UNIVERSITÀ DI TRENTO

Distribute System 1 2022

Final Project

# MultiCache
*MULTI-LEVEL DISTRIBUTED CACHE*

Ashkan **Alami**

Shuxin **Zheng**

Academic Year 2021/2022

# Contents

**1 Structure of the project** **2**

**2 Main design choices** **2**

2.1 Cache Data Structure . . . 2

2.1.1 Common Cache Data Structure . . . 2

2.1.2 Specific L2 Cache Data Structure . . . 3

2.1.3 Specific L1 Cache Data Structure . . . 3

2.2 Operations Implementation . . . 3

2.2.1 READ . . . 3

2.2.2 WRITE . . . 3

2.2.3 CRITICAL READ . . . 4

2.2.4 CRITICAL WRITE . . . 4

2.3 Workflow . . . 4

2.4 Crash and Recover . . . 5

2.4.1 Crash . . . 5

2.4.2 Recovery . . . 5

**3 Discussion of the implementation** **5**

3.1 Eventual consistency . . . 5

3.2 Consistency Possibility . . . 5

3.3 Liveness . . . 5

# 1 Structure of the project

Our project is designed to have a very simple structure:

One **Message** class containing all type of message types e.g. WRITE, READ, CREAD etc

Four **Actors** classes which extend the *AbstractActor* class: *Client*, *L2C*(L2 cache), *L1C*(L1 cache), and *Database*

One **task generator** called *God* here in our project, and

One **Controller** called *App* here in our project The main(entrance) class used to compile the project is *App.java* in the directory *src/main/java/playground* as written in the file *build.gradle*. All the components work like following:

**App:** here Actors such as Clients, L2 caches, L1 caches, Database, and God are created.

**God:** Creates the tasks, and randomly sends them to the clients. It is also responsible for sending the crash messages, and making the nodes crash and recover. In the end, it asks all nodes to print their log.

**Client:** It gets the tasks (messages) from God. It sends them to L2 cache and waits to get the answer for that message and after that, it sends the next message.

**L2C:** It is responsible for communication between the Client and the L1 cache. It usually passes the message from the Client to the L1 cache or gets an answer from L1 caches end to the Client.

**L1C:** It is responsible for communication between the L2 cache and the Database. It usually passes the message from the L2 cache to the Database or gets an answer from the Database end to the L2 cache.

**Database:** It gets the message from L1 and does the operation, and sends back the message with its result back to the L1 cache.

All the code can be found in the github: `https://github.com/lannnnn/DS-project`.

# 2 Main design choices

## 2.1 Cache Data Structure

The data structure for the caches of L1 and L2 have lots of common parts but there still be some differences. We put here both the structure definition with detailed annotations. We start with the common part and then the specific part for L1 and L2 (start with L2 since our project is developed from bottom to top).

### 2.1.1 Common Cache Data Structure

```java
private final int id;             // permanent id for visit
// current parent, database of L1, L1 or DB for L2
private ActorRef parent;
private Boolean Send;             // state: whether is processing message
private HashMap<String, String> Ldata; // cache data table
private List<Message> continer; // message queue
private int waitingTime;          // define timeout time
private int deletingTime;         // time for refresh cache
private boolean cw_waiting;       // critical write timeout
private Boolean crash;            // whether the cache is crashed
private Object lastMessage;       // remember the last message
private String MyLog;
private Random rnd = new Random();
private Cancellable timer;
private String ignoreKey;         // blacklist for critical write
```

### 2.1.2 Specific L2 Cache Data Structure

```java
public class L2C extends AbstractActor {
    private ActorRef L1;              // original assigned L1
    private ActorRef databaseRef;     // used when L1 crashed
    private Cancellable CWTimer;      // clear the blacklist cw data
    private int lastMassegeId;        // used to clear the sent state
    private int AbortwaitingTime;     // timeout for wait the cw data
}
```

### 2.1.3 Specific L1 Cache Data Structure

```java
public class L1C extends AbstractActor{
    private List<ActorRef> L2s;       // child nodes
    // child node to keep trying contact with
    private List<ActorRef> childrenDontKnowImBack;
}
```

## 2.2 Operations Implementation

We will here discuss the detailed implementation of the four main operations. The details will be listed in four separated layers: start from the client, the L2 cache, then the L1 cache, and finally the database. Nodes have a *Send* state, when they send a message forward to the Database, they change it to the *True*. as long as this variable is true, any received messages that they have to send to the database direction will be put and kept in the container, until they get the answer, after that, they change *Send* state, to *False* and send the oldest message in their container.
Nodes also have a timeout message which they use it as a timer. In Clients and L2 caches, they send this message to themselves after sending a message, to detect if a crash happened or not. if before the timeout message, they received the answer to the sent message, they cancel this timeout message. In Database and L1 caches this timeout is used for critical writing and recovery after a crash.

