BRICS PAY CONSORTIUM

TECHNICAL DOCUMENTATION

DECENTRALISED CROSS-BORDER MESSAGING SYSTEM

Update
March 03, 2025

team@brics-pay.com

TABLE OF CONTENTS

1. GENERAL INFORMATION

Decentralised Cross-border Messaging System, hereinafter referred to as **DCMS,** is a software package that provides decentralized exchange of financial messages between two or more participants in SWIFT message standard. The software package is a convenient, secure and low-cost tool for interbank settlements.

The key difference of the system is the impossibility of forced exclusion of participants, since there is no single owner.

DCMS is an alternative channel for transmitting financial messages and does not exempt from compliance with regulatory rules.

One of the most important features is the platform architecture, which allows achieving high performance and horizontal scalability, unattainable in a classic distributed ledger based on blockchain.

DCMS links business processes of transmitting financial messages and liquidity. In other words, the transfer of a financial message goes through nodes that are correspondents in a future transaction.

Documentation includes:

- Node configuration file
- Swagger http://dcms.dev.dltc.spbu.ru:3001/api#/

2. TECHNICAL REQUIREMENTS

The project is written using the languages Go, Javascript, Solidity.

Criteria for disk space. The project code takes up ~30 MB (excluding third-party libraries and version control systems). The required disk space is calculated from the planned message storage volume.

Can be run on any modern Unix system. When deploying via containerization, an additional 4 Gb of free space is required.

3. PROJECT ARCHITECTURE

The system is based on a multi-component (microservice) architecture.

3.1. SYSTEM ENTITIES

Private Key

A key used in asymmetric encryption. A part of a key pair that is not published in the public domain. Can be used for both encryption and decryption, depending on the algorithms used.

Public Key

A key used in asymmetric encryption. A part of a key pair that is not published in the public domain. Can be used for both encryption and decryption, depending on the algorithms used.

Node

A deployed instance of the system kernel with loaded keys of a system participant

System Participant

A system participant is a person (natural or legal) who has control over the node(s) and the key(s)

Network Address

The IP address and port of the running instance of the system kernel

Node Address

A fingerprint of the public key is used for addressing within the DCMS

Currency

A cross-section of accounting units in the system

Main currency

The currency in relation to which all rates are set

Currency rates

Coefficients in relation to the main currency for calculating limits and currency conversion rates

Agreements

The main essence of the system. An agreement can be concluded between a pair of participants and means trusting the public keys of the participants of the pair to each other.

Limits

When concluding an agreement, it is assumed that limits are set by each party to the agreement. Limits are set both generally and for each currency.

Setting a limit in a specific currency means the possibility of transferring liquidity between the parties to the agreement. It is also a risk management tool and a basis for auto clearing.

DCMS network

A set of nodes in the system that form a linked graph. There may be a number of independent DCMS networks.

Balances

Balances formed in the process of transmitting messages for agreements

WSS channel

Secure channel for exchanging messages between parties to agreements

Message

A message about the transfer of funds. The recipient can be any node of the DCMS network. It can be informational if the amount specified in the message is 0.

Information message (Intention)

Information message about the transfer of funds. The recipient can only be a node from the list of agreements.

3.2. SYSTEM COMPONENTS

The system is designed for easy integration into external systems and has a set of replaceable components.

Main components of the system and functional tasks.

Core

1. Stores and processes agreements and their parameters

2. Provides a mechanism for transmitting messages, including deferred ones.

3. Provides accounting of balances

4. Sends messages to the storage service

5. Sends information to the publication service

6. Limit request mechanism (request, acceptance, rejection) - no stages, accounts, etc.

7. Deployed for each participant in the system

Message storage service

1. Stores messages and their status (delivered, not delivered, awaiting sending)

2. Deployed for each participant in the system

3. Mechanism for receiving related messages

4. Deployed for each participant in the system, but can be used in general if there is access filtering across nodes

Access to the message storage service is provided via Rest API calls. This service is designed with the ability to replace or adapt to already used storage services. The architecture provides for replacement with another server implementing the interface described in the OpenApi scheme in the **configs/ms-api.json** file.

