CSD 2020/21 Report
BGWS System

Rodrigo Ribeiro

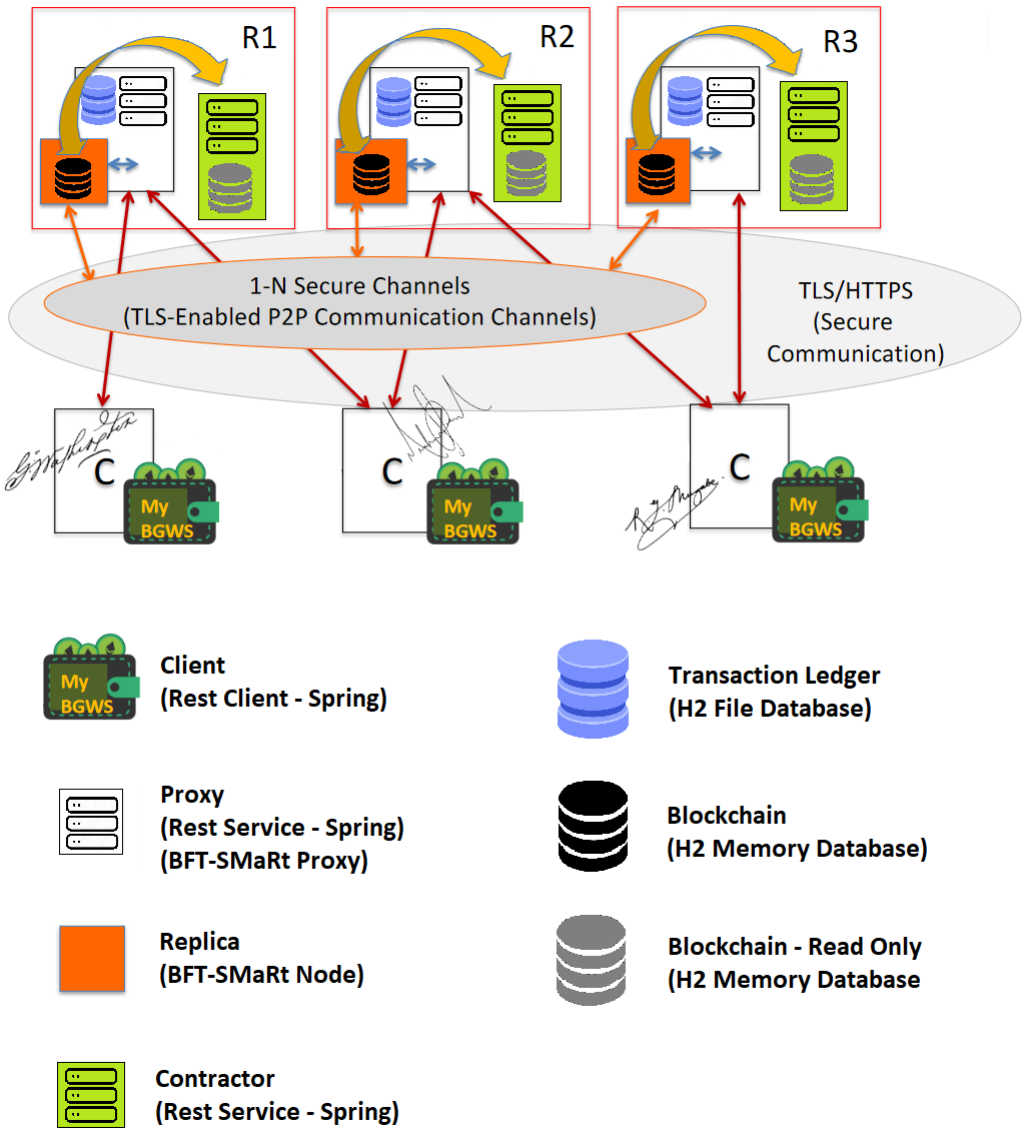
NOVA School of Science and Technology, Largo da Torre, 2825-149 Caparica, PT
https://www.fct.unl.pt/

Abstract. A Dependable Decentralized Ledger for a Blockchained Global Wallet Service (BGWS), supporting Distributed Client Wallets and P2P CryptCoin Transactions. Built on top of BFT-SMaRt, a Byzantine fault-tolerant state machine replication protocol.

