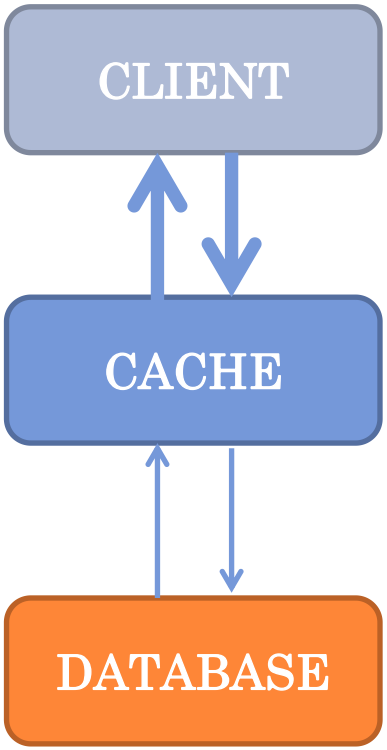
Distributed Systems 1 - Project 2022:
Multi-level Distributed Cache

Caching

- Caches store the most requested items in main memory to avoid expensive lookups at the main database

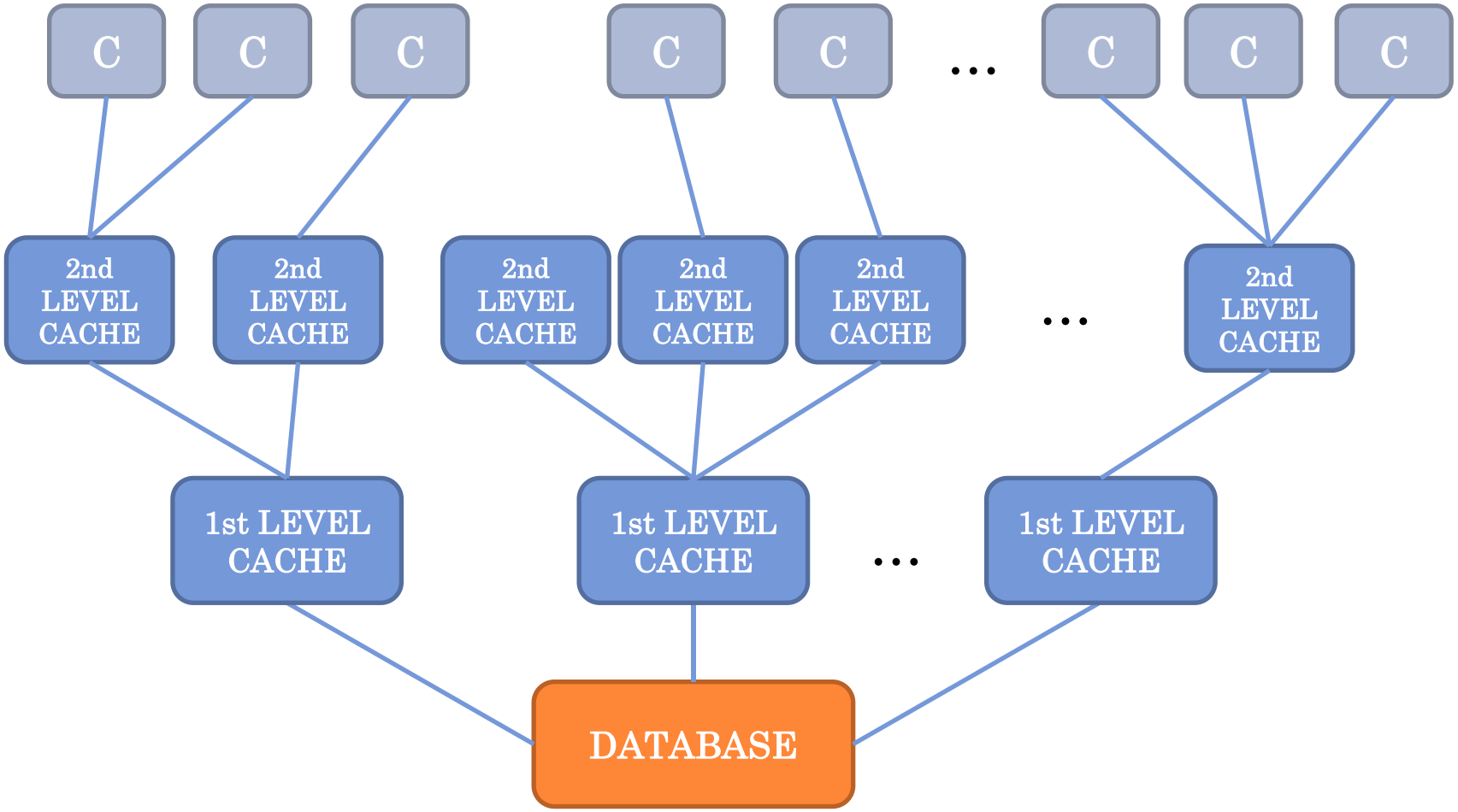


Advantages:

