Distributed Sysytems 1 Project

Multi-Level Distributed Cache

Clocchiatti Jacopo[1] and Vicentini Leonardo[1]

[1]University of Trento

January 14, 2024

1 Introduction

In today's data-driven world, efficient and scalable data access is a critical concern for applications and services. To address the challenges associated with high-demand data retrieval and storage, we have embarked on the development of a distributed cache system. The primary goal of this project is to alleviate congestion at the main database, enhance performance, and ensure data consistency in a distributed environment.

This distributed cache system is designed to support multiple clients that read and write data items stored in a database. However, it introduces an additional layer of cache nodes, strategically arranged in a two-tier tree topology. This hierarchical structure not only improves data retrieval efficiency but also ensures that write operations maintain eventual consistency across the cache layers. We can see an example of the (partial) structure in the following figure.

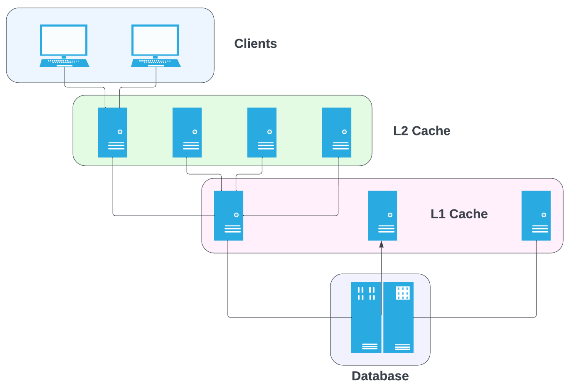

Figure 1: System diagram (partial view)

The system is optimized for read operations, ensuring that frequently accessed data is readily available to clients. By caching highly-requested items and handling most client requests independently, we reduce the burden on the main database and significantly enhance the overall system performance. However, for write operations, we follow a different approach. Writes are routed to the main database to maintain the integrity of the data, and updates are propagated through the cache layers to ensure **eventual consistency**.

In this project, we implement the distributed cache system using Akka actors. Clients, caches, and the main database are all modeled as actors within the Akka actor system. The choice of Akka provides a robust and concurrent framework for building such a distributed system, where each actor encapsulates specific functionality and communicates asynchronously.

Clients interact with the system through the cache nodes, which are responsible for processing read and write requests. These requests include basic operations like *Read* and *Write*, as well as critical variants, *CritRead* and *CritWrite*, each with specific guarantees.

Additionally, the system considers the possibility of cache crashes and implements crash detections based on timeouts.

In summary, this project presents the design and implementation of a distributed cache system that addresses the challenges of data access in a distributed environment. By utilizing Akka actors and a hierarchical cache structure, we aim to provide efficient, high-performance, and eventually consistent data access for clients. This report will delve into the technical details of the system's architecture, operations, crash recovery, and critical operations guarantees.

2 Requirements

The project includes clients, caches and one database, all being Akka actors. The initial tree structure is predefined and consist of two level of caches: L2, in direct interaction with clients, and L1, parents of L2 and directly connected to the database. A client contacts a given L2 cache with a request. All request types refer to a specific item, identified by a key. Write requests also include the new value for the item. Requests fail if the item is not accessible at the time of operation. The cache then replies with an appropriate message to the client. Caches may fail so the system should implement a simple crash detection algorithm based on timeout. A client detecting a crashed L2 will select another L2 cache and redirect its request. An L2 that detects its L1 parent has crashed will select the main database as its parent. The possible operations are four:

- ***Read***. When an L2 cache receives the request, it responds immediately with the requested value if it is found in its memory. Otherwise, it will contact the parent L1 cache which will do the same, contacting the database if needed. Responses follows the path of the request backwards, until the client is reached. On the way back, caches save the item for future requests.

- ***Write***. The request is forwarded to the database, that applies the write and sends the notification of the update to all its L1 caches. All L1 caches propagate it to their connected L2 caches. The update is potentially applied at all caches, which is necessary for eventual consistency. However, only those caches that were already storing the written item will update their local values.

- ***CritRead***. Fetches the current value stored in the database for a given key. Therefore, contrary to a *Read*, the request is forwarded to the database even if the L2 or the L1 cache already hold the item.

- ***CritWrite***. The request is forwarded to the database as in *Write*. However, before the write is applied, the database must ensure that no cache holds an old value for the written item. No client should be able to read the new value and then the old value, from any cache. Once the database has ensured the cached items have been cleared/updated, it finalize the update.

