УПРАВЛІННЯ В ОРГАНІЗАЦІЙНИХ СИСТЕМАХ

MANAGEMENT IN ORGANIZATIONAL SYSTEMS

DOI: 10.20998/2079-0023.2025.02.04
UDC 004.72

P. Y. ZHERZHERUNOV, Student, National Technical University "Kharkiv Polytechnic Institute", Kharkiv, Ukraine; e-mail: Pavlo.Zherzherunov@cs.khpi.edu.ua; ORCID: https://orcid.org/0009-0005-7240-9395
O. V. SHMATKO, Doctor of Philosophy (PhD), Docent, National Technical University "Kharkiv Polytechnic Institute", Ass. Prof of Software Engineering and Management Intelligent Technologies Department, Kharkiv, Ukraine, e mail: oleksandr.shmatko@khpi.edu.ua, ORCID: https://orcid.org/0000-0002-2426-900X

ARCHITECTURAL APPROACH TO DATA PROTECTION IN DISTRIBUTED SUPPLY CHAIN MANAGEMENT SYSTEM USING BLOCKCHAIN NODES

Dockerised blockchain solution can mitigate the low levels of distributed technology adoption in small and medium enterprises. It can be done via designing and implementing an environment which inherits ease of deployment and scalability of containerized systems with safety and transparency of distributed applications. Practical implementation of a dockerized blockchain solution designed as a demonstrative implementation for existing client–server architecture is described in this paper. This solution uses Docker containers to simplify the setup and deployment of a private blockchain network, a mediator server and a reverse proxy. Implementation of this system on a low scale demonstrates feasibility of integrating blockchain technology into existing business processes without fundamental architectural changes and acknowledges deployment and maintaining challenges that usually accompany distributed systems using private blockchain. Discussed implementation is a demonstration of designed architecture being potentially a reproducible and easily maintainable environment for logging and validating data through an immutable ledger on a smaller scale. Proof of concept successfully validates the core idea. The implementation shows a mediator server intercepting client request, recording them on a private Ethereum blockchain via a JSON-RPC interface, and then forwarding them to the original server. This confirms the solution's ability to introduce a trusted, intermediate layer for data immutability. The project demonstrates a working framework for embedding distributed ledger technologies into client–server ecosystems. While the current Proof of Work consensus mechanism presents scalability limitations, the architecture provides a strong foundation for future research, including migrating to more efficient consensus mechanisms and integrating smart contracts.

Keywords: dockerized blockchain architecture, supply chain management, containerized blockchain nodes, small-medium enterprises, supply chain, data protection in distributed system, blockchain proxy, hashing algorithms, ethereum.

Introduction. This paper provides a detailed practical overview of the fundamental layer of dockerised blockchain solution that allows for simplified process of setup and deployment of distributed tools into existing client–server architectures. This implementation overview is based on the architecture described in the previous research paper, which focuses on wrapping the blockchain network setup and connection in the Docker containers, connected into shared network, to enable easy and quick setup of these tools and integrating them into existing business processes [1].

Foundational part of the solution is a Docker tool, which enables networking and orchestration of several server-like containers, acting as separate virtual machines. It allows to construct a network of several servers with different purpose and blockchain nodes to simplify deployment and maintenance of all its parts. Crucial parts of the mentioned blockchain solution are proxy and reverse proxy for routing requests coming into the network, mediator server, which is responsible for taking original request, parsing its meta and body information and sending it to the blockchain ledger and blockchain node, which is deployed alongside the proxy and mediator to connect to the shared private blockchain network.

In this paper, private blockchain network is going to be contained on the same network as proxy, mediator and test original server for ease of testing and initial setup procedure. In the future this implementation will be extended to allow blockchain nodes, being setup via docker-compose, connect to an external network.

NGINX web-server is used as a proxy and reverse proxy in the solution subnetwork. Django framework is used to build a basic mediator server able to receive requests from the proxy, parse their metadata, save them to the blockchain and then pass the request to the original server. Private blockchain network is created using Ethereum geth tool with a PoW (Proof of Work) consensus. Thus, regular node, bootnode and mining node are contained within Docker subnetwork for testing purposes [2].

Client server architecture is expected to be able to integrate this solution into existing business process. But neither client nor server implementation have to be important for this architecture to be integrated, so regular API client is used as a client in this implementation and a

Research Article: This article was published by the publishing house of ***NTU "KhPI"*** in the collection "Bulletin of the National Technical University "KhPI" Series: System analysis, management and information technologies." This article is distributed under a Creative Common Creative Common Attribution (CC BY 4.0). **Conflict of Interest:** The author/s declared no conflict of interest.

simple Django web-server with Postgres database. Example web-server doesn't represent any real business activity and is provided for demonstration purposes as a proof of concept.