- No I/O bottlenecks
- Load balancing
- Extreme scalability
- Reduced latency if the cache is deployed near the user
- Protected access to the main database

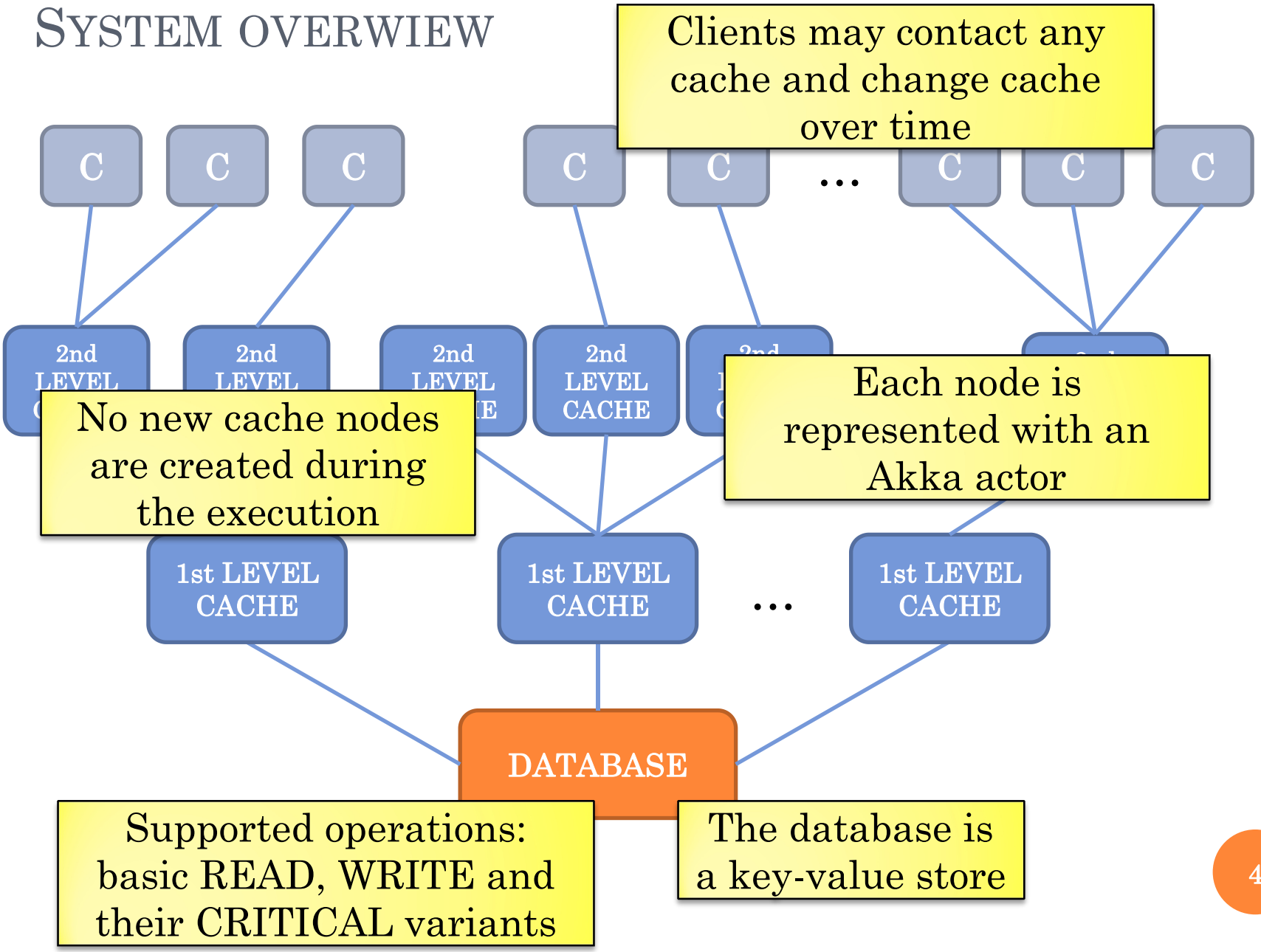
In this project, you will implement a multi-level distributed cache.

