# DADTKV

Design and Implementation of Distributed Applications 2023-2024
IST (MEIC-A / MEIC-T / METI)
Project Description

Version 1.0

## 1 Introduction

The goal of this project is to design and implement **DADTKV**, a distributed transactional key-value store to manage data objects (each data object being a <key, value> pair) that reside in server memory and can be accessed concurrently by transactional programs that execute in different machines.

## 2 Architecture

**DADTKV** has a first tier composed of client applications that use the **DADTKV** system and execute transactions on data stored in it. The operations are submitted to a set of transaction managers, the second tier, that provide transactional access to the stored data and replicate the transactions' updates to ensure the durability of the information. Leases are used to regulate concurrent access to shared data by different transaction managers. These leases are acquired by the transaction managers from the lease manager servers, the third tier of the **DADTKV** system. The group of lease manager servers determine which transaction manager a lease is assigned to by executing a consensus algorithm among themselves.

### 2.1 Data

To simplify the work, and make the project doable in the timeframe available for the project, the applications will only share a particular type of object, of type *DadInt*. Each *DadInt* object is a key-value pair, where the key is encoded via a string, and the value is an integer. For example a *DadInt* could be a pair composed of the key "account-43-balance" and the value -17. Each shared object can only be accessed in the context of a valid transaction. For simplicity, transactions cannot execute arbitrary logic but consist solely in the execution of a set of read and a set of write operations. So, in order to submit a transaction, clients need simply to send: i) the set of *DadInts* to be read and ii) the set of *DadInts* to be written to the transaction managers. To process a transaction, a transaction manager execute first the set of read operations, then the set of write operations; finally, it returns to the client the set of values it read once the written data is durably stored in the system. Overall, the execution of transactions must ensure strict serializability.

### 2.2 Clients

**DADTKV** clients are individual processes running transactions in a loop to manipulate data stored at the **DADTKV** servers. The sequence of operations executed by a client in each iteration of the loop is specified in a script that is executed sequentially. The script may also include a "wait" instruction (for a number of milliseconds) in order to insert a waiting time between other commands. Clients should print out in their consoles the result of each transaction that is concluded.

## 3 The DADTKV client library

Clients applications use the **DADTKV** library to interact with the transaction managers, that maintain a replica of the key-value store. The library exports a set of methods that provide access to **DADTKV**:

- $List\langle DadInt\rangle\ TxSubmit(string, List\langle string\rangle, List\langle DadInt\rangle)$: this method submits a transaction for execution at a transaction manager. The parameters are a string identifying the client executing the transaction, a list of strings indicating all of the *DadInt* to be read by this transaction and a list of *DadInt* objects to be written by this transaction. The returned list of *DadInt* is the result of the transaction's read operations. If a transaction is aborted the resulting return value will be a list containing a single *DadInt* with the key `abort`. `abort` is a restricted key name that may only be used for this purpose.
- *bool Status()*: this method makes all nodes in the system output their current state.

### 3.1 Leases

Leases are used by transaction managers to obtain a permission to access exclusively a set of *DadInts*. A lease is defined by a string identifying a transaction manager, e.g. TM1, and by a list of exclusive permissions for a set of *DadInts*, e.g. $"objA", "objB"$. Once obtained, a lease is held indefinitely by a transaction manager that has requested it until it detects that a conflicting lease has been attributed (by the lease managers) to a different transaction manager. Two leases conflict if they both include permission requests for the same *DadInt*.

## 3.2 Transaction Managers

The transaction managers are a set of nodes that execute transactions on behalf of client applications. They store, in memory, a set of *DadInt*s. All transaction servers store a full copy of the set of *DadInt* in the system.

When a client submits a *TxSubmit* request, the transaction manager will check if it holds the appropriate leases for the *DadInt*s included in the *TxSubmit* request it received. If it does, transactions can be executed immediately. If not, it sends a request to all lease managers to obtain a lease on all the to all *DadInt*s accessed by that transaction. By acquiring leases prior to executing the transaction, transaction managers ensure that they only execute concurrently non-conflicting transactions, while serializing conflicting ones. Leases allow a transaction manager to execute more than one transaction that requires the same lease (the same permissions) without having to use consensus (as in the classic state machine replication approach). Clearly, this is only advantageous if the transactions executing at different managers are likely to access different sets of data items, so that leases do not need to be frequently acquired.

Note that, in order to ensure *liveness*, if a transaction manager crashes while holding a lease on some data item, the other transaction managers need to eventually forcibly free that lease. Additionally, they need to determine which transactions TM had executed and reestablish a consistent state in the system prior to start executing new transactions (that may access data items associated with a forcibly freed lease).

The result of the lease request will arrive asynchronously from the lease manager servers (see Paxos algorithm below) and will include the order number of the consensus instance (or epoch) to which it applies and a list of leases that have been assigned for the upcoming epoch. A transaction manager may receive multiple (identical) messages with the leases for an epoch. The first message it receives marks the beginning of that epoch for that transaction manager.

Note that if a TM is holding a lease that conflicts with another assigned lease, it should release the lease after executing a single transaction. Otherwise it may continue to hold the lease indefinitely and execute multiple transactions without ever releasing its current lease.

For simplicity, it can be assumed that, although concurrent client requests may be received and stored by each transaction managers, transactions will be executed sequentially by a single thread at each of the transaction managers.

## 3.3 Lease Managers

The role of lease managers is to order lease requests and assign leases to transaction managers.