2.1 Assumptions

We highlight some assumptions about the system and the implementation. Starting from the data held by the system which consist of integer values, each attached to a globally-unique key (a key-value store). We assume that the node of the system do not change over the execution of the protocol. We assume **links are FIFO and reliable**, caches have unlimited memory for storing items. A single cache could potentially deal with multiple requests at one time. Cache nodes may crash, entering a "crashed mode" in which they ignore all incoming messages and stop sending anything. In addition, all the items are removed from the crashed caches but they retain information about the system, like topological information. We assume only one crash at a time, system-wide. Client actors and the database actor do not crash. Before sending a new request, a client waits for its previous one to finish.

3 Architectural Choices

3.1 Actors structure

All the actors have some common attributes. These common attributes are: the id of the actor, the timeouts data structure (an hash map with the type of timeout and its value in seconds).

3.1.1 Client

Clients have information only about L2 caches, because if they receive a timeout from the L2 cache that they're connected to, they need to know to which L2 cache they need to try to connect. Moreover they keep a list of all the operations they performed during the execution.

3.1.2 Database

The database actor, besides the key-value store, has information about all caches and also data structures for the support of *CritWrite* operation (described later).

3.1.3 Cache

Caches are by far the most complex actor and have many additional information, among which:

- the type of cache they represent (L1 or L2).
- a list of Requests in which the cache is and was involved.
- additional information of support for the crash recovery procedure.
- an hash map containing temporal data waiting to be confirmed to be written in the context of a *CritWrite* operation.
- additional information of support for *CritWrite* operations.

3.2 Configuration and Set-up

We implemented a way to configure the system based on a YAML file. In this file there are specified information to build system, the maximum number of each actor type (clients and caches), a fixed number if we want complete control, all the timeouts durations. The system has two "modes" to be built, a custom mode and an automatic mode. In the custom mode the system will create the specified number of caches and clients. In automatic mode there's another option that set the system to be structured in a balanced or unbalanced way. If the system is set to be balanced all L1 caches will have the same number of L2 cache as children and similarly all L2 caches will have the same number of clients connected to them. If the system is unbalanced those number varies for each actor (every L1 cache will have a random number bounded to the specified max number and so on). Lastly, a master actor with information about the whole system is added.

3.3 Monitoring and Interactivity

To monitor and interact with the entire system we choose to follow an approach based on a **web server**, using the specific Akka component. We can send certain operations to execute by sending a request to this server that interact with the system to concretely execute said operations. Two of these operation are the **crash** and the **recovery** of caches.

Another operation is the **consistency check**. It is executed by the master that has information about the whole system. It send a requests for all their data to the database and to all the caches. It takes the information of the database as ground truth and checks if the information of the caches are consistent with the database and if they're not it highlights which cache is inconsistent and for which value. All 4 operations (*Read*, *CritRead*, *Write*, *CritWrite*) could be started for clients from the web server with a specific endpoint.

The list of available endpoints is available on the "readme.md" file of the project.

3.4 Operations implementations

Each client can fire *Operations*, which are translated into various *Requests* along the tree of caches. Each cache could be responsible for multiple *Requests* at the same time. A *Requests* is considered fulfilled when the involved cache send a response (either successful or failed).

3.4.1 Read

When a client starts a *Read* request it asks to the L2 cache it is connected to for the key it needs. If the L2 cache contains that value it directly responds to the client otherwise it redirect the request to its parent L1 cache adding itself to the **path** field of the message. The L1 cache works in the same way, with the only difference being that if it does not contain the needed value it will ask the database for the value. The database checks if the value exists, if it does not it will respond with an error otherwise it will send the response with the requested value. In the reverse path (recovered from the path field of the request message) the visited caches will add the value to their internal storage.

