CSD 2020/21 Report
BGWS System

Rodrigo Ribeiro

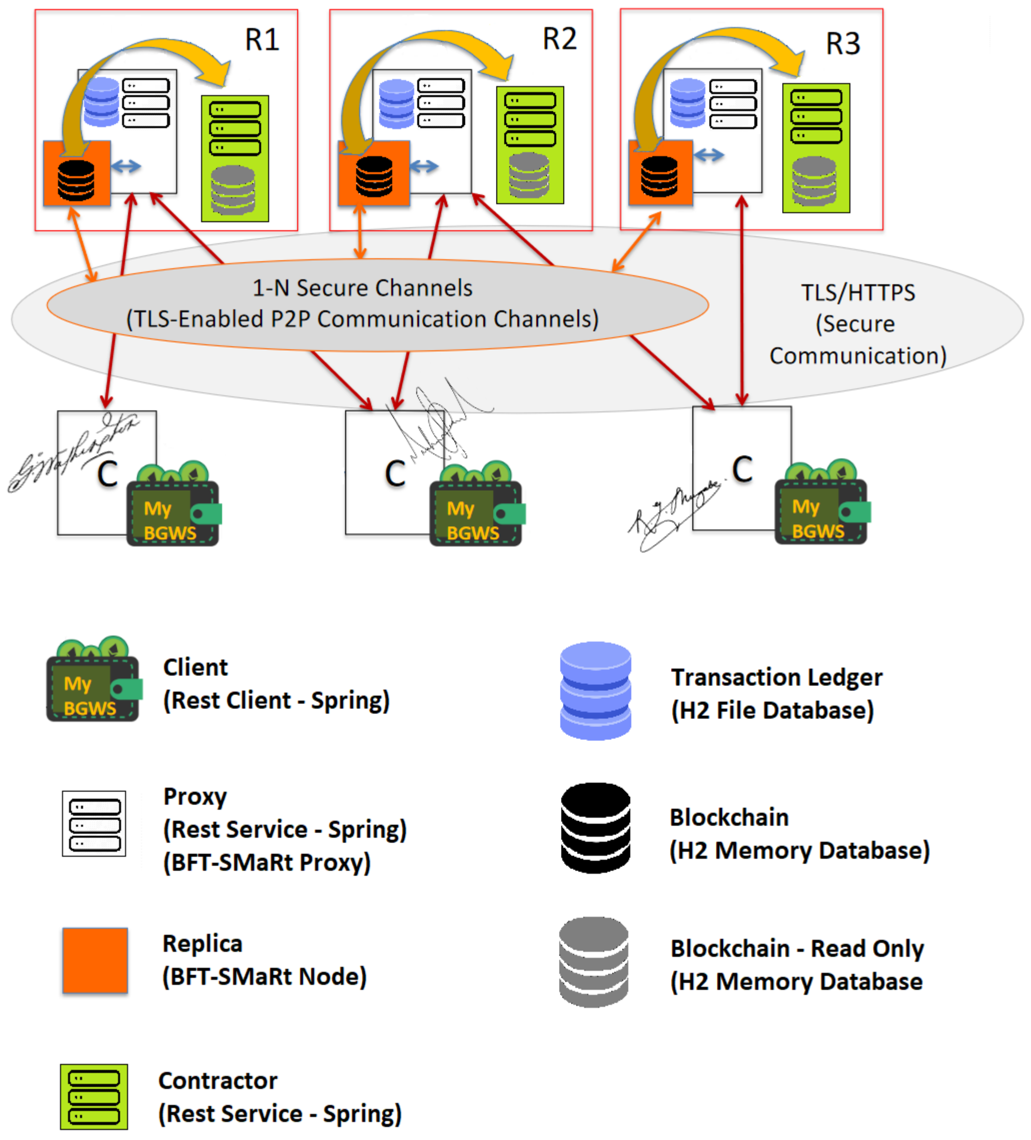
NOVA School of Science and Technology, Largo da Torre, 2825-149 Caparica, PT
https://www.fct.unl.pt/

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.

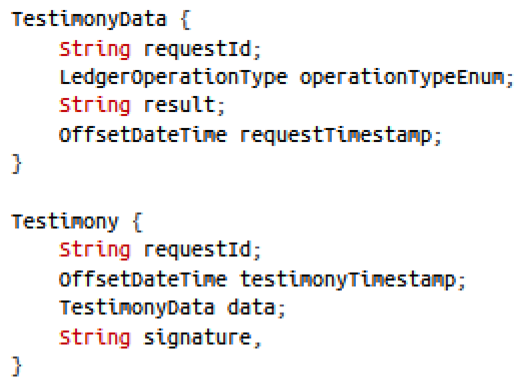
An nonce is used to ensure the uniqueness of requests and to safeguard against replay attacks (for example, from a malicious proxy).

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,
}
```

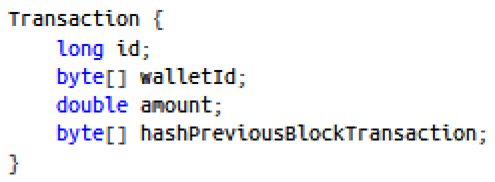
The testimony schema doesn't include details of the operation, in order to preserve the privacy of the clients evolved. Since the information contained in the testimonies repository is not confidential, it can be used freely by intrusion detection systems to identify compromised BFT-Replica components.

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.



```
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.

There is no obvious inference method to connect two transactions, if the system receives a significant number of requests from different clients with the same amount.

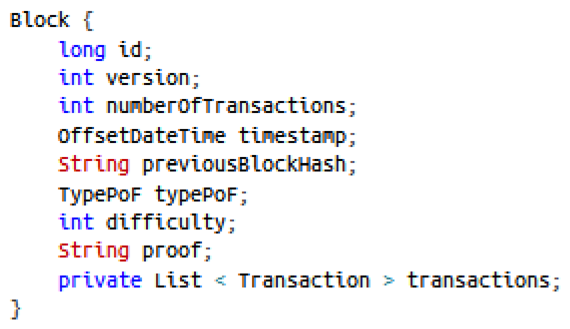
To break inference, transactions can be split into multiple transactions with standardized amounts.

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 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 that receives the coins.

The back-end services can also

Block schema



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

Mineration

Smart contracts

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 not 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;

Testimonies replication Each node's testimony database should be replicated, across other nodes.

6 Experimental observations and analysis

(All results and experimental validations and analysis go in this section. This includes experimental data and resulsts, plots/graphics, tables or analysis as valid and observed "answers" to evaluation and validation questions.Make sure "to have a question" before each analysis/answer in your report. Paraphrasing is okay, as long asit is clear what it is you are answering and what you can argument as a valid observation to show that you have a solution for your "problem". Important results should be clearly represented (units, metrics, meaning, etc) and it must be clear that the way you measured is correct for what you are trying to present as validations

7 Coverage of the requirements

Summarizeherethecoverage(emphasiningthecompletness, correctness, drawbacks, limitationsortradeofs) formtheexpectedrequirements

References