Information publishing service

1. A layer for connecting to a common public layer, standardizing the interaction of the core to provide publicly available information, in particular: about node detection, conversion rates and exceptional situations

2. Deployed for each participant in the system, but can be used in general if there is access filtering in terms of nodes

Like the message storage server, interaction with the service is carried out via the REST API, and it is also possible to implement your own component.

Common public layer

1. Stores information published via the publishing service.

2. Characterized by low performance.

3. Assumes the use of a public or private blockchain network.

Below we list the auxiliary components of the system. Auxiliary components solve a number of problems:
1. Simplify interaction with the system.
2. Solve a specific business problem.

Client server

1. Offers the ability to easily deploy a node

2. Offers access roles to the deployed node - in addition to interfaces for adding and blocking users, contains a repository that determines the correspondence of authorization data, node and role

3. Offers a public ability to deploy a node with a given interface, node in the agreement

4. Offers a full* software interface for interaction with the corresponding user node of the system

5. Offers an interface for managing information for generating an invoice.

The server is a layer that allows you to fully manage your node and in the standard delivery does not contain logic for access restrictions to the core.

System interface

1. Simplifies the initial deployment of the node
2. Displays the system interface corresponding to the user role
3. Gets a list of related messages in terms of the limit request
4. Calculates the stage of the agreement on the limit request based on the received messages.
5. Calculates the available limit for transfers.
6. Generates messages with additional fields (not standardized by the core), including files, BIC, SWIFT code, etc.

These fields are encoded in json and sent in the body. File encoding is applied in RAW or base64.

3.3. CORE ARCHITECTURE

The system core is written in the Go programming language and contains the following logical and functional components implemented by packages.

Components

1. Message component
   Defines the structure of transmitted messages
   a. Message sending component

2. Agreement component

3. Balance accounting component
4. Node connection component
5. Message processing service
   a. Message sending service
   b. Message receiving service
   c. Message forwarding service
6. Node connection maintenance service
7. Command processing service
   a. CLI interface (depricated)
   b. TCP interface
   c. gRPC interface (in future versions)
8. Node service startup script
9. Settings storage component
   a. System behavior settings
      i. General behavior settings (encryption mechanisms, paths to discovery services)
      ii. Settings taken out into agreements (e.g. encryption mechanism)
      iii. Sending and forwarding settings
      iv. Key storage
      v. Business settings (conversion rates, etc.)
   b. Agreement settings
10. Message encryption component
    a. Encryption implementation component
11. Logging component
12. Detection service interaction component
13. Message storage service interaction component

Packages:

- **app/**
  Contains the basic types shared by the components and the application instance that is launched, which, depending on the passed strategy, builds and launches the node.
- **components/**
  Contains packages of components from which strategies are assembled. Each component must satisfy the interface.
  Component or ActiveComponent from the app/ package
- **components/ui**
  Contains user interface packages for accessing the core, as well as auxiliary functions and structures.

As of the date of this document, an interface has been implemented over the tpc protocol that accepts json messages of the form

```
{
"type": "<command>",
"payload": <command argument>
}
```

It is possible to connect to a socket to send commands, for example, using the netcat software utility.

```
foo@bar$: nc -l 127.0.0.1:<node command port>
```

- **components/transferMessage/encoder**
  Contains message encryption algorithms. Each algorithm is implemented by a structure that satisfies the MEA interface from this package. After adding a new encryption algorithm, it must be registered in the encoder component

- **exceptions/**
  Contains user errors

- **loader/**
  The package contains the system kernel configuration loader.

- **logger/**
  Contains several logger implementations that satisfy the Logger interface from the app/ package.

- **state/**
  Contains a version of the application state implementation. The state stores the necessary information for processing messages and the logic for saving the state to disk. Each state implementation must satisfy the State interface from the app/ package.

- **store/**
  Contains implementations of message storage access components. Implementations must satisfy the Store interface from the app/ package.

- **strategies/**
  Contains possible strategies for launching the application. Strategies must satisfy the Strategy interface from the app/ package.