3.4.2 Write

When a client starts a *Write* request, the message will reach the database passing through the caches populating the *path* field. The database will write the new value and will send the

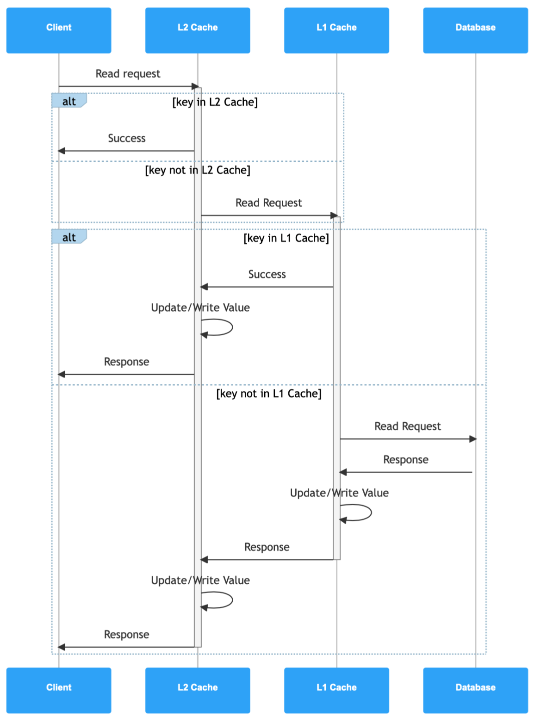

Figure 2: Read operation sequence diagram

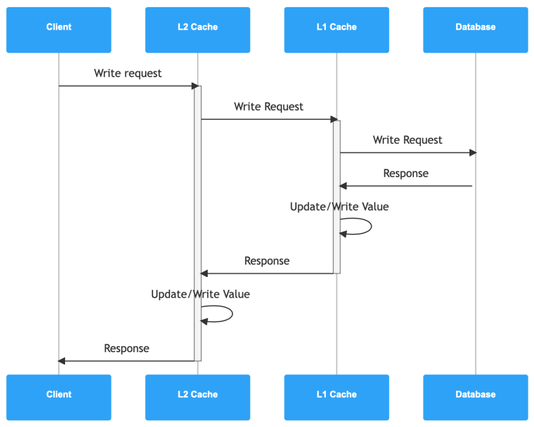

Figure 3: Write operation sequence diagram

successful response to the client leveraging the *path* field from the request message. After that the database will broadcast an *update* value to all the caches connected to it. As a matter of fact, every cache that receives the *FillMsg* checks if the updated key is present in their local storage, if there is, then they update the value in their local storage and they pass the update on their child caches (not to clients). This is a **tradeoff**, as it was assumed that if the value is not present in the current L1 cache, it is not present in the L2 cache children.

3.4.3 CritRead

When a client starts a *CritRead* request, said request will be **passed directly to the database** with the *path* field of the request populated by the caches from which it passes through (even if the caches hold the value). The database then checks if the requested key exists, if it does not then it responds with an error, otherwise it responds with the corresponding value. The caches "traversed" by the response will add or update the key-value pair to their local storage.

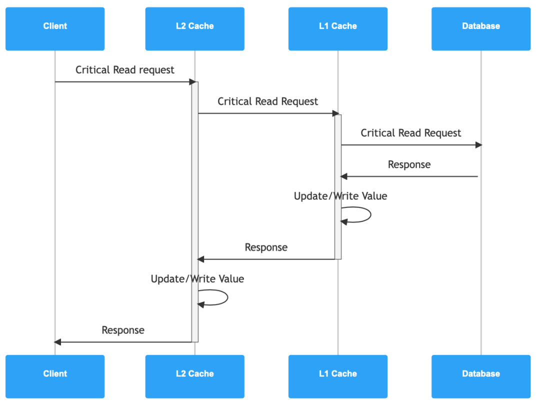

Figure 4: Critical Read operation sequence diagram

3.4.4 CritWrite

CritWrite is the most complex operation of the system. In particular, aside from the conventional request and response it leverages also the following (specific) messages:

- *ProposedWriteMsg*
- *AcceptedWriteMsg*
- *ApplyWriteMsg*
- *ConfirmedWriteMsg*

When a client starts a *CritWrite* request the message will reach the database passing through the caches populating the *path* field just like a standard *Write* operation. To explain how *CritWrite* was implemented, it is useful to go through every step of the final phase in details, highlighting each actor involved:

