# *Dependable Distributed Systems* (2019/2020)

**Final Project (delivery until 15/Jun/2020)**
**Final Workshop for presentation and discussion**

The final project will integrate, consolidate, optimize and extend the successive requirements (initially conducted as Work-Assignments in Labs) and will be conducted in two parts:

Part I: will integrate, consolidate and optimize the work assignments from WA#1 to WA#4, in a final improved solution for your final project.
Part II: will include challenges of selected options, using materials, guidelines and discussions conducted in lab classes.

After the conclusion of both parts, students must prepare and deliver a final project package and experimental evaluation for demonstration, discussion, validation and evaluation (submitted on date from a Git Repository)

## PART I Variants (according to the prior Work-Assignments) Byzantine fault-tolerant system for Management of Wallets Auction Manager System

### 1. Introduction

The goal of this part is to create a system that maintains information about wallets or auctions. This system could be used, for example, to maintain the information about some virtual currency and P2P transactions, to maintain the information about wallets in some application (e.g. game) or to be used as an Auction Management System. For the development of PART I please follow the successive challenges in Work-Assignments (WA#1 to WA#4) and related references.

### 2. Architecture and specification

### 2.1 Operations

The system will support client interactions to manage wallets and transactions or wallets of auctions.
For the case of auctions you must support the following operations for the clients using the system:

- Create (who, auction): ok ?
- Terminate (who, auction): ok ?
- CurrentOpenAuctions ( ): open-auctions ?
- ClosedOpenActions ( ): closed-auctions ?
- AuctionBids (Auction): bids ?
- ClientBids (who): bids ?
- CheckBidClosedAuction (Auction): bid

For the variant of the Wallet Management System you must have the following operations for clients:

- createMoney( id, amount)
  // Add *amount* to the wallet identified by *id*. You should define under which conditions this operation succeeds or fails.
- transfer (id_from, id_to, amount)
  // Transfer *amount* from the wallet identified by *from_id* to wallet identified by *to_id*. This operation should fail if balance of the *from_id* wallet is smaller than *amount*.
- get( id)
  // Return the amount of money associated to the wallet identified by *id* or 0 if that wallet does not exist.

**2.2 Architectural requirements**

The system should be implemented using a client/replicated server architecture and expose a REST API (or a SOAP API), protected by TLS. The system should be able to tolerate Byzantine faults in the servers, using the traditional assumption that for tolerating f faults, the number of replicas should be *3f+1*. For tolerating Byzantine faults, the server should implement Byzantine Fault- tolerant replication, using the BFT-SMaRt to achieve this goal. Your solution should guarantee that:

- The client is connecting to a server of the systems and avoid replaying (Base: WA#1)
    - Suggestion: this will be achieved by TLS connections between the client and the server (and you can use the TLS setting with server-authentication only). If you want you can provide extensions for TLS parameterizations (cryptosuites of TLS enabled version, bust this will be regarded as an extra-value evaluation);
- The client should verify that the replies from the server are correct (Base: WA#2, WA#3 and WA#4)
    - Suggestion: the server must send to the client the replies received by the BFT-smart client, so that the client can verify that the result was returned by *f+1* replicas, and then resisting to a misbehavior of the contacted server;
- The systems must guarantee that the client has permissions to execute operations, implementing the notion of smart-contracts (together or complemented by a Mandatory-Access Control and/or Discretionary Access Control Models) for permission/access control management (WA#4 context).
    - Suggestion: you must provide code for permission control as a "smart card" (code passed as argument in a specific operation to install such smart cards to be executed by the servers, or code for the validation and permission as extended arguments in your operations) – in this way all the servers can load dynamically these "smart contracts" (as serialized executable objects or executable scripts), that will be executed consistently in all the replicas. The authorization to install or to send smart smart-contracts can be optionally based on initial MAC and/or DAC mechanisms.

Refined features that can be considered (in sequence if the prior developments from work-assignments) for the final project can include:

- For the client of your system, create a class that encapsulates the interaction with the servers, and maintains the client private information – e.g. if using Bitcoin/Blockchain approach for identities, with the private keys kept in the state of this class. For the purpose you can consider that public keys can be adopted now as client identifiers (or part of client identifiers).
- Furthermore, clients can generate and use different keypairs, meaning that during operation clients can exhibit different public keys (following the same idea as adopted for anonymous identifiers in permissionless blockchains). Only the smart cards will require identifiable and initially registered clients, to be allowed to submit/send smart contracts for access control on operations (or sent for the execution of submitted operations).
- For the final project, students can also improve dependability (reliability and security) guarantees, using the analysis from the prior Work-Assignment #3.