Docker Environment Description. To start with the implementation of the system, high level description of the fundamental tools has to be provided. The most important tool in this implementation is Docker. Docker is an open-source platform that has simplified process of how applications are developed, shipped, and run. At its core, Docker uses OS-level virtualization to package software into standardized units called containers [3].

In the Docker network every container is emulated as a separate virtual machine allowing them to run as different client and server machines. In our case, containers will be NGINX reverse proxy, Django mediator server and Blockchain network, which is being represented as several different containers with nodes tailored to different purposes. It's the implementation needed at this stage. Blockchain nodes are going to be connected so a bridge subnet work to differentiate them from the main network.

On the Docker application network, we will also have container for original server and container postgresql with a database for that server. Here and further on "original server" refers to a type of server which is not a part of dockerised blockchain solution architecture, but presumably a part of the client–server system that business planning on integrating blockchain solution already owns. In this paper for the sake of demonstration this "original server" will be a simple todo list backend implementation, as specifics of this type of server is not important for the dockerised blockchain implementation. It is only required from it to have a API interface and means of providing necessary credentials for Mediator server to be able to connect to it.

Each container is first being built as an image in docker infrastructure. A Docker Image is a lightweight, standalone, executable package that includes everything needed to run a piece of software. It can be considered a blueprint, template, or a snapshot of an application and its entire environment at a specific point in time. It's the "buildtime" artifact in the Docker ecosystem and it's created before the container.

A Docker container is the running instance of a Docker image. If a Docker image is the blueprint, a Docker container is the actual virtual machine built from that blueprint, where your application is executing. The key difference of container from image is that after container is built from image, it obtains several writing layers that allow to mutate data inside that container. Meanwhile in docker images data mutation is prohibited and cannot be done [4].

The drawback of containers is that when containers are removed, they lose the data the store in their data storage, as they store everything in the runtime memory (RAM). Solution for that is volume, a preferred mechanism for persisting data generated by and used by Docker containers. It provides a way to store data outside the container's writable layer, ensuring that the data remains intact even if the container is stopped, removed, or recreated. But in this research paper volume storage is not used, it is going to be implemented as a later improvement.

Dockerfile configuration files are used for defining each image scheme, which container is built from later on. A Dockerfile typically consists of several instructions, each on a new line. The order of these instructions is crucial, as each instruction creates a new layer in the final Docker image. The most important commands crucial for our implementation are those below.

FROM: specifies the base image for the container. It defines the environment this image is going to be run in, as it sets the image for each subsequent instruction. There are different variations of the images and light weight ones have to be a priority, to reduce container loading times and size taken.

WORKDIR: Sets the current working directory for any subsequent RUN, CMD, ADD, or COPY instructions.

COPY / ADD: Copies new files or directories from <src> (host path) and adds them to the filesystem of the image at the path <dest> (image path). COPY is generally preferred over ADD because it's more transparent and less prone to unexpected behavior.

RUN: Executes Commands during Image Build. Executes any commands in a new layer on top of the current image and commits the results.

EXPOSE: Command that is used for providing information about the ports, that can be used to send requests to the container based off this image. This command does not do any actual networking, but is useful for maintainability of the Docker environment.

ENV: Sets environment variables that will be available inside the container at runtime. This command is used excessively in our test implementation of the dockerised blockchain implementation to configure communication between containers in the sub-network. It is not as useful for configuring external variables, as we don't want to alter Dockerfiles directly after they are already established.

CMD: Specifies the Default Command to Execute when a Container Starts. It is crucial command to build the image and must not be omitted [5]. requests to a necessary service in the docker network or outside of it, to the existing applications.

Dockerfiles are written to build planned images, and basic structure of the docker looks can be seen on Fig. 1.

