Implementing Private Data Collections in Hyperledger Fabric

A channel is a private subset of communication between two or more organizations. The data stored inside channels will be available for all the peers of an organization or a group of organizations. In cases where a group of organizations on a channel need to keep data private from other organizations on same channel, they have the option to create a new channel. The problem with this approach is generation of duplicate data and creation of web of channels among the group of organizations. This makes tracking communication difficult and confusing.

The current version of Hyperledger Fabric introduces private data stored in SideDBs. This solution offers the option to build blockchain solutions that are compliant with GDPR.

Advantages of Private Data Collections

Private data collection is a way to keep certain data/transactions confidential among members sharing the same channel. Only authorized peers will see the hash of the data on the main ledger and the actual data in the private database. Unauthorized peers will not have the private database synchronized and such members can only view the hash on the ledger. Since hashes are irreversible, they will not be able to view the actual data.

Implementation of Private collections

Step 1

Firstly, we need a collections configuration file “collections_config.json” containing collection name and policy. The policy is similar to an endorsement. We can define our own policy based on the organization’s structure.

```
[
{
"name": "collectionSensor",
policy": "OR ('Org1MSP.member','Org2MSP.member',,'Org2MSP.member')",
"requiredPeerCount": 0,
"maxPeerCount": 3,
"blockToLive":1000000
}
]
```

‘blockToLive’ parameter defines the maximum blocks after which data will be purged automatically. A value of 0 represents ‘never purge’.

Step 2

This step involves writing chaincode. In Hyperledger web application demo, we have implemented go language chaincode which helps to store private data on collections.

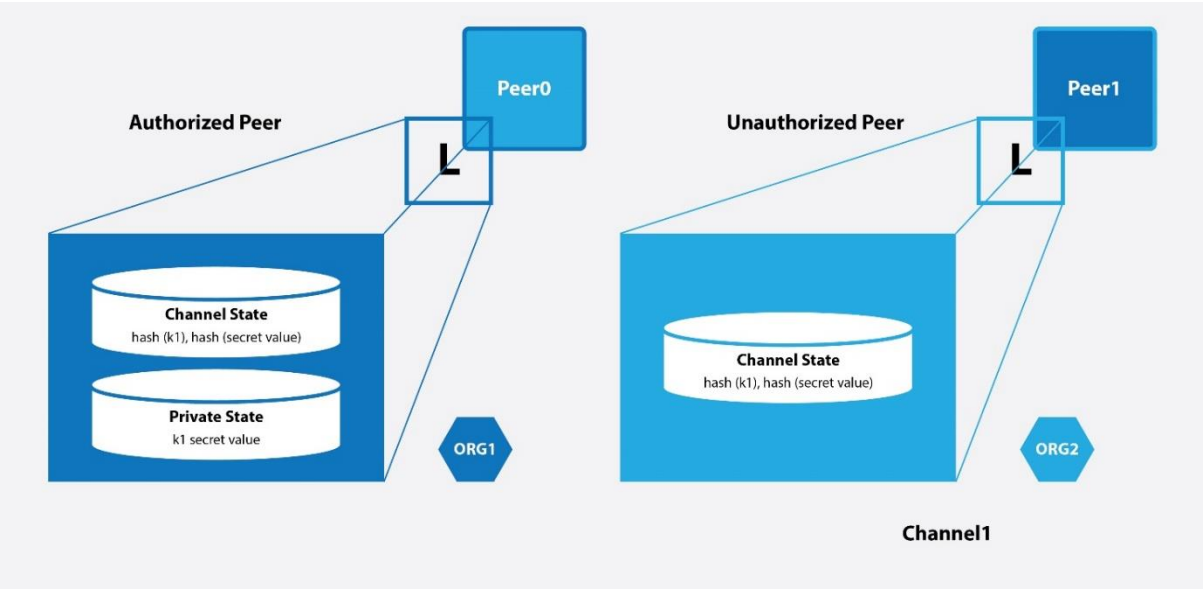
```
func (t *SensordataChaincode) AddSensor(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
        sensorId := args[0]
        location := args[1]
        latitude := args[2]
        longitude := args[3]
      orderId := args[4]

        // ==== Check if Sensor already exists ====
        sensordataAsBytes, err := stub.GetPrivateData("collectionSensor", sensorId)
        if err != nil {
                return shim.Error("Failed to get sensor: " + err.Error())
        } else if sensordataAsBytes != nil {
                fmt.Println("This sensor already exists: " + sensorId)
                return shim.Error("This sensor already exists: " + sensorId)
        }

        // ==== Create sensor object and marshal to JSON ====
        objectType := "privateasset"
sensordata := &sensordata{objectType, sensorId,location,latitude,longitude,orderId}
        sensordataJSONasBytes, err := json.Marshal(sensordata)
        if err != nil {
                return shim.Error(err.Error())
        }

        // ==== Save sensor private details ====
        err = stub.PutPrivateData("collectionSensor", sensorId, sensordataJSONasBytes)
        if err != nil {
                return shim.Error(err.Error())
        }
```

The following picture shows the ledger contents of a peer authorized to have private data and the one without the access to it.



Tools and Modules

We have used various tools and node modules to deploy node application on Hyperledger fabric 1.2.

The table contains the list of tools we used in both Hyperledger fabric V1.1 and V1.2

	Hyperledger fabric-1.1	Hyperledger fabric-1.2
Docker-Engine	18.06.1	18.06.1
Docker-compose	1.13.0	1.18.0
NPM	5.6.0	5.6.0
Node	8.11.3	8.14.0
Composer-playground	0.19.0	0.20.0-0
Composer-restapi	0.19.0	0.20.0-0
Blockexplorer	3.2	3.7
Composer-cli	0.19.0	0.20.0-0
Composer-admin	0.19.0	0.20.0-0
Composer-client	0.19.0	0.20.0-0
Composer-common	0.19.0	0.20.0-0
Composer-connector-hlfv1	0.19.0	0.20.0-0
Composer-connector-embedded	0.19.0	0.20.0-0
Grpc	1.10.1	1.10.1

Key Issues and Learnings

While deploying the application, we found many issues and solved them in an efficient manner.

1. **Error:** "Failed to connect client peer, please check the configuration and peer status"

The error was identified while launching the Block explorer. To solve the above error, we have added extra environment variables to peer configuration 'yaml' environment section.

CORE_PEER_GOSSIP_BOOTSTRAP=peer0.org1.example.com:7051

CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer0.org1.example.com:7051

2. **TypeError:** "Cannot read property 'size' of undefined in Block explorer"

The connection profile structure was modified from 1.1 to 1.2, hence we used latest configuration for connecting to peer.

3. **Error: "**Composer-connector-hlfv1 fails to load for connection profile hlfv1 when deploying BNA file on cloud server"

After launching node web application, the system was not able to establish a connection with the network and this error was detected. We have modified node modules in 'package.json' as shown in the above list.

Contact for more details:

Narasimha Murty Sesetti

Technology Analyst

narasimhas.in@mouritech.com

MOURI Tech