- **tests/**
  Contains e2e tests, as well as helper packages mock/ and helpers/

The helpers/ package contains an alternative version of the strategy builder, allowing for easy and transparent management of multiple nodes via code.
utils/
Contains common helper functions

Kernel Modification Options

The core of the system is written in a component style, which means that many of its parts can be replaced, for example:

- The message acceptance component can act as a gateway, or it can, for example, wait for a manual message acceptance
- The system state component can be replaced with a Redis-based component
- The core can be controlled via other protocols, such as gRPC or http

To create and use your own system components, you need to create a new strategy in the strategies package

Based on the data model, the system is implemented in a reactive style, so each component is a set of subjects and subscribers that are linked by a strategy. The strategy can also implement dependency injection.

After creating a strategy and describing the component connections, you should create a runtime package in the cmd/ directory. It is delegated to load configurations and initiate state components, message storage service access components, and a logger.

Also, when a node is started, an object of the context.Context type is set, from which all node components inherit their contexts.

Disabling a context entails stopping all application components.

4. MESSAGE FORMATS

Payment message (Payment)

When creating a message, the sender fills in the following fields:

- **Receiver** (string 32 byte) — unique identifier of the recipient node
- **Currency** (string 3 byte) — currency name
- **Amount** (float64 8 byte) — amount of currency to be transferred in the specified currency
- **Text** (string) — meta information that can be encrypted so that only the recipient can read it. The size may be limited by the encryption mechanism chosen.
- **MaxFee** (float64 8 byte) — the maximum fee the sender is willing to pay. If it is impossible to send a message with the specified MaxFee, the message will not be sent.

- **Encoding** (string 4 byte) — a string corresponding to the available encryption algorithm with which the Text field will be encrypted
- **TransferTimeout** (float64 8 byte) — the maximum sending date, until which time the message can be transmitted via the WSS channel. Specifies the time in seconds after which the message will be discarded if it has not reached the recipient due to the lack of a path or insufficient commission level MaxFee.
- **ConfirmationTimeout** (float64 8 byte) — the deadline for receiving confirmation, before this date it is possible to receive confirmation via WSS. Specifies the time in seconds after which the message must be published in the Detection Service, otherwise the message will not reach the sender.
- **DiscoveryTimeout** (float64 8 byte) — the deadline for receiving confirmation from the detection service, before this date it is possible to receive confirmation from the detection service. Specifies the time in seconds after which the message should be discarded if the sender has not received confirmation of the transaction. After this period, the message is canceled.

  When transmitting between nodes in search of a path to the recipient, the message (Transfer) has the following fields:
  - **Id** (string 16 byte) — a message identifier that does not change when the message is transmitted.
  - **Receiver** (string 32 byte) — unique identifier of the recipient node specified by the sender.
  - **Currency** (string 3 byte) — currency name specified by the sender.
  - **Amount** (float64 8 byte) — the amount of currency to be transferred, specified by the sender.
  - **Path** ([]string) — list of nodes involved in message transmission
  - **Text** (string) — meta information specified by the sender
  - **TransferDeadline** (time.Time 24 byte) — timestamp after which a message that has not reached its recipient will be discarded.
  - **ConfirmationDeadline** (time.Time 24 byte) — timestamp after which the message that reached the recipient will be published in the Discovery Service.
  - **DiscoveryDeadline** (time.Time 24 byte) — timestamp after which a message whose acknowledgement has not reached the sender will be discarded and cancelled by all nodes.
  - **Encoding** (string 4 byte) — the encryption algorithm with which the Text field is encrypted
  - **MaxFee** (float64 8 byte) — the remainder of the commission that the transit hub can take.

When the message is received by the recipient, the recipient signs and sends a confirmation (Confirmation) to the sender with the following fields:

- **Id** (string 16 байт) — a message identifier that does not change when the message is transmitted.
- **ConfirmationDeadline** (time.Time 24 byte) — timestamp after which the message, having reached the recipient, will be published in the Discovery Service.
- **DiscoveryDeadline** (time.Time 24 byte) — timestamp after which a message whose acknowledgement has not reached the sender will be discarded and cancelled by all nodes.
- **MaxFee** (float64 8 byte) — the remainder of the commission that the sender does not pay. Accordingly, the commission that the sender must pay is equal to the difference between the MaxFee specified by the sender when creating the message and this remainder value.
- **Path** ([]string) — the full path from the sender to the recipient that the message passed through.
- **Receiver** (string 32 byte) — unique identifier of the recipient node specified by the sender.
- **Cert** (string 1172 byte) — public certificate of the sender to verify the signature, in case the transit node does not know the recipient.
- **ReceiversSign** (string 344 byte) — signature of the recipient of the confirmation.