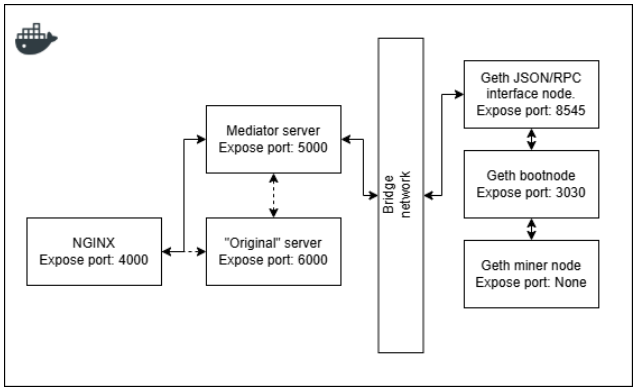

Fig. 1. Docker Container Structure

Basic docker container structure contains separate containers for NGINX reverse proxy, "Original" server and Mediator server. They are not connected into subnetwork

and "original" server is meant to be reached outside of the docker network, but for this test purposes it's located in the same docker network.

Bridge network connects Geth bootnode, geth miner node and Geth JSON/RPC interface node into the same sub-network inside docker. It is done to simplify access to Geth JSON/RPC interface outside of the bridge network and prevent direct access to the geth bootnode and geth miner node, which are not required for the implementation to function and could allow harmful actions, as bootnode is responsible for connecting new nodes to the blockchain network.

It is important to select optimal base images for all the containers mentioned as we want to reduce the time to setup and size of the memory, that is going to be allocated to running those containers.

For NGINX reverse proxy existing implementation is used, which is called nginx-proxy. It provides an extensive tool to work with rerouting requests from outside to internal network and vice versa. Using this service allows to omit implementing generation of NGINX configs from ground up by using docker-gen file generator [6]. Docker-gen is a powerful utility that generates files based on Docker container metadata and Go text/template language. Essentially, it acts as a dynamic configuration tool that watches for Docker events (like containers starting or stopping) and then automatically updates configuration files, scripts, or other artifacts [7].

Table 1 describes all the images needed to create respective containers. Selected images are described in more detail in the information below.

nginx-proxy requires these base images to build from: docker-gen:0.15.0, forego:0.18.3, nginx:1.29.0-alpine. So total size of the proxy container will be sum of those images' sizes, but it is a necessary compromise to be able to utilize proper reverse proxy implementation to orchestrate requests between networks and containers.

Mediator server in its foundation requires python:3.13 image and we are going to use python:3.13-slim specifically to reduce image size. Total container size is going to also include all the libraries required to run Django server and server itself.

Test version of "original" server is going to be similar to that of mediator, as they share Django framework as a base for web server implementation. In both of them default SQLite database is used to simplify setup process of the env and WSGI (Web Server Gateway Interface) is used to run Django servers.

For all blockchain nodes the same ethereum/client-go:v1.10.1 is used, which allows to setup and run Ethereum nodes in private blockchain network. It is important to have that exact version of the base image, as newer version don't support PoW (Proof of Work) consensus protocol, which is easier to setup as a private blockchain network locally. It is an area for future improvement to replace PoW consensus mechanism with PoS (Proof of Stake), which is used by latest Geth library version and corresponds with the current consensus mechanism used on mainnet – global Ethereum blockchain network [8].

Full information about the base images used for this demonstration implementation can be found on Docker hub website, which provides hosting for public Docker containers [9].

This is the configuration needed for building our demonstrative solution. First step is to set up basic docker network and deploy reverse proxy in it, mediator server and "original" server. Mediator server is not sending information to blockchain ledger until ethereum private network is deployed, which is described later in the paper.

Docker Proxy and Basic Server Setup. Dockerized blockchain implementation is expected to operate on a single server machine to lower maintenance cost and simplify setup process. But it is desired to leave possibility to send requests not only to mediator server but also to original one, for the term of setting up new environment and for possible emergencies which might occur on the early stages of distributed solution deployment.

For that purpose, we are setting up nginx-proxy container first, so that it acts as a reverse proxy for our network of mediator server, original server and blockchain private network, that is going to be examined later in this paper. Idea is to be able to send the same requests, which are already being sent from the client, but to a different host, process the metadata and then redirect request to the original server and return its response back.

Table 1 – Selected Images Characteristics

Image	Description	Container	Size	Last Updated
python:3.13-slim	Python is an interpreted, interactive, object-oriented, open-source programming language.	Mediator server, "Original" server	43.43 MB	Jul 24, 2025
nginx:1.29-alpine	Official build of Nginx	NGINX Proxy	20.57 MB	Jul 18, 2025
jwilder/docker-gen:0.15.0	File generator that renders templates using docker container metadata	NGINX Proxy	11.42 MB	Jul 23, 2025
nginx-proxy/forego: 0.18.3	Foreman in Go	NGINX Proxy	6.62 MB	May 8, 2025
ethereum/client-go:v1.10.1	Official golang implementation of the Ethereum protocol.	Ethereum Bootnode, Ethereum JSON/RPC node	20.58 MB	Mar 8, 2021

Imagine that client.com is a client's host and original-server.com is a host of original server. Before implementing this blockchain solution, they would simply send messages to each other: client.com, original-server.com. Dockerized solution introduces mediator-server.com host for a mediator server inside docker network. So new chain of requests looks like this: client.com to mediator-server.com to original-server.com.

