Course Project: Multi-level Distributed Cache

Distributed Systems 1, 2021 – 2022

In this project, you are requested to implement a protocol for a multi-level distributed cache. The goal of a distributed cache is to prevent congestion at the main database, as caches store highly-requested items and handle most client requests independently. The system comprises multiple clients that read and write items stored in a database. They cannot access the database directly, but only through a network of cache hosts, which are arranged on two levels and connected in a tree topology. The system is optimized for Read operations, which are typically served immediately by the cache layer close to the clients. Instead, Write operations always involve the main database. The protocol should provide eventual consistency, and clients that interact with the same cache are guaranteed not to see old values after their write operations have been confirmed. There are also two "critical" variants for the basic operations, Critread and Critwrite, that provide stronger guarantees.

1 General description

The project should be implemented in Akka with *clients*, *caches* and one *database*, all being Akka actors. For simplicity, we assume that the data held by database and caches consists of integer values. Each of these values is attached to a globally-unique key, and together they constitute a database item.

We assume that the nodes of the system do not change over the execution of the protocol: when the program starts, a configurable number of clients and caches are created. The initial tree structure of caches is also predefined. Some caches operate at the first level, with a direct connection to the database. We refer to those as L1 caches, borrowing from CPU architecture terminology. L1 caches are the parents of L2 caches in the tree. The latter interact with clients, receiving their requests and responding with the results.

Client requests. A client contacts a given L2 cache with a request (read request: Read/Critread, or write request: Write/Critwrite). All request types refer to a specific item, identified by a key. Write requests also include the new value for the item. The client waits for a response from the same cache it sent the request to. In the case of a read, the response is the value of the requested item. For the write, it is a confirmation that the write operation was successful. Requests may fail if the item is not accessible at the time of the operation, e.g., due to a cache crash or other issues in the system. The cache then replies with an appropriate message to the client.

Basic operations.

- Read. When an L2 cache receives a Read, it responds immediately with the requested value if it is found in its memory. Otherwise, it will contact the parent L1 cache. The L1 cache may respond with the value, or contact the main database (typically referred to as read-through mode). Responses follows the path of the request backwards, until the client is reached. On the way back, caches save the item for future requests. Client timeouts should take into account the time for the request to reach the database.
- Write. The request is forwarded to the database, that applies the write and sends the notification of the update to all its L1 caches. In turn, all L1 caches propagate it to their connected L2 caches. In this way, the update is potentially applied at all caches, which is necessary for eventual consistency. Note that only those caches that were already storing the written item will update their local values.

Crash and recovery. Caches may fail at key points of the algorithm. The system should implement a simple crash detection algorithm based on timeouts, as seen in the labs. A client detecting a crashed L2 cache will select another L2 cache and redirect its requests. A L2 cache that detects its L1 parent has crashed will select the main database as its parent. When they crash, caches lose all stored items (we assume they were stored in volatile memory). However, they retain knowledge about the system, including the ActorRef of their neighbors in the tree and the database actor. Caches recover after some configurable time and resume operations.

Critical operations. Some client request may be more important than others, and require stronger guarantees.

- Critread. Fetches the current value stored in the database, but contrary to a Read, the request is forwarded 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, it propagates the update as for WRITE.

2 Implementation-related assumptions and requirements

- We assume links are FIFO and reliable.
- To emulate network propagation delays, you are requested to insert small random intervals in message transmissions.
- Ensure actor encapsulation. Avoid sharing objects unless they are immutable.
- A cache has unlimited memory for storing items.
- To emulate crashes, a cache should be able to enter the "crashed mode" in which it ignores all incoming messages (except the message for recovery) and stops sending anything.
- Upon crashing, all items are removed from the cache. However, the crashed node may retain some knowledge about the system, as detailed in the crash description. In your implementation, the retained information should be simply stored in local variables of the actor, which, unlike items, are not cleared upon crashing.
- During the evaluation it should be easy, with a simple instrumentation of the code, to emulate a crash at key points of the protocol, e.g., upon the reception of an update message within a WRITE operation.
- Clients do not crash. The actor representing the main database does not crash.
- Before sending a new request, a client waits for its previous one to finish, either when a response is received, or due to a timeout.
- Your implementation must include a mechanism to assess whether the state of the system provides eventual consistency after a number of operations have taken place.

3 Grading

You are responsible to show that your project works. The project will be evaluated for its technical content (algorithm correctness) and your ability to discuss implementation choices. *Do not* spend time implementing features other than the ones requested — focus on doing the core functionality, and doing it well.

A correct implementation of the whole requested functionality is worth 6 points. It is possible to submit programs implementing a subset of the requested features or systems requiring stronger assumptions. For example, you may assume no crashes. Or, you may not implement CRITWRITE. In these cases, lower marks will be awarded.

You are expected to implement this project with exactly one other student, however the marks will be individual, based on your understanding of the program and the concepts used.

4 Presenting the project

- You MUST contact through e-mail the instructor (gianpietro.picco@unitn.it) AND the teaching assistant (davide.vecchia@unitn.it), well in advance, i.e., at least a couple of weeks before the presentation.
- You can present the project at any time, also outside of exam sessions. In the latter case, the mark will be "frozen" until you can enroll in the next exam session.
- The code must be properly formatted otherwise it will not be accepted.
- Provide a short document (typically 3-4 pages) in English explaining the main architectural choices and discussing how your protocol satisfies the requirements of the project.
- Both the code and documentation must be submitted in electronic format via email at least one day before the meeting. The documentation must be a single self-contained pdf. All code must be sent in a single tarball consisting of a single folder (named after your surnames) containing all your source files. For instance, if your surnames are Rossi and Russo, put your source files and the documentation in a directory called `RossiRusso`, compress it with zip or tar ("`tar -czvf RossiRusso.tgz RossiRusso`") and submit the resulting archive.
- The project is demonstrated in front of the instructor and/or assistant.

Plagiarism is not tolerated. Do not hesitate to ask questions if you run into troubles with your implementation. We are here to help.