### 2.2.1 READ

**Client** sends a read message (*Message.READ*) to a randomly picked L2 cache. If it receives the result from L2 it cancels the timeout message and sends the next message from its container. in case of timeout, the client resends the message to another randomly picked L2 cache (not contain the current one) and repeats all the procedures for sending a message.
**L2 cache** on receiving a read message from the client, the cache first checks if the key exists on its table and is not on the critical write blacklist. If its exits and it is not on the blacklist, return the value and send it to the Client. If not, the cache sends the message to its parent. when the cache gets the answer, cancels the timer and sends the answer to the client, and checks and sends the oldest message in its container to its parent. In the case of timeout, the parent is changed to Database.
**L1 cache** does the same thing as L2. Since the database does not crash, doesn't have a timeout for checking the database's crash.
**Database** read the data from its dataset and gives it back to the cache that sent the message.

### 2.2.2 WRITE

**Client** do almost as same as READ operation, but the message sent is in class *Message.WRITE*.
**L2 cache** on receiving a write message from the client, the cache forwards it to its parent and set a timer. if the write message is received from its parent, if the message key exists in its local dataset, the cache updates the local dataset, and if this message is the message that was sent to the parent and the cache is waiting for the answer, the cache cancels the timer and sends it back to the client. In the case of the timeout, the parent is changed to the Database.
**L1 cache** do the same thing as L2. Upon receiving the result from the Database, update the data iff it contains the key and propagate the data to all the known children (L2 cache).
**Database** update the data and propagate the data to all the known children (L1 cache).

### 2.2.3 CRITICAL READ

Actors do almost the same with read operation, but the message sent is in the class *Message.CREAD* and the forward is mandatory - must read from the database no matter whether the cache contains the key or not.

### 2.2.4 CRITICAL WRITE

we can divide critical write into two operations, one is sending and receiving the message, which is as same as the read operations, each cache passes the message from the client to the database, and the message's answer to the client. another operation is the database checking process to confirm the critical write or abort it. since the first operation is similar to the previous ones we explain the second one.

**Database** after receiving the critical write, the database checks if all the L1c are alive and not crashed, sends the CW_check message to them, and waits for all of them to respond, if all of them answer, the Database sends the second message WriteCW message to all the L1 caches, write the new value, and send the confirmation of critical write message back to the L1 that sent the critical write message. if one of the L1s crashed and doesn't respond to the first message, the database, sends the abort message to all L1s and the abortion of the critical write message to the L1 that sent it.

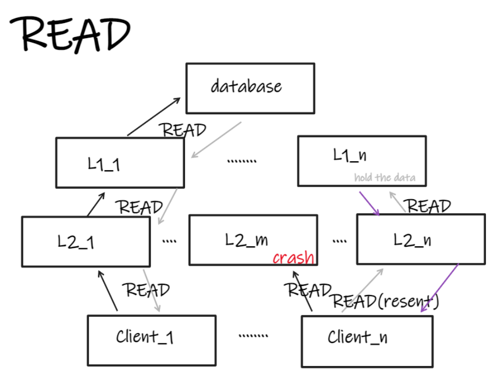
**L1 cache** when it receives CW_check, it put the key value to the blacklist and sends the CW_check to all its children and also to the database. on the second message (WriteCW) updates the local dataset if the value is available in its local dataset and sends the message to its children. in case it receives the abort, removes the value from the blacklist, and sends the abort to its children.

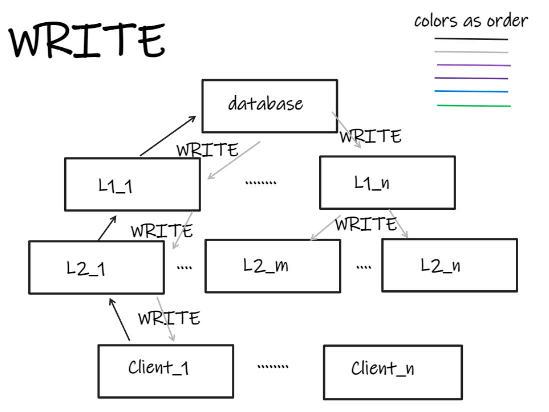
**L2 cache** when it gets the CW_check message, the key value is put to blacklist, and a timer sets. on the WriteCW message, the cache changes its local value if the value is available in its local dataset. if the cache receives the abort message, it cancels the timer and removes the key from the blacklist. in the case of timeout, the cache deletes the value from its local dataset.