1. [DB] Checks if the key is already present in *ongoingCritWrites*.
   (a) [DB] If yes, the *CritWrite* **will not be accepted**.
   (b) [DB] Sends a refused *CriticalWriteResponseMsg*.
2. [DB] Otherwise, adds into *ongoingCritWrites* the key requested.
3. [DB] Sends *ProposedWriteMsg* to all connected caches.
4. [DB] Creates a set of caches that are expected to accept the *CritWrite*, starting from the L1 caches.
   (a) [DB] Sends the message also to all L2 caches that are connected directly with the database (due to previous crashes), if any.
5. [DB] Sets a timeout ("accepted_write"), waiting for all expected *AcceptedWriteMsg*.
6. [L1 cache] Receives *ProposedWriteMsg*.
   (a) [L1 cache] Writes into *tmpWriteData* structure the key-value pair.
   (b) [L1 cache] Forwards to all children the *ProposedWriteMsg*.
   (c) [L1 cache] Waits for all *AcceptedWriteMsg* of children.
7. [L2 cache] Receives *ProposedWriteMsg*.
   (a) [L2 cache] Writes into *tmpWriteData* structure the key-value pair.
   (b) [L2 cache] **Clears** from the key-value store the involved key-value.
   (c) [L2 cache] Sends to parent an *AcceptedWriteMsg*.
8. [L1 cache] Receives *AcceptedWriteMsg*: checks every time if all children caches have responded with this message for that specific key.
   (a) [L1 cache] If yes, **clears** from the key-value store the involved key-value. Clearing only when all children respond is useful to avoid an "unnecessary data loss" for an aborted *CritWrite* due to the children of this specific cache.

   (b) [L1 cache] Sends to database an *AcceptedWriteMsg*.
9. [DB] Receives *AcceptedWriteMsg*: checks every time if all children caches have responded with this message for that specific key.
   (a) [DB] If yes, we are **guaranteed all caches have cleared the old value**, **DB adds new key-value pair** to its key-value store.
   (b) [DB] Sends *ApplyWriteMsg* to all children caches.
   (c) [DB] Sets a timeout ("confirmed_write"), waiting for all expected *ConfirmedWriteMsg*.
   (d) [DB] If instead, the **"accepted_write" timeout is reached**, database starts the **abort procedure**.
10. [L1 cache] Receives *ApplyWriteCache*.
   (a) [L1 cache] **L1 cache adds involved key-value pair** to its key-value store.
   (b) [L1 cache] Removes key-value pair from *tmpWriteData*.
   (c) [L1 cache] Forwards *ApplyWriteMsg* to all children caches.
11. [L2 cache] Receives *ApplyWriteMsg*
   (a) [L2 cache] **Adds key-value pair** to its key-value store.
   (b) [L2 cache] Removes key-value pair from *tmpWriteData*.
   (c) [L2 cache] Sends to parent a *ConfirmedWriteMsg*.
12. [L1 cache] Receives *ConfirmedWriteMsg*: checks every time if all children caches have responded with this message for that specific key.
   (a) [L1 cache] If yes, sends *ConfirmedWriteMsg* to database.
13. [DB] Receives *ConfirmedWriteMsg*: checks every time if all children caches have responded with this message for that specific key.
   (a) [DB] If yes, sends an **accepted** *CriticalWriteResponse* back to the requesting subtree (leveraging the *path* as always).
   (b) [DB] If instead, the "confirmed_write" timeout is reached, we anyway proceed with an **accepted** *CriticalWriteResponse*: **we are guaranteed that no old key-value pair remained in any cache**. Probably some caches, due to crashes will not hold the new value but this is not a strict requirement of *CritWrite*.

More assumptions or edge cases. Concurrent *CritWrite* operations are supported but only with different keys: database will refuse a *CritWrite* request with a key involved at the moment, checking the *ongoingCritWrites* data structure.

During *CritWrite*, **timeouts are fired only by the database**: "accepted_write" and "confirmed_write". In future versions of the system, caches could also deal with timeouts of the *CritWrite*. For instance, L1 caches waiting for *AcceptedWriteMsg* from L2 caches children could be such a case.

Crashes during *CritWrite*. A crash that precludes the arrival of one of the *AcceptedWriteMsg* to the database leads to the abort procedure. In this context, as can be seen following the steps in the previous section, the system experience a so-called **"unnecessary data loss"** since some caches have cleared the key-value pair involved in the *CritWrite* that at the end is not concluded but aborted.

Abort procedure. If database reaches an "accepted_write" timeout, an abort procedure is put in place. At this point: database did not add the key-value pair to its data, some caches have accepted the *CritWrite* (they have *tmpWriteData* for that key), and they will have cleared the value on data hashmap (unnecessary data loss). We need to clear *tmpWriteData* for that key to allow future *CritWrites* on that key which otherwise caches would consider the previous *CritWrite* still ongoing. Therefore, database sends a *DropTmpWriteDataMsg* to all connected caches. A L1 cache receiving said message will remove from *tmpWriteData* the key-value pair and will forward the message to L2 cache children. In addition, database will clear operation-specific data structures and send a **refused** *CriticalWriteResponseMsg* to the requesting child. Note that during a cache crash, *tmpWriteData* is cleared as well as standard data.