Multi-level distributed cache



- Multiple levels of caching (in a tree topology) reduce the number of messages at the database and in the cache layers

System overwiew



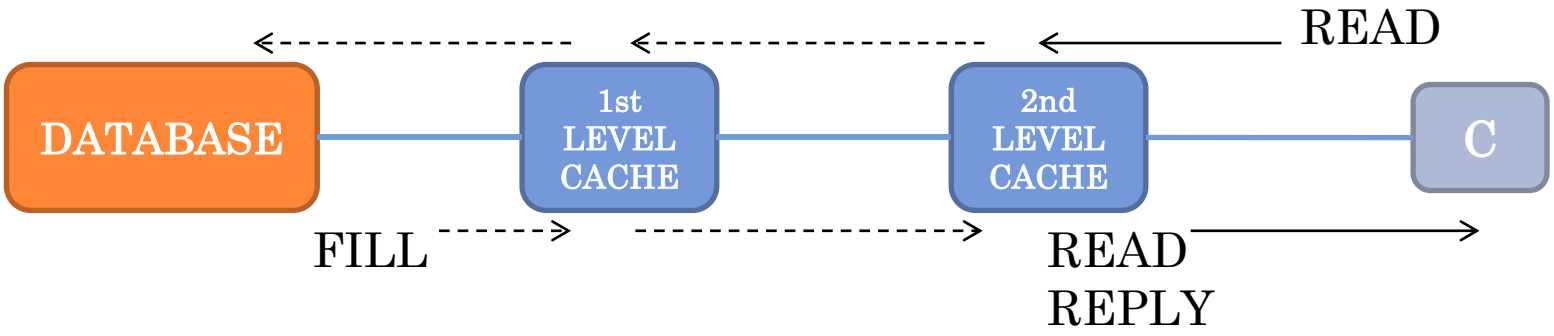
A SIMPLE KEY-VALUE STORE

- For simplicity, the database is simply a collection of key-value pairs. We call them *items*. Clients read and write the value identified by a unique key.

- When you create the database, it already holds the items. The set of keys is known in advance to the clients and read/write requests are only on those items.