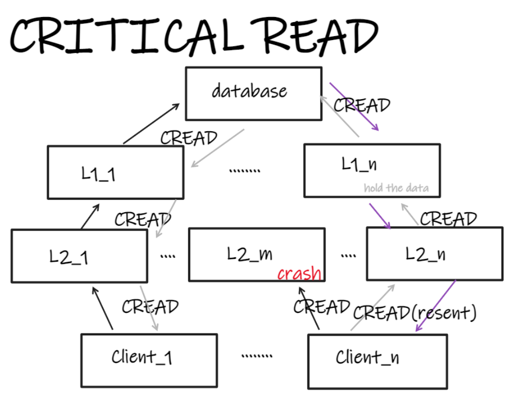
**Client** does almost the same as the previous operations, but the message sent is in class *Message.CWRITE.*

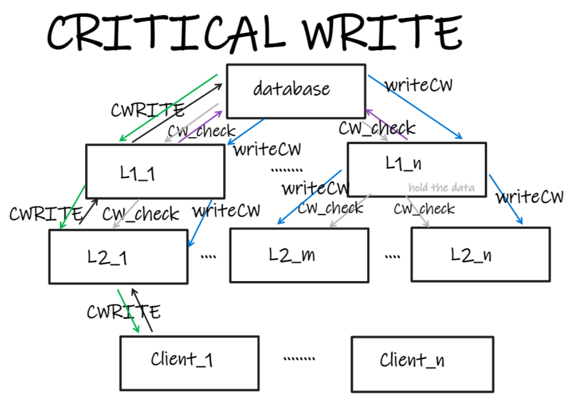
## 2.3 Workflow

Here we put some special cases for illustrating. We can not contain all the cases due to space limitations.









## 2.4 Crash and Recover

### 2.4.1 Crash

The crash is simulated by sending a CRASH message to the target L1/L2 cache. when a node receives the CRASH message, the cache process the crash process if is not currently crashed. During the crash process, all the data in the data table, the message queue, and the blacklist for critical write is totally lost, together with all the state values. However, the system topology knowledge(typically children/parent list) is preserved.

### 2.4.2 Recovery

We are here to discuss the behavior among recovery separately for the L1 and L2 cache. We start with L2 as we always do.
**L2** cache will start with setting the parent as the origin L1 no matter who is the last one when the crash happened (this avoids L2 being lost in connecting to the Database when L1 recovers during the L2 crash). And then it will handle the requests just as normal.
**L1** cache will tell all the L2 children it has come back so the L2 caches which are still alive will then change the parent back. a message will be sent to all the known children and when they get it they acknowledge it. A list of none acknowledged L2 children is kept so periodic notification is valid to mask the performance fault. In this design, L2 caches will not be got lost in the Database when the assigned L1 is alive.

# 3 Discussion of the implementation

## 3.1 Eventual consistency

Eventual consistency guarantees that if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

A tricky and useful part in our design to guarantee this feature is that we make every cache periodically delete the oldest cached data in its data table, which means if a client queries for a data, it will finally get the latest updated one even without the critical read operation. The write and critical write operations are guaranteed to modify the data in the database, and propagation is made to notify the caches that contain the corresponding key. while the critical one will make sure all the caches clear the old value. So by doing this, the data accesses will eventually get the latest modified one - eventual consistency.

## 3.2 Consistency Possibility

We are here to discuss the achievable consistency models from weak to strong.

**weak consistence** can be achieved by using the critical write as the sync operation in the weak consistency model.

**FIFO, casual and sequential consistency** can be achieved by replacing all the read operations with the critical read, or replacing all the write operations with the critical write. In the first case, since all the write operations will access the database, once the write is done, all clients can get the latest data since we have assumed that the channel is reliable and FIFO. For the second, all the cache data will be updated in the same order again since we assumed FIFO reliable channel. But the critical read will be less 'expensive' than critical write since non block required, and also the probability to abort is lower. But the second one may work better than replacing the read operations when the number of read operations is much larger than the number of write operations and vise versa.

## 3.3 Liveness

Our multi-level distributed cache system enjoys excellent liveness. It can even resume from fully crashing e.g. all the caches are not valid. When an L2 cache crashed, the client will keep trying another till the requirement is satisfied. A crashed L1 will not affect the read/write/critical read operations since the L2 simply asks the database directly, the only bad thing, in this case, is that critical write will never succeed. During recovery, L2 will notify the L1 parent, but if the parent does not survive, it turns to the database, this mechanism avoids the 'lost L2' - or else the L2 may never come back to L1 if L1 recovers during L2 crashed. And a recovered L1 will also send a notification to tell the children to turn back to it, in this case, there will not be more and more L2 'get lost' e.g direct access to the database. All the operations will eventually be fed, and the system has a strong ability to survive even catastrophic crashes.