Some tradeoffs. Since the amount of messages used in a *CritWrite* operation is quite heavy and because "unnecessary data loss" could occur during aborted *CritWrites*, to speed up future reads (system should be read-optimized), it has been decided to update the value involved in the *CritWrite* in any case, even if the cache did not previously hold the value.

It could have been decided to store the *CritWrite* key-value pair in the tmpWriteData structure on the path that is getting up to the database, to save messages. It has been chosen that the value must not be stored on the way up because a concurrent *CritWrite* for the same key could be on the way up on another path. As it is currently implemented, the first *CritWrite* request that arrives to the database will be performed.

Due to lack of time, the protocol of the *CritWrite* was simplified from the initial high-level ideas of the team. As a matter of fact, crashes **before the gathering of all AcceptedWriteMsg** by the database will always lead to an abort of the *CritWrite* operation.

In some sense, the steps after the addition of data by database could seem not needed. These "extra steps" were added to the protocol to counterbalance the "unecessary data loss" that occurs on an aborted *CritWrite*. Due to the simpler and less tighter protocol (timeouts managed only by database), this situation could potentially occur many times in a real scenario.

The "clear data" step on the caches makes possible the fact that on a "confirmed_write" timeout, database has already the **guarantees** that no old value for the involved key can be seen by Clients. Therefore the database can send back **anyway** an approved *CriticalWriteResponseMsg* back to the requester, since the key-value pair is cleared or updated with correct value. If all children answer with a *ConfirmedWriteMsg*, the procedure will simply be quickened in comparison to waiting for said timeout.

3.5 Crash system and Recovery

3.5.1 Crashes

L1 and L2 caches can be put in "Crash mode", that is, ignoring all incoming messages until a RecoverMsg, via a specific endpoint of the HTTP server (or by strategically placing the *crash()* function in code). When a crash occurs, all data and previous requests information are deleted. However, caches keep their topological knowledge, for instance a L2 cache keeps the reference of its parent.

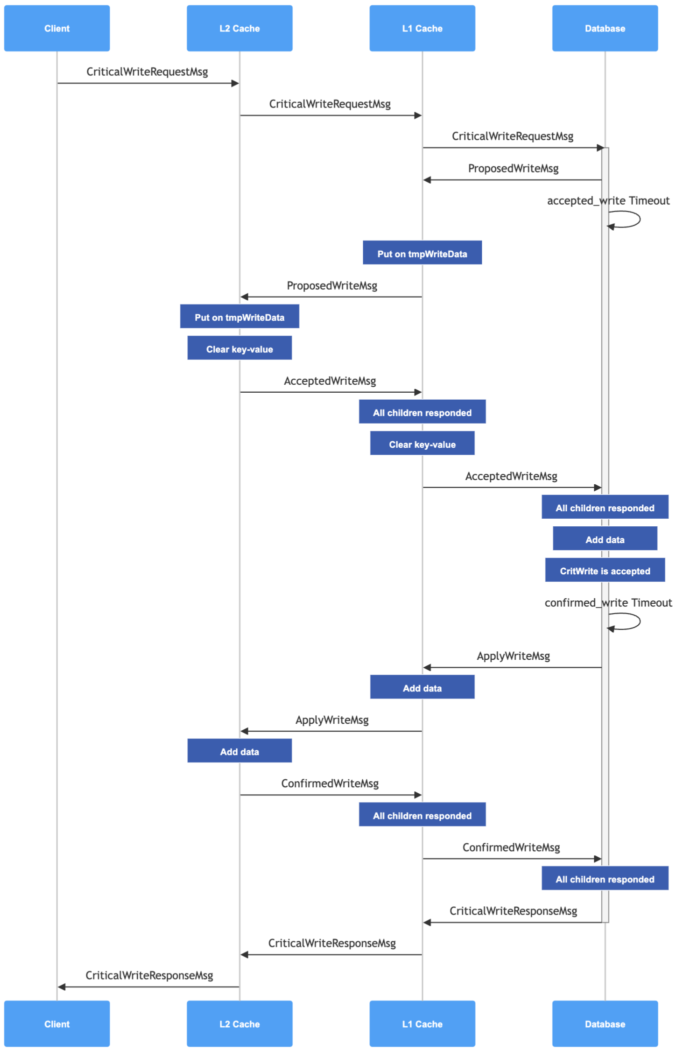

Figure 5: Critical Write operation sequence diagram

3.5.2 Timeouts

In order to detect crashes of caches, clients and caches leverages **timeouts**, which are basically messages to be sent to themselves after a specific timespan.