1 Goals

1. A dependable fully decentralized service, no central authorities.
2. Replicated service (SMR model), w/ replication of consistent registry-logs of operations maintained by distributed multi-server instances, with total-order guarantees, global consistency and isolated persistence.
3. Byzantine fault-tolerant and resiliency guarantees for Intrusion Tolerance.
4. Isolated multi-server instances can be deployed with diversity in a large scale enabled distributed system environment.
5. Ledger maintained with Integrity, Irreversibility or immutability Guarantees, fully auditable and verified by any user (ex., client) or any third party.
6. Based on a permissionless Blockchain enabled Model: ledger usable as an open-service to support client-wallets with different security guarantees.
7. Support for Smart-Contracts, transactions can be done under smart-contract enabled verifications.
8. Incentivization mechanism, with fairness and sustainability conditions.
9. Possible anonymity of users and their transactions.
10. Possible privacy-preservation of data in transfer transactions
11. Support for the integrity of funds and balances of any user (as UTXOs), w/ counter-measures against double or multi-spending.
12. Transactions stored in the ledger maintain authenticity and integrity proofs about the provenance and destination of transfers, allowing the use of these proofs for any necessary dispute or auditing verification, by third parties or witnesses.

2 GBWS System model and architecture



2.1 Proxy Component Role and API

The main role of the proxy service is broadcasting client request to bft replicas. Read-only request are processed internally, without consulting other services.

- **obtainValueTokens** `https://.../obtain @Post`
  - Parammeters: ProtectedRequest<ObtainRequestBody>
  - Return: RequestInfo
- **transferValueTokens** `https://.../transfer @Post`
  - Parammeters: ProtectedRequest<TransferRequestBody>
  - Return: RequestInfo
- **consultBalance** `https://.../balance @Post`
  - Parammeters: AuthenticatedRequest<ConsultBalanceRequestBody>
  - Return: Double balance
- **allTransactions** `https://.../transactions @Post`
  - Parammeters: AllTransactionsRequestBody
  - Return: Transaction[]
- **clientTransactions** `https://.../transactions/client @Post`
  - Parammeters: ClientTransactionsRequestBody
  - Return: Transaction[]
- **consultTestimonies** `https://.../testimonies/requestId @Get`
  - Parammeters: String requestId
  - Return: Testimony[]
- **startMiningAttempt** `https://.../block @Post`
  - Parammeters: int size
  - Return: MiningAttemptData
- **submitMiningAttempt** `https://.../block/size @Get`
  - Parammeters: ProtectedRequest<MineRequestBody>
  - Return: RequestInfo
- **installSmartContract** `https://.../contract @Post`
  - Parammeters: ProtectedRequest<InstallContractRequestBody>
  - Return: RequestInfo
- **runSmartContract** `https://.../contract/run @Post`
  - Parammeters: ProtectedRequest<SmartTransferRequestBody>
  - Return: RequestInfo

2.2 BFT-Replica Component Role and API

The BFT Service nodes provide byzantine fault-tolerant state machine replication. All requests that alter the block-chain are handled here.

- **obtainValueTokens** `TLS v1.2`
  - Parammeters: ProtectedRequest<ObtainRequestBody>
  - Return: ReplicaReply
- **transferValueTokens** `TLS v1.2`
  - Parammeters: ProtectedRequest<TransferRequestBody>
  - Return: ReplicaReply
- **submitMiningAttempt** `TLS v1.2`
  - Parammeters: ProtectedRequest<MineRequestBody>
  - Return: ReplicaReply
- **installSmartContract** `TLS v1.2`
  - Parammeters: ProtectedRequest<InstallContractRequestBody>
  - Return: ReplicaReply
- **runSmartContract** `TLS v1.2`
  - Parammeters: ProtectedRequest<SmartTransferRequestBody>
  - Return: ReplicaReply

2.3 Contractor Component Role and API

The contractor service's role is to run smart contracts in a fully isolated environment. The smart-contract outcome is returned to BFT-Replica component for additional validation and block-chain persistence. This component's API is only available inside the node and is not publicly accessible.

- **runSmartContract** `https://.../contract @Post`
  - Parammeters: SmartTransferRequestBody
  - Return: Transaction[]

2.4 Wallet Component Role

Clients interact with this component to make requests. This component only communicates with the proxy component.