Non-payment messages

Non-payment messages for exchanging documents or agreeing on a contract between nodes (Intention) contain the following fields:

- **Id** — message group id
- **Source** — unique identifier of the node that sent this message.
- **Dest** — unique identifier of the node that received this message.
- **Currency** — currency name
- **Amount** — amount of currency to reconcile balances between nodes in the specified currency.
- **Done** — is the message complete
- **State** — a meta number describing the state of a group of messages with the specified Id.

When transmitting messages between nodes, the message is encoded and written into a Transaction with the following fields:

- **Id** — unique transaction identifier
- **Type** — a string that takes one of the following values:
  - "transfer" (1.b)
  - "confirmation" (1.c)
  - "Intention" (2)

- **Dest** — unique identifier of the intermediate node that received this transaction.
- **Source** — unique identifier of the intermediate node that sent this transaction.
- **Sign** (string 344 byte) — signature of a transaction by an intermediate node that sent the transaction.
- **Payload** — coded message (Transfer, Conformation или Intention)

4.1. CONFIGURATION FILE (SETTINGS)

The configuration file can be initially generated when the node is deployed. The file contains the node's private key, currency settings, and agreements.

4.2. MULTILINGUALISM

Multilingualism concerns only the client interface of the node. The current system has 2 localization files "Russian" and "English". It is possible to add new localization settings in src/utils/languages.ts.

5. VULNERABILITY ANALYSIS

Possible vulnerabilities and methods for eliminating them.

The main vulnerabilities under consideration are related to the possible modification of an attacker-controlled node.

Vulnerability	**What will happen**	**Why**
Send a payment message to a node with which there is no agreement	The message will be discarded.	Nodes do not accept payment messages from nodes with which an agreement has not been established.
They will send a payment message with an incorrect recipient.	The message will be discarded.	The message will not find a recipient and will be discarded after TransferTimeout.
Substitution of one or more fields id, currency, amount, Receiver, ConfirmationDeadline, DiscoveryDeadline	It will be noticed when sending confirmation and the transaction will be cancelled due to a breakdown.	The confirmation fields are signed by the recipient, if replaced again the signature will be incorrect. And if there is no repeated substitution, the other node will notice that the fields before confirmation and

during message transmission		during confirmation are different and will cancel the message.
Пpath override during message transmission	It will be noticed when sending confirmation and the transaction will be cancelled due to a breakdown.	If the path block is damaged, one of the nodes will not be able to decrypt it and will cancel the message.
In case of substitution of MaxFee in the higher direction	It will be noticed when sending the confirmation and the transaction will be cancelled due to a breakdown or the node that substituted the message will suffer losses	If a node replaces the MaxFee field again, the signature will be invalid and the message will be canceled or the node that performed the replacement will pay part of the sender's fee
In case of substitution of MaxFee to the lower side	The message will go through without errors.	In this case, the node debited the commission from the sender.
In case of replacing MaxFee with a negative value	The message will be discarded or cancelled.	When processing transfer and confirmation, MaxFee is checked. If the value is negative, the message will be discarded or canceled accordingly.
If the recipient and other nodes team up and decide to change fields in the message. When substituting, the alliance can also change the confirmation signature.	The message will reach the sender without any problems. Attackers will not be able to make money on it	The sender and other nodes will not lose anything, and the recipient will receive funds from the nodes with which it has merged, which is disadvantageous for these nodes.

In case of a change in the node code by one or more participants in the chain, the following mechanisms are provided to protect payments from intruders:

When creating an agreement using Intention messages, you can agree on the terms of the agreement between nodes. These messages allow you to exchange documents for the purpose of concluding agreements and formalizing the legal relations of the participants. In addition, all transactions between nodes linked by the agreement are signed. If one of the parties refuses to pay the funds transferred in the system, the other party can present the agreement and a list of signed transactions to confirm the transfers.