Clients. When a *TimeoutMsg* is received by a client: we first check if the specific *Operation* was concluded in the meantime and so if the timeout must be ignored. In order to perform such control, requestId field is leveraged. If the Operation is still ongoing (not finished) then we check if the timeout should be skipped due to previous communications by L2 cache parent (*TimeoutElapsedMsg*). This is the case when a L2 cache timeout its parent L1 cache and so tells the client to wait more and to skip the very next timeout, like explained in the next section. Therefore when receiving a **TimeoutElapsedMsg** another timeout will be started by the client, this is not a problem since *onTimeout()* will check the requestId.

The *retryOperation()* function is used to retry to perform the operation when a client connects to another L2 cache (due to previous timeout on L2 cache).

Clients will not receive double response messages since presumably (we assume that) the L1 cache parent (mandatory transition point) is crashed after sending a request message to the database (and not yet recovered). So the first response is **lost** since it will be sent to a presumably crashed cache. Therefore, another response to a L2 cache is fine.

L2 Caches. If a L2 cache timeouts on a L1 cache parent, then it tries to connect directly to the database. L2 cache also sends a *TimeoutElapsedMsg* which is used by the L2 cache to inform the client to wait more and to **skip a timeout** (and starts a new one). In that way the client does not wrongly try to connect to another L2 cache. It would be wrong because the problem (crash) is upon the upper L1 cache and the L2 cache child has experienced a timeout against it. The L2 cache simply needs more time to connect to the database and to retry the request.

When said L2 cache receives a *ResponseConnectionMsg* it means that now the L2 cache is directly connected with the database. Then, the *retryRequests()* function is called to retry to forward not yet fulfilled requests. Note that could be more than one since unlike the clients, caches do not perform only 1 requests at time.

In the cases of *Read* and *CritRead*, no particular problems arise: L2 cache forward the request to the database, keeping in mind that the database could have already been asked the same request from the crashed L1 cache (in some edge cases), the response however never arrived to the L2 cache (due to the L1 cache crash), so asking again is correct.

In the case of *Write*: if the database has already been requested the same request from the crashed L1 cache, we overwrite the same value in the database, which is not an issue.

Finally, in the *CritWrite* case: if the database has already received the same request from the crashed L1 cache, it will deal by itself with the new message sent during *retryRequest()* procedure and send a refused *CriticalWriteResponseMsg*, since the key is already present in the *ongoingCritWrites* data structure, otherwise, it will start the normal procedure of *CritWrite*.

As clients, L2 caches **ignore timeouts** for already fulfilled request.

L1 Caches. L1 caches do not fire timeouts during the 4 *Operations*, due to assumptions (database does not crash). But they leverage a specific timeout ("response_data_recover") on a step of the recovery procedure as further described in the following section.

Database. Database could potentially start 2 timeouts, during the *CritWrite* operation: "accepted_write" and "confirmed_write".

Upon receiving a **"accepted_write"** timeout database checks 2 cases: if requests is already fulfilled (similarly to clients and L2 caches) or when a *CritWrite* is **known to be accepted** but not yet completed (confirmed) using the specific data structure *acceptedCritWrites*. If one of the above described cases is true, the "accepted_write" timeout is ignored. If one L1 cache do not respond to database with an *AcceptedWriteMsg*, it could mean 2 things: L1 cache is crashed or one of the children L2 cache is crashed. In any of these two cases, if "accepted_write" timeout is reached (and the cases to ignore do not hold), the previously described abort procedure is put in place (refused *CritWrite* and remedial actions).

Upon receiving a **"confirmed_write"** timeout, database approves anyway the *CritWrite* because the protocol ensures that no old value for the specified key is stored in any cache, as described in the operation's steps.

If the "confirmed_write" timeout is not reached before finishing the operation it is the case that all *ConfirmedWriteMsg* are collected: we can speed up the sending of *CriticalWriteResponseMsg* to the requesting client using *path*. This timeout is just used to finalize the operation, since guarantees are already satisfied when database adds data, so **before** setting the "confirmed_write" timeout. Then, as always, when the "confirmed_write" timeout is reached anyway, it is ignored since operation is finished.