3 Adversary Model

The wallet is the sole component that can be trusted. All other components are regarded as being part of the attack surface area.

This solution employs techniques to secure communication channels and system components from masquerade, replay, black-hole, and modification attacks.

The system's doesn't offer any safeguards against denial-of-service attacks that originate internally from compromised or malicious Proxy/BFT-Replica components. Some measures were taken to mitigate the impact of denial-of-service attacks that target the peripheries of the system.

The system is designed in a manner that minimizes the threat of honest-but-curious adversaries.

4 Mechanisms and service planes

Client requests have different security requirements:

- `Protected<>` requests require the properties: [non-repudiation, integrity, access control and uniqueness]
- `Authenticated<>` requests require the properties: [non-repudiation, integrity, access control]
- Normal requests require the properties: [integrity]

The integrity of "Normal" requests is guaranteed by the https protocol. Because these request only evolve two parties: the wallet and proxy components, the integrity of the request can only be broken in the communication channel.

In order to achieve the properties of [integrity, non-repudiation and authentication], the required operations are signed with the affected wallet private key. The signature is validated using the client's public key; if it is invalid, the request is rejected on entry.

Access control is accomplished by the validation mechanism described above and by using the wallet public key as the client identifier. As an optimization, a hash of public key was used for smaller client ids.

An nonce is used to ensure the uniqueness of requests and to safeguard against replay attacks (for example, from a malicious proxy).

The nonce implemented is a request counter.The nonce must be kept in sync by the BFT-Replicas and Wallet components. BFT-Replicas save the most recently received nonce. A nonce is valid, if it is greater then the previous nonce. If the nonce is invalid, the request is rejected.

Replies and Testimonies ;

Requests that evolve only the Proxy component, return the outcome of operation directly in the reply.

Requests that evolve the BFT-Replica component are asynchronous. The reply of these requests is a request id, which clients can use to review the outcome of the operation at a later time. Requests are identified by a UUID that is generated in the proxy component. BFT-replicas reply to each relayed request with a signed testimony of the outcome of the operation. These testimonies are persisted in the Proxy component.

```
TestimonyData {
    String requestId;
    LedgerOperationType operationTypeEnum;
    String result;
    OffsetDateTime requestTimestamp;
}

Testimony {
    String requestId;
    OffsetDateTime testimonyTimestamp;
    TestimonyData data;
    String signature,
}
```

Testimonies can be used by intrusion detection systems to identify compromised BFT-Replica components.

Persistence ;

Each Proxy component contains a persistent repository of closed transactions, the "Transaction Ledger".

This repository is updated from the replies of requests to BFT-Replica component. The replies contain not only the outcome of a client request, but also, N closed transactions the repository has missing.

To ensure correct system behaviour under a byzantine model, a transaction is only recorded in the ledger, if it is supplied by 3f+1 BFT-Replicas. `3f+1 = ceil(NumberNodesView + 1)/3)`

```
Transaction {
    long id;
    byte[] walletId;
    double amount;
    byte[] hashPreviousBlockTransaction;
}
```

Confidentiality ;

To strengthen the confidentiality of clients evolved in a request, the schema of a transaction contains only the identification of one wallet. Requests involving N wallets, produce N transactions.

Inference method's that link two transactions with opposing amounts, are weakened, if the system closes around the same moment, a significant number of transactions with the same amount.

To further weaken inference, transactions can be split into multiple transactions with standardized amounts.
eg. A transfer from Bob to Alice of 239 coins would result in the following transactions: Bob:[-100,-100,-50, 10, 1] Alice:[100, 100, 50, -10, -1]

