FlushMessage` to all its child L2 caches after it has recovered. This ensures, that the L2 caches remove all of their temporary data.

3.4 Timeout

To implement the operations defined in Section 2.2, at certain points during the message exchange, actors are required to timeout. This is required to ensure liveness in this scenario. Therefore, to make sure that no actor remains trapped in a state because of another crashed actor or a different error, actors are required to timeout and cancel the ongoing operation in that case.

Whenever required to timeout, an actor schedules to send a `TimeoutMessage` to itself after n seconds, when it sends a message to another actor. If an answer has been received during the n seconds, the actor marks the message as confirmed and ignores the `TimeoutMessage`. Otherwise upon receiving a `TimeoutMessage`, if the operation is still unconfirmed, the actor will timeout and handle the situation as expected by either sending an `ErrorMessage` to the client or resetting the configuration for the ongoing operation and forcing everybody else to timeout as well.

3.5 Storing and Accessing Data

Actors are required to store data, either temporary or persistent. Caches store data temporarily to simulate that they lose all data upon crashing. The database and clients are not supposed to crash and therefore store data persistently. Furthermore, during *Write* and *Critical-Write* operations, it is important to have a locking mechanism that prevents other actors from either reading or writing the same key at the same time. This is required to prevent conflicts. Furthermore, it is also important to ensure, that newer data is never overwritten by older data. To ensure these properties, the `DataStorage` class as well as the *Update-Count* have been implemented are further explained in this section.

3.5.1 DataStorage Class

The `DataStorage` class is a custom class that provides CRUD (*Create Read Update Delete*) operations over a collection of key-value pairs. In addition to that, it can lock the value of a specific key. When a key is locked, it cannot be read or overwritten. If an actor tries to read or overwrite a locked value, an error is thrown. The actor is then supposed to act in a correct matter by either timing-out or responding with an `ErrorMessage`. Furthermore, the `DataStorage` also stores an additional *Update-Count* value for each pair.

3.5.2 Update-Count

The *Update-Count* determines how often a key-value pair has been written/updated. It will be increased by *1* on every write action to that key. The caches use the *Update-Count* to decide whenever they need to forward a *Read* operation to the next level or if they can respond immediately.
In practice, whenever a client instantiates a *Read* (and its critical version) operation, it appends it's known *Update-Count* of that key to the message. On receiving, a cache checks if its *Update-Count* is at least equal-to or higher-than the received *Update-Count* to serve the request immediately. Otherwise, it has to forward the message to the next level, to receive the most recent value.

3.6 Atomic Commitment

Atomic commitment is required during the *Critical-Write* message exchange to decide if all actors can commit. A peculiarity in this scenario is, that two coordinators exist, the database that coordinates all L1 caches, and the L1 caches that coordinate its L2 caches. Therefore, to reduce the complexity of the implementation both coordinators implement the `Coordinator` interface and own an `ACCoordinator` instance that handles the functionality for census between all coordinated actors.

3.7 Client-Centric Consistency

This project requires achieving *Eventual Consistency*. However, this project can achieve stronger consistency by providing all four *Client-Centric Consistency* models. This section explains, how the *Client-Centric Consistency* models are achieved.

3.7.1 Monotonic Reads

If a cache reads the value of x, it will never read an older value of x at a later time, for example by contacting a different L2 cache.
This client-centric consistency model is achieved using the *Update-Count*. It prevents the client reads an older value because a cache will never return a value with a smaller *Update-Count* than the one given by the client. Furthermore, when a `ReadMessage` or `CritReadMessage` reaches the database, it will always respond the most recent value.

3.7.2 Monotonic Writes

This consistency model assures that, if a client performs two writes *W1* and *W2* sequentially, *W1* happens before *W2*, the state of *W1* is always available before the state of *W2*.
A client can only perform one write operation at a time. If two writes are performed at the same time (e.g. using the `makeClientWrite` method, see Section 3.2) the client throws an error.

3.7.3 Read Your Writes

This consistency model assures that, once a client writes/updates a value for key x, any successive read of x by the same client will never return an older value.

Both the *Write* and the *Critical-Write* operations make sure, that after the value was updated at the database, no cache (that has accessed the key previously) keeps an older value. *Write* uses `RefillMessages`, while *Critical-Write* depends on an *Atomic-Commitment* protocol. Furthermore, the *Update-Count* additionally prevents reading an older value, as mentioned before.

3.7.4 Writes Follow Reads

Once a client reads x, which was written by *W1*, and successively performs another write *W2*, then the effect of *W2* must be available after *W1*. The purpose is that any writes should start from the state of the read, and it is not possible to change the past.
It is also achieved because a client can only perform one write operation at a time. Therefore, it prevents that a client cannot perform two writes, *W1* and *W2* where *W1* happens before *W2* at the client, that results that *W2* may arrive at the database before *W1*.