**3. Integration and Support for Experimental Evaluation for PART I**

The system must be evaluated experimentally using, preferably, a distributed setting with at least four server-replicas (tolerating one possible byzantine failure) in the replication cluster. You can chose to create dockers implementing server-replicas, you are free to use the programming support and runtime for the development and deployment of the server support (ex.,web-application and rest-services support technology). For the state-machine-replication and consistency you will use the BFT-Smart library (as suggested) to have support for ordered operations and byzantine-fault tolerance guarantees.

To finish and to evaluate experimentally your Part I implementation you must conduct:

- The integration of the functionality described above, with possible variants, optimizations or refinements you can propose for the final project implementation.
- You must have a client as a client-demonstrator for the provided operations and a client benchmark, with multiple threads, to test workloads on your system. The benchmark should start by creating a set of wallet with some initial money, followed by, in different threads, a sequence of transfer operations.
- Suggestion: create 100 * *num_threads* wallets; run threads by at least 3 minutes. It would be nive to be prepared to have observed results in a graphic with throughput vs. latency indications. See the BFT smart paper for inspiration for presentation of such results
- Another suggestion: can add an operation (rest) allowing the client to ask the server to perform a "reference

benchmark", using provided benchmarks in the BFT-Smart code (see the repository for sketched benchmarks for inspiration), to measure the specific impact due to the BFT-smart solution.

- Compare a configuration of the system with BFT-smart configuring and testing it to tolerate 0 crash or byzantine faults and to tolerate 1 crash fault. The report should present the throughput of the system and latency of the operations (or an hybrid throughput vs. latency graphic). Optional: compare with an additional configuration where there is a byzantine replica. In order to inject byzantine faults and to show that your system (overall) will be now byzantine fault tolerant. This can be implemented by having one replica that returns incorrect values randomly.
- It will be interesting to see the impact (on *thoughput vs latency*) observations, the impact of fail-stop and byzantine faults are injected
- Finally you can propose to enhance the PART I with any improvement you consider as valuable for the solution.

**PART II – System extensions**

The goal of this part is to improve the project with the revised/consolidated/optimized components, to create a dependable system with additional features. The following table suggests different options that students can consider, and at least one option must be considered as mandatory.

| | |
|---|---|
| A | Maintain additional state and be able to execute operations on the state (using homomorphic encryption facilities) and to provide support for operations performed on encrypted data in the storage layer. |
| B | Improve the system with Trusted Execution Environment guarantees for possible isolated execution in SGX, using SGX-enabled containers, and inclusion of a attestation |
| C | Support recovery facilities of the servers |
| D | Integration of the storage backend with a Backed-Blockchain Platform as a ledger solution registering all the operations in your system (as a complementary or as alternative solution for the storage layer initially provided in Part I) |
| E | Integration of the storage backend with a REDIS based cluster, providing improved resilience to the storage layer of your server replicas. |

Next we can find the initial indications for the above suggestions. You can (and you must) iterate or refine the requirements, discussing your ideas with the teacher.

**Initial Specification for A**

Besides maintaining information about a wallet, which can be maintained in clear text, the system should be able now to maintain additional state in encrypted mode. The system should support, at least, the following data types:

- HOMO_ADD: an integer, stored in cyphered mode, that supports an add operation with cyphered data (addition over encrypted integers):
- HOMO_OPE_INT: an integer, stored in encrypted mode, that supports the retrieval of database keys, whose values are between two given values.
- HOMO_MUL: an integer, stored in cyphered mode, that supports a multiplication operation with cyphered data (addition over encrypted integers):

For the support of smart contracts as previously implemented, the API of the system should support the following operations (for all the required data types)

- create (key, initial_value, type)
  // Add the (key, value) pair of type "type" to the database. The following types should be supported "HOMO_ADD", "HOMO_OPE_INT", "HOMO_MUL" and "WALLET".
- get (key)
  // Returns the value associated with key "key".
- get (key_prf, lower_value, higher_value)
  // Returns all keys with key prefix "key_prf" and values between "lower_value" and "higher_value" (inclusive).
  Simple alternative:
  Get (lower_value, higher_value)
  // Returns all key with values between "lower_value" and "higher_value" (inclusive).
- set (key, value)

// Set the value of "key" to "value".