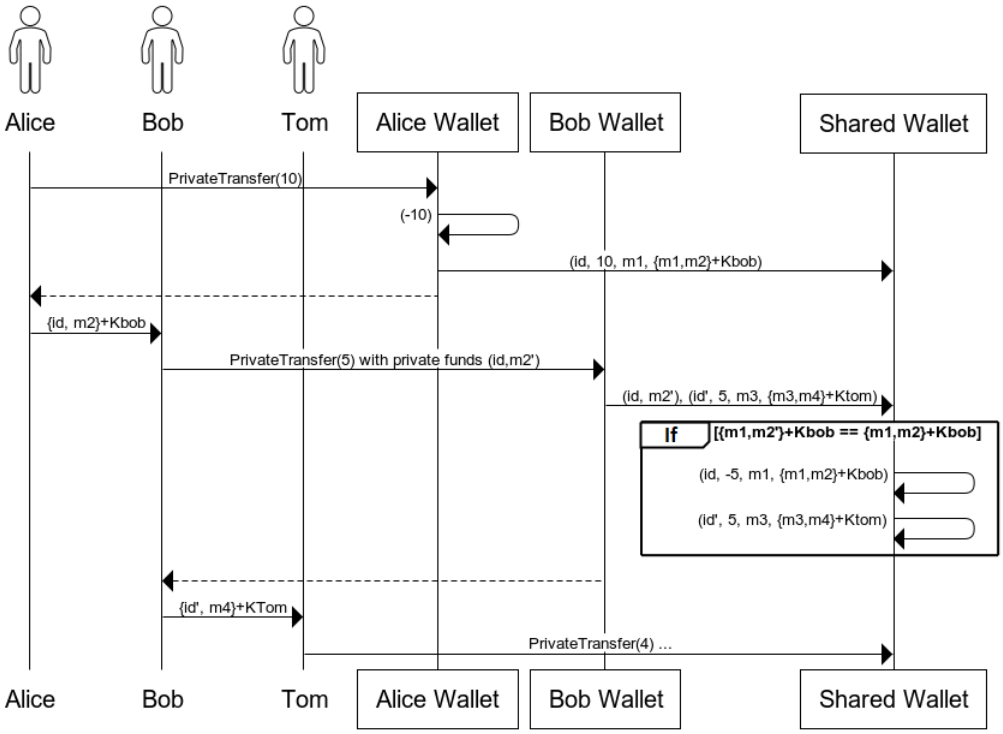
This scheme lowers the throughput, due to a substantial increase in the number of resulting open transactions per request. Due to this drawback, the back-end components don't enforce this splitting scheme, it is up to the wallets to encourage clients to make transfers in standardized amounts. In case the transfer involves a non standardized amount the splitting scheme described above is applied, with a fee to the party's that receive the amount's.

Back-end services can also postpone closing transactions that require privacy, until N open transactions with the same value are requested. This delay mechanism was not implemented; it is simply a suggestion for future improvements.

The implemented solution described above, doesn't provide confidentiality of a wallet balance, a better solution was conceptualized but not implemented.

To achieve confidentiality of a wallet balance, the wallet is shared by N clients. To make a private transfer, Alice would deposit funds in a shared wallet with a challenge that can only be solved by Bob. When Bob wants to spend the value, he solves the challenge and requests the shared wallet to make a transfer on his behalf. For secrecy to be kept the challenge cannot be used to identify Bob.

In the following scheme the challenge is presented as a cyphertext of two concatenated messages, only one message is publicly known, the two messages were encrypted with Bob's public key and the hidden message is sent off-chain to Bob. For Bob to solve the challenge he must send to the Shared Wallet the hidden message. This way even if the hidden message is disclosed the value can only be claimed in Bob's wallet. And because operations that deduct balance from wallet need a signature, the value cannot be stolen.



Under this system, the balance of Bob and Tom is never revealed. Confidentiality can only be broken, if an attacker listens to the initial request that placed the value in the escrow wallet. If an attacker remembers which wallet placed a amount in the shared wallet and waits for someone to successfully retrieve it, confidentiality is not secured.

Mineration :

Clients can participate in the system by closing transactions. They are incentivized to close transactions by receiving a substantial reward, when they submit a valid block that is accepted and added to the blockchain.

```
Block {
    long id;
    int version;
    int numberOfTransactions;
    OffsetDateTime timestamp;
    String previousBlockHash;
    TypePoF typePoF;
    int difficulty;
    String proof;
    private List < Transaction > transactions;
}
```

A block is valid if:

1. It doesn't contain previously closed transactions.
2. It includes an ordered sequence of open transactions, with no missing transactions in the middle or beginning.
3. The proof is valid for PoF method and difficulty.
4. The previousBlockHash is equal to the last mined block hash.
5. The PoF method, difficulty and version are equal to the ones in last block. (In order to modify these settings, a special type of block that ignores this rule is required.)