3.8 Logging

One critical feature of this project is its custom logging. It is mostly used to track the message exchange during operations and is therefore critical for testing.

```
TIME      | ID       | ACT | MESSAGE TYPE       | INFO
--------- | -------- | --- | ------------------ | ------------------------------
00:01,104 | L2-1-4   | REC | JOIN               | L1 Cache of 1
00:01,104 | L2-1-2   | REC | JOIN               | L1 Cache of 1
00:01,104 | L2-2-1   | REC | JOIN               | L1 Cache of 1
00:01,104 | L2-1-1   | REC | JOIN               | L1 Cache of 1
00:01,106 | L2-1-4   | REC | JOIN               | Database of 1
00:01,104 | L1-1     | REC | JOIN               | Database of 1
00:01,106 | L2-2-1   | REC | JOIN               | Database of 1
00:01,106 | L2-1-1   | REC | JOIN               | Database of 1
00:01,106 | L1-1     | REC | JOIN               | L2 Caches of 4
00:01,107 | L2-2-2   | REC | JOIN               | L1 Cache of 1
00:01,104 | L1-2     | REC | JOIN               | Database of 1
00:01,107 | L2-2-3   | REC | JOIN               | L1 Cache of 1
00:01,107 | L2-2-4   | REC | JOIN               | L1 Cache of 1
00:01,107 | L2-2-2   | REC | JOIN               | Database of 1
00:01,108 | L1-2     | REC | JOIN               | L2 Caches of 4
00:01,108 | L2-2-3   | REC | JOIN               | Database of 1
00:01,108 | L2-2-4   | REC | JOIN               | Database of 1
00:01,106 | L2-1-2   | REC | JOIN               | Database of 1
00:01,105 | Database | REC | JOIN               | L1 Caches of 2
00:01,105 | L2-1-3   | REC | JOIN               | L1 Cache of 1
00:01,109 | L2-1-3   | REC | JOIN               | Database of 1
00:01,112 | Client-2 | REC | JOIN               | L2 Caches of 8
00:01,112 | Client-1 | REC | JOIN               | L2 Caches of 8
00:03,106 | Client-1 | REC | INIT-READ          | key: 9, is-critical: false
00:03,106 | Client-1 | SEN | READ               | key: 9, uc: 0
00:03,117 | L2-1-1   | REC | READ               | key: 9, msg-uc: 0, actor-uc: 0, is-locked: false, is-older: false, is-unconfirmed: false
00:03,118 | L2-1-1   | SEN | READ               | key: 9, uc: 0
00:03,118 | L1-1     | REC | READ               | key: 9, msg-uc: 0, actor-uc: 0, is-locked: false, is-older: false, is-unconfirmed: false
00:03,119 | L1-1     | SEN | READ               | key: 9, uc: 0
00:03,119 | Database | REC | READ               | key: 9, msg-uc: 0, actor-uc: 4, is-locked: false, is-older: false, is-unconfirmed: false
00:03,119 | Database | SEN | FILL               | key: 9, value: 384, uc: 4
00:03,119 | L1-1     | REC | FILL               | key: 9, new-value: 384, old-value: -1, new-uc: 4, old-uc: 0
00:03,119 | L1-1     | MUL | FILL               | key: 9, value: 384, uc: 4
00:03,120 | L2-1-1   | REC | FILL               | key: 9, new-value: 384, old-value: -1, new-uc: 4, old-uc: 0
00:03,120 | L2-1-1   | MUL | READ-REPLY         | key: 9, value: 384, uc: 4
00:03,120 | Client-1 | REC | READ-REPLY         | key: 9, new-value: 384, old-value: -1, new-uc: 4, old-uc: 0
```

Listing 4: Log of a *Read* operation

Listing 4 shows an example of the log during a *Read* operation (see Section 2.2.1). As seen, there are four different columns. First, the time, second the ID of the actor who is currently performing an action. Next, the `ACT` (abbreviation for *Action*) column, that determines what action the actor is currently performing. Possible actions are Error (`ERR`), multicast (`MUL`), receive (`REC`), and send (`SEN`). The fourth column represents the type of the message that is currently processed (see Section 3.1 for possible messages). Lastly, the fifth column shows additional information about the operation, that is important for testing purposes.

4 Problems

This section introduces problems and their solutions encountered during the implementation of the project.

4.1 Testing

One problem during this project was how to test the message exchange. Overall, this is a very important and difficult task, because message exchange happens in milliseconds. As mentioned, this project was implemented using the *Akka Classic* toolkit, which provides a testing suite[2]. However, the *Akka Classic* testing suite seems to be unmaintained and uses deprecated dependencies. Therefore, the decision was, not to use this test suite. Instead, a detailed logging functionality was implemented, introduced in Section 3.8. It has been proven to be very useful and helpful in general to find bugs in the protocol implementation of the operations.

4.2 Timeouts

Another problem was choosing meaningful timeout delays for various actors during different scenarios. To demonstrate this problem, one example is, when an L2 cache crashes during the voting of a *Critical-Write* operation (see Section 2.2.4). The problem is, whenever an L2 cache crashes and is not able to vote, in the ideal case, the L1 cache times-out before the database and all remaining L2 caches. If the database times-out before the L1 cache, it will multicast a `CritWriteAbortMessage` to all L1 caches twice. First, because the database times-out during the voting, and secondly, because the L1 cache will timeout afterward and sends a message indicating to abort to the database. If the L2 caches timeout at first, they will abort locally. Still, the protocol will finish correctly eventually, because the client will timeout as well and all caches abort. However, this is not the ideal case. This example is one of the multiple scenarios encountered during this project, where fine-tuning timeouts is a significant part of the implementation. No overall working solution for all scenarios has been discovered during the implementation. Finding suitable timeout delays requires testing the specific operation.

5 Conclusion

To conclude, the introduced system works and provides Client-Centric Consistency. All operations and their critical version have been implemented and tested for correctness. A version of *Critical-Write* was introduced that is strongly inspired by the Two-Commit protocol.

6 References

I hereby assure that the only resources used for this report are the lecture resources provided during the course *Distributed Systems 1 2021/22* (slides, recordings, and laboratories) by Prof. Picco and his teaching assistants.

[2] `https://doc.akka.io/docs/akka/current/testing.html` (Accessed: 02-04-2023)