Due to the signature of the confirmation by the recipient, the fields cannot be changed when the confirmation is transmitted to the sender, since the signature of the recipient is verified with each transaction transmission. If the fields were changed before the confirmation was signed, then after the confirmation was signed, one of the nodes in the message path, including the sender, will notice the substitution and notify the other nodes about the cancellation of the message. In this way, the substitution of message fields by one or more participants is detected.

If a node is disconnected from the network, the network will continue to operate without the disconnected node. Even if the node is involved in the transfer of funds, but it disconnected without waiting for confirmation, the message will either wait for the node to connect to the network or the message will be sent to the Discovery Service. Then the disconnected node will be able to recover messages from the Discovery Service.

6. SYSTEM NODE DEPLOYMENT

6.1 ASSEMBLY WITH A DOCKER

Loading the necessary repositories and starting a cluster consisting of 4 nodes with private message stores is done by a script

autostart.sh

On the main branch of the dcms-docker-build repository.

The container set will use the following ports

> :8000 - general service detection port
>
> :300[1-4] - server port. This port can be used to access the REST API and dynamic web page
>
> :400[-4] - message store port. This port can be used to access the API of the message store

6.2 ASSEMBLY WITHOUT A DOCKER

1. Node

The source path of the node is located in the dcms-node repository

The node startup file is located in /cmd/main/main.go

The node needs to be configured at startup. A list of existing configuration parameters:

There are several ways to configure the node startup

Using flags

The following flags are used:

- -**id** - Node name
- -**ip** - Host address
- -**wsport** - Wss port for receiving and transmitting messages
- -**port** - http port for connecting to other nodes
- -**command-port** - Commands reception port
- -**transfer-timeout** - Standard timeout for sending messages
- -**confirmation-timeout** - Standard timeout for receiving confirmations
- -**discovery-timeout** - Standard timeout for reading confirmations from the discovery service
- -**buffsize** - Size of message sending buffer
- -**discovery** - Discovery service address
- -**store** - Address of message store
- -**tls-cert** - Path to certificate
- -**tls-key** - Path to certificate key
- -**cwd** - Path to save temporary files
- -**state** - Path to save node state
- -**save-state** - Whether to save node states to disc
- -**allow-transit** - Whether to allow transmissions of transit messages
- -**allow-cipher** - Whether to allow transit transmission of encrypted messages
- -**allow-gateway** - Whether to convert currency
- -**gateway-stake** - The portion of the payment that is included in the currency conversion fee.
- -**discovery-poll-interval** - Frequency of accessing the discovery service to retrieve lost confirmations (in seconds)
- Path to configuration file

Using the prompt bootloader

If you run the file without any additional arguments, the node will prompt you to enter the necessary parameters for launch manually.

Using a configuration file

If the only configuration parameter passed is config (path to the configuration file), then all other settings will be read from it.

The configuration file is a json file according to the scheme described in config/config-schema.json

It is also possible to start a node from a ready state file. To do this, place the saved state in the stdin of the process at startup.

Before starting a node, the discovery service and the message storage service must be running.

```
foo@bar$: go run cmd/main/main < saved_state.json
```

Some of the settings can be subsequently changed during the operation of the node using a special command.

2. Message storage

The message storage is a Rest Api service that performs CRUD operations on message entities.

OpenApi schema is stored in configs/ms-api.json

The implementation from dcms-message-storage accepts as settings only the port number to run and the path to the sqlite database file and reads them from the PORT and DB_PATH environment variables respectively.

Launch example:

```
PORT=12345 go run cmd/main.go
```

3. Discovery service

The discovery service is located in the repository dcms-discovery.

The discovery service accepts the following parameters as file launch flags cmd/main/main.go:

- **port** - server startup port
- **address** - EVM wallet address (private key) for smart contract execution
- **contract** - smart contract address for publishing and reading node information and messages
- **ip** - EVM compatible node address

Important note: *all the components described do not contain authorization and authentication mechanisms, and access security must be organized by other methods - physical or network, or through integration with internal authorization and authentication systems*.