Lease assignment is implemented using the Paxos algorithm. Lease managers periodically execute a new instance of the Paxos algorithm to establish the order with which the requested leases are attributed to the transaction managers. We call each instance of Paxos algorithm an epoch. When a new epoch $i$ begins, a lease manager orders deterministically any new lease request received since the start of epoch $i-1$ and proposes this list to a new instance of the Paxos algorithm.

### 3.3.1 Fault Tolerance

**DADTKV** server processes (transaction managers and lease managers) may fail by crashing but, for simplicity, are assumed to never recover after that. Only a minority of the servers (processes) in each of the transaction manager and lease manager tiers may fail if the system is to continue making progress. The transition from "normal" to "crashed" is triggered via a control interface.

Each server keeps a"*not-suspected*" or"*suspected*" flag, that captures its own guess about the state of other processes of the its type (transaction or lease managers), i.e., if it believes other processes are "correct" or"crashed". The "suspected"/"not-suspected" may be inaccurate (for instance, a process may "suspect" that another process is "crashed" when it is not). How processes "guess" the state of other processes is described later in the text.

Both the transaction manager group and the lease manager group require a majority of active processes to make progress. Finally, for simplification, students do not need to ensure that the client can retrieve the results of its transactions (i.e.., the set of DadInts read by a transaction it submitted) if the contacted transactional manager crashes. However, as already mentioned, transactional managers must ensure that if a transaction commits (independently of whether its result message is delivered to the client), its updates are propagated to every correct server.

## 3.4 Time Slots

For testing purposes, it is assumed that the state of the processes in the system is updated in a sequence of time slots. For each time slot, there will be different state of the processes in terms of fault tolerance. "Slots" are used by the "control" plane of the project, to change the state of processes (from normal to crashed or between suspected and not-suspected).

The system assumes that all processes have access to loosely synchronised clocks. Slots are attributed to (configurable) $\delta$ second intervals, starting at a given (configurable) instant, that is known by all participants at bootstrap. For simplicity, the period $\delta$ defining the duration of time slots should coincide with the period after which a new epoch/Paxos instance is started by the lease manages.

## 3.5 Deployment

An example configuration of the system could be a set of 12 processes:

- 3 lease manager servers
- 3 transaction servers
- 6 clients

These processes should be started by a management console that reads the systems configuration file and launches all required processes. For debugging ease it should also be possible to shutdown the whole system from the management console.

## 3.6 System configuration file

There is a shared configuration file that indicates:

- The identifiers and roles of each process (command P). You can assume that server processes are listed before clients. For the case of server processes, this command includes a URL where they can be contacted. For the case of client processes, this includes the path to client script.
- For how many slots the system is supposed to operate (command S).
- The value of $\delta$ that specifies the duration of a slot in milliseconds (command D).
- The global "wall" time of the beginning of the first slot (command T)
- Furthermore, for each slot, the system specifies if each node (in the **DADTKV** server groups) is "normal" or "failed" and a list of which nodes are suspected by other processes during that time slot (command F). A suspected node is represented by a pair containing the id of the suspecting process and the id of the suspected node.

Sample configuration files will be provided in the course website.

## 4 Implementation

The project should be programmed using C# and using gRPC for remote communication. For simplicity, it can be assumed that the configuration files and scripts are available on all nodes. It is also important to note that, although the scripts syntax should not be altered, additional parameters can be sent in the communication between servers or in process startup.

## 5 Final Report

Students should prepare a final report describing the developed solution (max. 6 pages). In this report, students should follow the typical approach of a technical paper, first describing very briefly the problem they are going to solve, the proposed implementation solutions, and the relative advantages of each solution. Please avoid including in the report any information already mentioned in this project description. The report should include an explanation of the algorithms used and justifications for the design decisions. The final reports should be written using LaTeX. A template of the paper format will be provided to the students.

## 6 Checkpoint and Final Submission

The grading process includes an intermediate optional checkpoint and a mandatory final submission. For the checkpoint the students may submit a preliminary implementation of the project; if they do so, the checkpoint grade may improve their final grade. The goal of the checkpoint is to control the evolution of the implementation effort.

The checkpoint submission should include at least the full system architecture with clients submitting transactions and transaction managers submitting lease requests to the lease managers. Lease managers and Paxos should be fully implemented. It is acceptable for the checkpoint that transaction managers do not execute transactions yet and to not handle scenarios in which transaction managers crash while holding some leases.

The final submission should include the source code (in electronic format) and the associated report (max. 6 pages).

## 7 Relevant Dates

- October 6th - Electronic submission of the checkpoint code;
- October 9th to October 13st - Checkpoint evaluation during the lab sessions;
- October 27th - Electronic submission of the final code and report.

## 8 Grading

The project grading will depend on a discussion at the end of the semester where all members of the groups must be present and where individual grades will be determined. That grade will depend on, besides the quality of the project, the individual performance in the discussion and the lecturer's evaluation.

The project grade (45% of the course's grade) is the *best* of the following two:

- Final_Project_Grade
- 85% of the Final_Project_Grade + 15% of Checkpoint_Grade

## 9 Cooperation among Groups

Students must not, *in any case*, see the code of other groups or provide their code to other groups. If copies of code are detected, all groups involved will fail the course.

## 10 "Época especial"

Students being evaluated during the "Época especial" will be required to do a different project and an exam. The "Época especial" project will be announced on the first day of the "Época especial" period, must be delivered on the day before last of that period, and will be discussed on the last day.