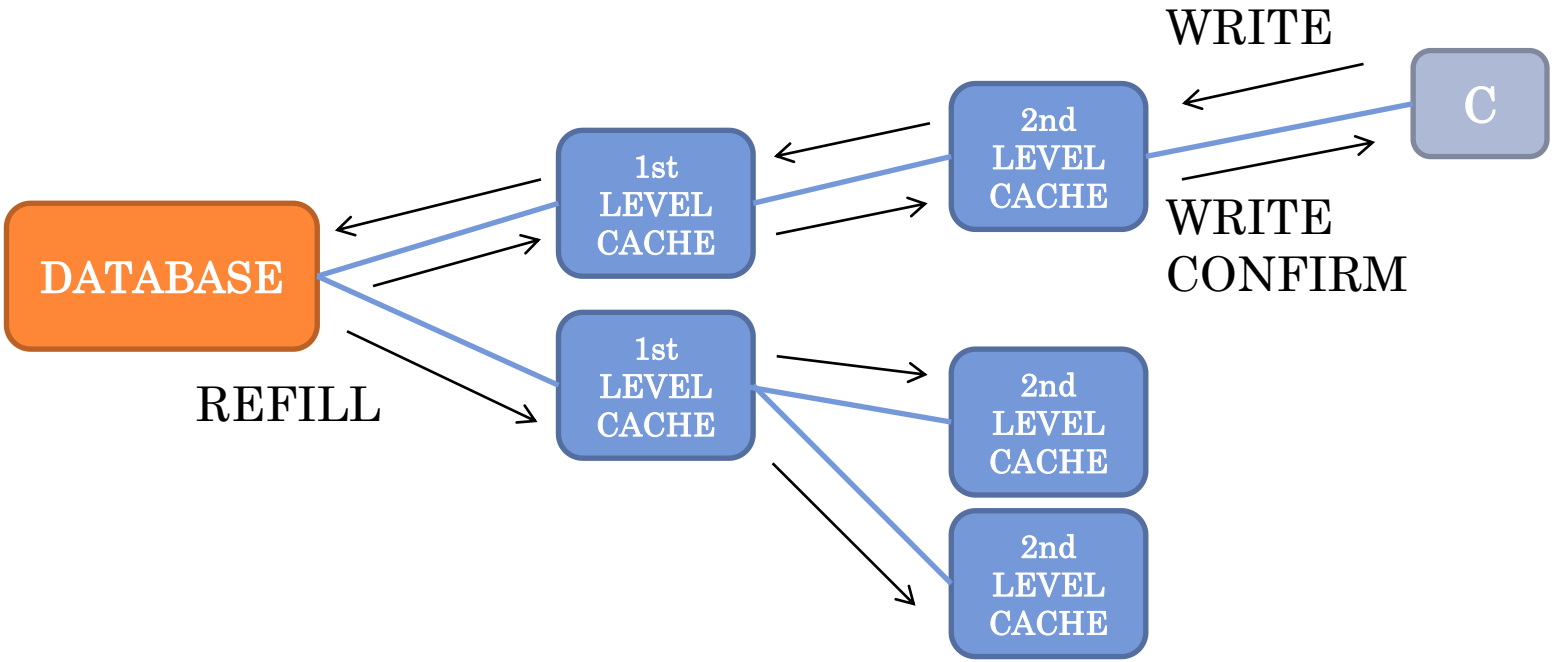
KEY	VALUE
0	7
1	129
...	...
9	16

CLIENT OPERATIONS: READ



- The READ request is sent by the client to the 2nd level cache instance (L2) it interacts with.
- If L2 already holds the item with the requested key, it immediately returns the value to the client.
- Otherwise, it contacts the parent cache, L1 (*read-through*).
- If L1 does not hold the key, it contacts the main database.
- On the way back (on the requesting path), fill the cache with the item.

CLIENT OPERATIONS: WRITE



- The WRITE request is *always* sent all the way to the main database (*write-through*).
- Then the database propagates the update along the tree to (re)fill **every cache** with the item.
- Only the cache nodes that were holding the item should add it in memory.
- L2 finally sends confirmation to the client. From that point, the client considers the write to be applied.

Consistency guarantees

- When interacting with the same cache, a client is guaranteed not to read a value older than a write confirmed to be applied. After its WRITE is applied, the READs of the client must observe the written value.
- The system should provide **eventual consistency**: updates eventually propagate to all replicas.
  If there are no more updates to an item, eventually all READs will return the last updated value.