Clients can request a sub-sequence of transactions to create a block from the Proxy component, which holds a cache of N sorted open transactions. Open transactions are sorted by timestamp ascending, amount ascending, and wallet id ascending in BFT-Components. When the Proxy component sends a request to a BFT-Component node, the reply includes a list of open transactions, which is utilized to refresh the cache.

Requests for open transactions are not forwarded to BFT-Component nodes, in order to confine denial-of-service attacks, that exploit this method, to only the targeted Proxy components. (This method would be favored by attackers for denial-of-service attacks, because it may potentially return a large amount of data).

The implementation supports blocks with a configurable number of transactions. If two valid block submissions are received, the one that is favored is the one with the most transactions. If there is a tie in the number of transactions, the block with the smallest proof is chosen; if the two proofs have the same length, the one chosen is the one submitted by the wallet with the smallest id. To combat DoS attacks, the Proxy component also validates the block before propagating it to the BFT-Replica nodes. If a client submits a block with an invalid proof, the client must pay a fee; this discourages methods that attempt to partially solve a PoF and hope to get fortunate by sending repeated requests. The implementation's default mechanism is proof of work, but it can be easily extended to support other types of PoF.

Smart contracts :

Smart contracts can freely be installed by clients. When smart contracts are installed, they are not evaluated; instead, they are declared invalid if a java policy permission is infringed. When the system discovers an invalid smart contract, the contract is removed.

Smart-contracts are executed at least once, because they are not validated during installation, only at execution. To dissuade attackers from submitting malicious smart contracts repeatedly, a fee is required for smart contract installation.

Smart-contracts are executed in a dedicated component and have read-only access to closed transactions. Because there is no Java policy that allows control over allotted resources, smart contracts must be run in a dedicated JVM. If contracts are allowed to be executed in a non-isolated environment, a malicious contract could make the service unresponsive by allocating all available memory.

The implementation supports smart-contracts with an arbitrary number of parameters. The execution of smart-contracts has no side effects, with only outputs a list of transactions that are further validated by BFT-Components. Output transactions are only valid if the following conditions are met:

- The sum of the amount of each transaction is zero.
- Negative-amount transactions only apply to the wallet's that issued the request.

Non-deterministic operations are not permitted in smart contracts; if they were, each node would output a distinct set of transactions, making it impossible to reach a consensus.

All networking and I/O operations are also restricted. Operations that change the environment's configuration are likewise banned.

5 Implementation issues and improvements

Transaction ledger consistency There will be inconsistencies when submitting queries to distinct proxy's, because their transaction repository's are eventually consistent. A way to solve this problem, is for clients to send with their request, the last known transaction. When the proxy's receive a request, it checks to see whether the transaction is known; if it is not, it may redirect the request to a more updated node or take the initiative by requesting an update from BFT-Replica component before replying to the client. This mechanism was not implemented due to lack of time and other priority's.

Smart contract serialization The serialization mechanism for smart contracts was researched but was not successfully implemented. To achieve class definition serialization in java, the compiled java "Target".class file needs to be serialized in bytes, in conjunction with the full class name and package. To deserialize the target class, it must be first defined by calling "defineClass(className, bytes, 0, bytes.length)" in a ClassLoader instance. A instance of the target class can be made by calling newInstance() on the return of the method defineClass;

6 Experimental observations and analysis

All experiments were made in a Windows machine with:
CPU: Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz
Memory: 16Gb DDR4 Ram.

The following metrics were taken in a dockerized deployment with 7 BFT-Replica nodes, 4 Proxy node, 0 Contractor nodes and 2 client's. To obtain these metrics, Five tests of 1000 requests were conducted, these metrics represent the average of those tests.

6.1 Latency of Rest Operations

.

[concultBalance]
Average Latency: 243 ms
Standard Deviation: 5.04043 ms

[obtainValueTokens]
Average Latency: 257 ms
Standard Deviation: 13.92289 ms

6.2 Throughput

.

[concultBalance]
Average Throughput: 4.11522 operation/s

[obtainValueTokens]
Average Throughput: 3.89105 operation/s

6.3 Latency of successful mining

These measurements were taken with a single miner and a single client issuing synchronous requests.

Average Latency: 3138 ms
Standard Deviation: 3019.9646 ms

The poor mining metrics may be explained by the serialization methods used in implementation. Before being signed, Java objects are converted to JSON strings and then to byte[] using the Standard.Charset(UTF8).