3.5.3 Recovery

In order to recover from a crash, a *RecoverMsg* must be sent to the crashed cache. Different recovery approaches were chosen based on the type of the cache involved.

L1 cache recovery procedure

1. L1 cache crashes, loses all data.
2. L1 cache recovers.
3. L1 cache **asks all its children** for the keys they hold.
4. L2 caches **respond with their own key-value pairs**, with the exception of those keys that are currently involved in a not yet completed *CritWrite* operation.
5. When all responses from children arrive or a specific timeout ("response_data_recover") is reached, L1 cache **aggregates those keys** in a set called *recoveredKeys*.
   (a) This step is done because those keys are the one that previously L1 cache held, being that L1 cache a mandatory transition point in the operations that filled L2 caches.
   (b) If not all the L2 cache children respond to the key request (and so a specific timeout is fired), at the end the recovering L1 cache will have only a subset of its data before the crash.
   (c) In addition, the *recoveredValuesOfSources* data structure on the L1 cache holds the key-value pairs for each child contacted in the previous step.
6. L1 cache asks the database (source of truth) for "fresh" values for the *recoveredKeys*
7. Database sends back to requester L1 cache the requested key-value pairs.
8. L1 cache stores the data received and therefore holds the whole set of key-value pairs prior to the crash (or a subset if some children crash during L1 cache recovery procedure).

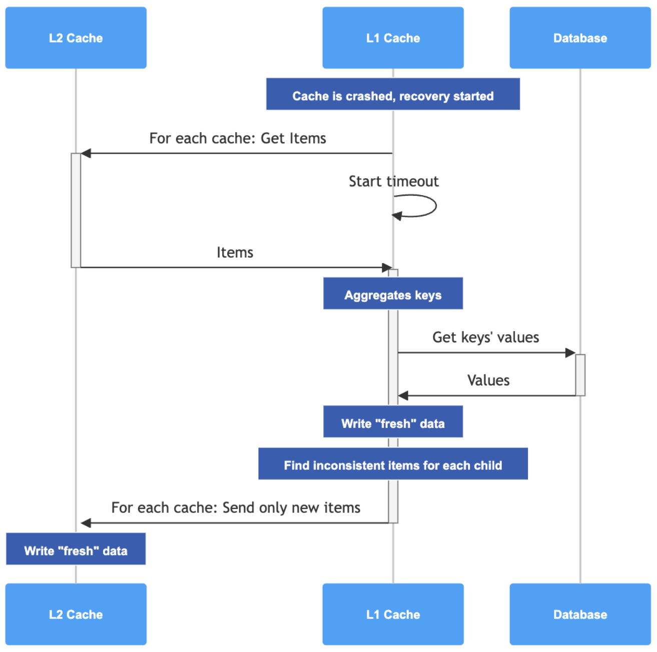

Figure 6: L1 cache Recovery operation sequence diagram

9. L1 perform an **extra step** to update the data of L2 caches children if needed.
   (a) The *recoveredValuesOfSources* data structure is leveraged to loop through each (non-crashed) child and and each key-value pair they held to look for inconsistencies with respect to fresh data coming from the database.
   (b) An *UpdateDataMsg* containing only inconsistent key-value pairs is sent to children, if at least one inconsistency is found (to save bandwidth).

The goal of this approach is to move toward **eventual consistency**. L2 caches may become "isolated" with stale data for a certain amount of time: without the connection to a L1 cache parent and without new requests (that would eventually lead to a direct connection with the database). With the (eventual) recovery of the L1 cache parent, the related subtree is subject to a beneficial **"data refresh"** with data values coming from the source of truth for keys previously held.

For instance, a use case is a L1 cache crash before receiving the *FillMsg* of a *Write* operation. The related subtree, which previously held that specific key involved, is not filled but the *onRecover()* function of L1 cache will refill the subtree with "fresh" data.

A different strategy may have used fewer message exchanges, with the request of the whole data dump from the recovering L1 cache to the database. The aforementioned approach is not ideal if the database is very large (and so the data dump time), as the L1 cache will fill up with a large and not specific amount of key-value pairs. For these reasons we chose the tradeoffs described before.

L2 cache recovery procedure

1. L2 cache crashes, loses all data.
2. L2 cache recovers.
3. No specific actions are put in place: L2 cache will simply re-populate itself by being involved in future *Operations*.