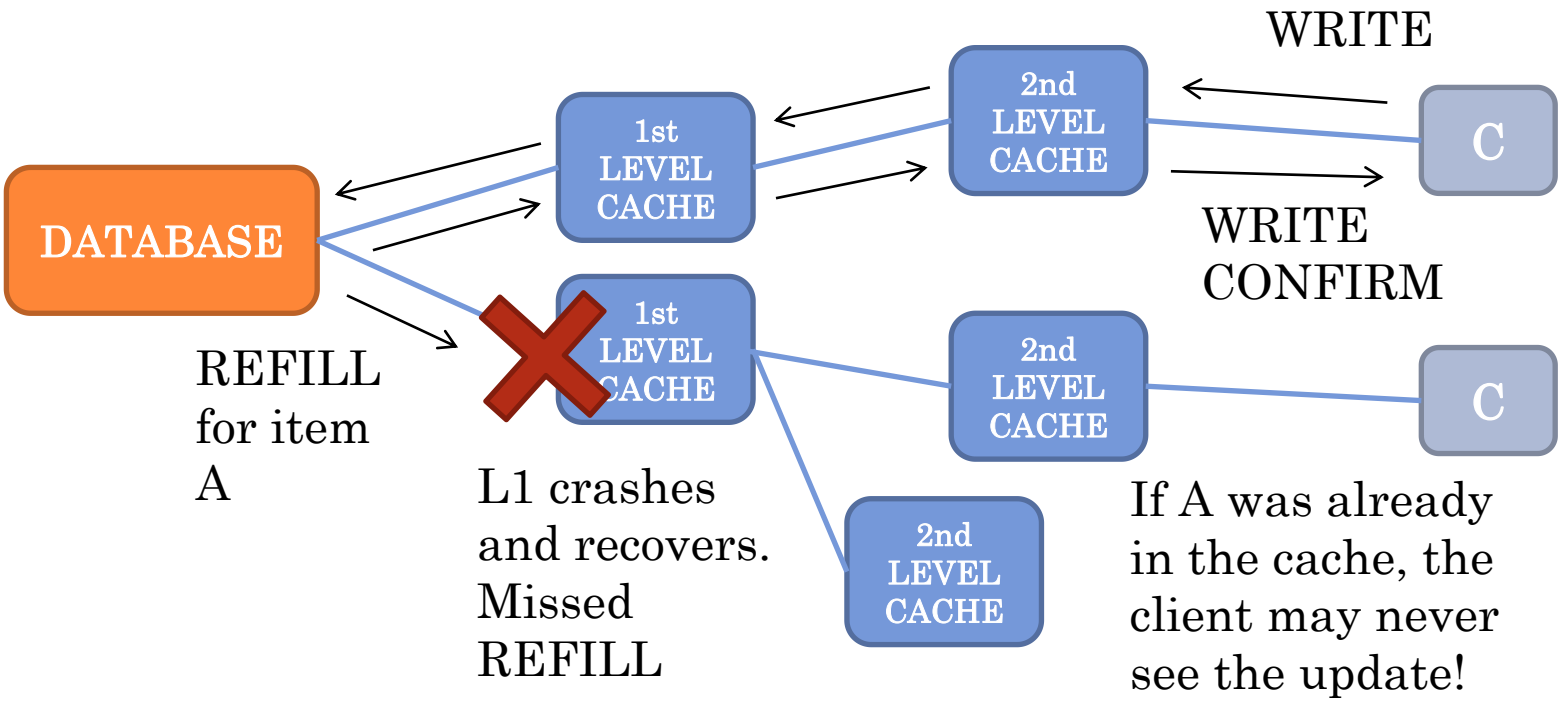
These consistency guarantees must hold even in the case a cache node crashes.