- sum (key, value)
  // Sum "value" to the value of "key".
- sub (key, value)
  // Subtracts "value" from the value of "key"
- mul (key, value)
  // Multiply the value of "key" by the value "value" .
- conditional_upd (cond_key, cond_val, cond, list[(op,upd_key, upd_val)])
  // Executes the list of updates if the condition holds, where the condition is parameterized by "cond", as follows:
  - 0 -> db[cond_key] = cond_value
  - 1 -> db[cond_key] != cond_value
  - 2 -> db[cond_key] > cond_value
  - 3 -> db[cond_key] >= cond_value
  - 4 -> db[cond_key] < cond_value
  - 5 -> db[cond_key] <= cond_value

  // The list of updates is parameterized by "op", expressing:
  - 0: set value: db[upd_key] = upd_val
  - 1: add value: db[upd_key] = db[upd_key] + upd_val

  // Simple alternative. Implement the following operations:
  - conditional_set (cond_key, val, upd_key, val2)
    - IF db[cond_key] > val
      - db[upd_key]=val2
  - conditional_add (cond_key, val, upd_key, val2)
    - IF db[cond_key] > val
      - db[upd_key]=db[upd_key]+val2

NOTES:

- The solution to be developed may include additional parameters for any operation – e.g. to provide the data type of the keys. The system should be able to execute these operations on all data types, either cyphered or plain text. For a given operation the system should execute the operations resorting to the cryptographic algorithms used to support it.
- You can consider for the solution a key-value store technology (as a single REDIS instance, that can be used as an in-memory as well and in-disk solution) as a more easy approach for the persistency layer in your system.

Observation: For the case of Auction Manager System, you can support end-to-end encryption when supporting auction management operations. If you want to add the extension A in this case you must discuss the operations that you can provide to support encrypted data types and homomorphic operations, in order to give support with similar equivalent) operations as above.

**Initial Specification for B – Support for service supported in Isolated and Hardware-Backed TEE**

The system should be able to execute the operations in such a way that the server endpoints will be protected backed by an hardware-enabled trusted execution environment that can be leveraged by Intel SGX technology. The server will run in isolation. The suggestion is to run it within a SCONE container, that can bee tested to run in SGX simulated mode as well as supported on real SGX processing environment.
See (https://sconedocs.github.io/Java/) to execute the operation in a private context and https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov to know about SCONE.

NOTE: if you cannot run the Java service in SCONE in your machine because you dont have enabled SGX, try the SGX simulation mode or (as an alternative plan) just create the service and run it as a normal Java program. Later on we can check your solution on a real machine with enabled SGX.
It will be interesting for this option to obtain in the end a comparative experimental evaluation between the solution (in the end of PART I) and the solution using SCONE and a real environment using Intel SGX (to observe the differences).

**Initial Specification for C - Server recovery and authenticated boot**

The system should include a mechanism for executing server recovery, by stopping and restart a server when necessary. To this end, the system should include, in each machine, a service that can be accessed by the administrator (by using a specific client-side application) with the following operations:

- launch (url, hash, nonce, mainclass, parameters, smartcontract, signature)
  - URL: URL for obtaining a signed jar containing the software to run
  - Hash: hash of the jar to verify integrity // the jar must must be signed and the launch must verify the digital signature)
  - Smartcontract: will be the smartcontract for the server to execute the request under the access-control policy send by the administrator // Note that the request must be
  - Mainclass: Mainclass to run
  - Parameters: parameters to be used in the mainclass

The return with success of this operation must return a digital signature of the loaded class and an identifier of the execution

- attestation (nonce, smartcontract, signature)

  In any moment, any client (normal client or admin client) can ask for the attestation of the SW running in a server/replica. The result must be a digital signature of the digitally signed SW (image or jar corresponding to the launch operation together with the response to the sent nonce) that must be done according to the expressed smart contract sent and signed by the client. The idea is to offer a guarantee to clients that the SW stack launched and running in the server/replica is the one that the client expects.

- stop (id, smartcontract, signature)

  id: id of the software to stop, as returned by the launch. NOTE: for Smart-BFT to continue working, it is necessary that a quorum of replicas is running. The system should avoid stopping multiple replicas at the same time to avoid the unavailability of the system for normal clients using the system.

**Evaluation** The goal of the evaluation is to compare the performance of the different operations for the different data types. Thus, the benchmarks to run should, for each data type, start by creating (key,value) pair, followed by a sequence of equal operations.

**Initial Specification for D – Backend-Blockchain Integration for a decentralized ledger used by the system**

A suggestion here is to use Hyperledger Fabric and address a server node as a node that will use the HLF backend-blockchain as a ledger for the registration of all events (operations) as ledgered transactions. The integration architecture must be specified by the students and discussed with the teacher. The implementation will allow an experimental study on the dependability properties offered by the Hyperledger solution to enhance the project implementation and the overall dependability guarantees.

**Initial Specification for E – Use of a REDIS service for the resilience of the persistency layer of the system**

A suggestion here is to use REDIS in possible redundant configurations, for example primary-backup or replication cluster. The implementation will allow an experimental study on the dependability properties offered by REDIS to enhance the project implementation and the overall dependability guarantees.