Crashes

- You will simulate the crash of **cache nodes**. Since they store items in main memory, they will lose them all upon crashing. They may still keep some information in persistent memory, e.g., the ActorRefs of their neighbors. In the implementation, simply use some variables of the actor, that you do not reset upon crashing, as if they were saved persistently.
- Clients can detect a crashed cache using timeouts. In that case, they will contact another L2 cache as their new parent.
- L2 cache uses timeouts as well for L1 cache. In case of a crash, they select the main database as their parent.
  - In a real system, another process would then "reconstruct" the tree; do *not* implement this in your project.
- When possible, ensure operations continue even in the presence of a crash.

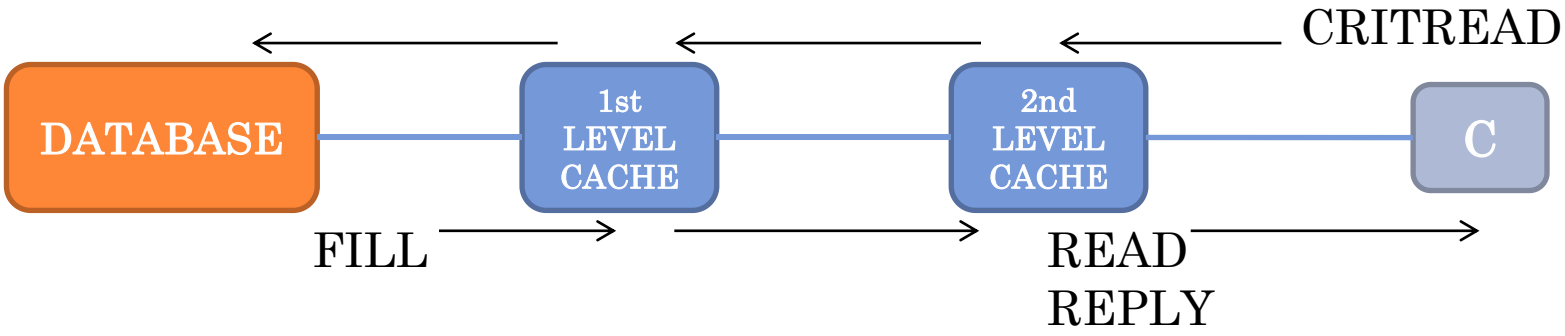
Eventual consistency with crashes

- What if a cache crashes before receiving the REFILL and misses it?



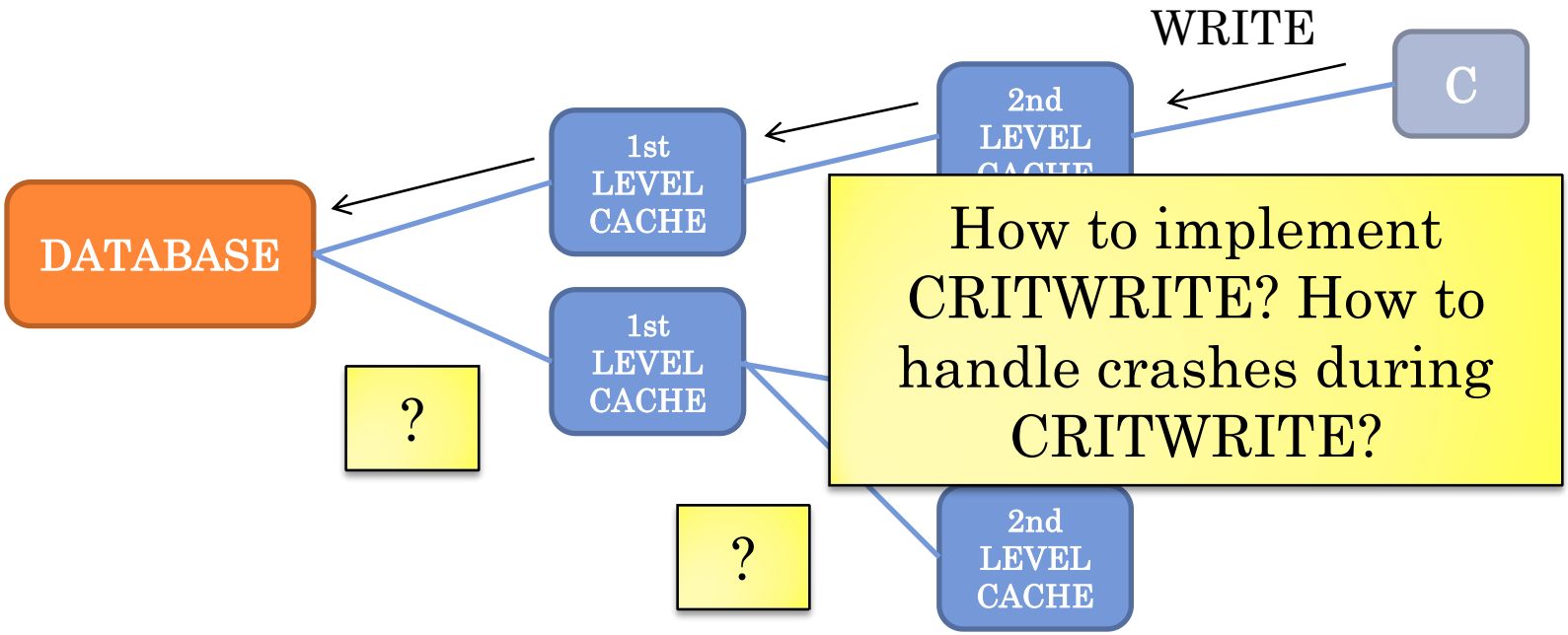
*Hint: remember the system provides **eventual consistency**. Make sure updates are eventually propagated, or that stale data is eventually evicted from the cache.*

CLIENT OPERATIONS: CRITICAL READ



- The system supports additional operations with stronger guarantees.
- CRITREAD ensures that the returned value is not stale, as the request is sent to the main database regardless of cached items.

CLIENT OPERATIONS: CRITICAL WRITE



- Before it confirms the write was applied, CRITWRITE ensures that no old item remain in the whole cache.
- This means that **no client** can read the new value and then the old value **at any cache**.
- Therefore, when using CRITWRITE, the client is guaranteed not to see old values even if it interacts with **different caches**.

Assumptions

- Reliable FIFO channels.
- In a real system, the cache would have limited memory, and some (less used) items would need to be removed. In this project, you can assume that the cache has unlimited memory for storing items.
- Cache nodes may crash. Those that crash recover after a specified delay. Assume only **one crash** at a time.
- The Akka actor representing the main database never crashes.
- The clients do not crash.
- A client waits for the response (or the timeout) before sending the next request.

Suggestions

- First, create a fixed multi-level structure for caches. Use configuration parameters for the number of L1 caches and the number of L2 for each L1. At startup, also create a (configurable) number of clients.
- Then, implement the system with no crashes, and only with READ and WRITE operations.
- Add crashes and check that all updates are eventually applied.
- You can test this be checking that any write operation that was applied at the main database is also applied to all caches that hold the item. The order of operations is given by the processing order at the main database.
- Finally, include the CRITICAL variants.

Project report

- Structure of the project
- Main design choices
  - How did you implement the data structure for the cache?
  - How did you implement READ, WRITE, CRITREAD and CRITWRITE?
  - You may include pictures with the workflow and the various messages used in your implementation.
  - How are crashes simulated and what do nodes do upon recovery?
  - …
- Discussion of the implementation
  - Does it achieve eventual consistency? How?
  - What consistency model can be provided using CRITREAD and CRITWRITE? What are the trade-offs?
  - Comments on liveness; availability upon cache failure.
  - Are there any additional assumption about the system that you need to make?
  - …