Reverse proxy requires DNS of the host machine to be configured in a way, so that hosts original-server.com and mediator-server.com are routed to our localhost:80 [6]. 80 is a port to which we expose our nginx-proxy in the Dockerfile. In our case local machine is used as a server machine for the Docker network, so Windows hosts file has to be adjusted with added hosts mappings. File can usually be found here: C:\Windows\System32\drivers\etc. Added mapping might look like this: 127.0.0.1 original-server.com, 127.0.0.1 mediator-server.com.

We could also specify hosts for the links we are setting as redirects to localhost, but for ease of setup in our case we are going to redirect links without any port specified.

We have several ways to deploy nginx-proxy container in our network. First one is to download source code from public nginx-proxy repository and build container from the Dockerfile, it has two different files for Debian and Alpine environments. By default, Debian one is used and we don't have specific requirements to change that approach, so we are going to proceed with Debian dockerfile too.

We can specify Dockerfile to build in the docker-compose file under the container section that we want to build, in this case it's nginx-proxy. It is done to remove building Docker image step from the setup chain, as we could first created image with docker build command and then spin up container with docker run. docker-compose.yaml file is a config for creating containers where we specify either path to the local Dockerfile build file or specify remote image, that is already hosted on Docker Hub. To reduce complexity and code overhead remote image from Docker Hub is used, as it already includes all the images required for container to work and we don't have to attach and build them manually [5].

nginx-proxy implementation includes default docker-compose.yaml file, which is used for running this tool via command line interface, as specified in the Readme instruction in the nginx-proxy repository. To include this container as part of our network we just have to include nginx-proxy container section in dockerised blockchain solution docker-compose.yaml file, but omit whoami container from it, as it is a container created for testing purposes required to ping the proxy functionality if no other servers configured. In our case we will have original server and mediator servers in the network, so we don't need whoami container.

Servers are created as Django containers. As first step we have to create local virtual environment to be able to setup servers via Django CLI, but after initial setup is complete, instructions will be provided in Dockerfile for each server how to setup corresponding environment in the docker container. To create virtual environment python venv command is used and, depending on the operating system, different scripts inside that venv are used in CLI to activate it. Main dependency for newly created virtual environment is Django. We install it with pip install command. Installation of Django will also include dependencies it needs to operate. After that we have to create requirements.txt file to be able to let Docker environment know, what dependencies have to be installed inside container environment. It is done via pip freeze > requirements.txt command and not by hand, to include internal dependencies and simplify the process. Django and its internal dependencies are enough for original-server, but we also have to include requests library in the mediator-server virtual environment to enable main functionality mediator-server is created for.

It is going to receive requests, process the request body and metadata, save it to the blockchain ledger, and then send this request further to the original-server. As a response it returns response from the original-server. It is basically acting as a middleware between client side, which is Echo VS Code plugin in our case (simple API client), and original server. As the first step, mediator server will wait for result of writing data to the blockchain network. Making it work on eventbased system is an improvement that is planned for future work. Requests library is needed to be able to send requests to the external hosts, which original-server will act as, based on our configuration.

Due to the limits and demonstrative nature of the solution described in this paper, original-server is included in the docker network, so it is factually a server located on the same network as mediator. But nginx-proxy is set up to treat it as external host, so that when solution is adjusted for the real infrastructure, implementation won't be deprecated and not applicable or at least require minimal intervention. Ideally most the configuration should be done via environmental file, except security related things like secret keys.

Both servers' projects contain one Dockerfile each, which describes how image has to be built. By default, EXPOSE command in the dockerfile is declarative but does not serve any function. But as we use nginx-proxy reverse proxy, we have to specify ports with EXPOSE command, as nginx container uses that metadata to dynamically generate IP addressed of the servers inside the Docker network. There is an alternative way to let nginx-proxy know to which ports different hosts are rerouted, which is specifying ports under server container section in the docker-compose.yaml file. But to avoid confusion and possible errors, exposed port for each server is specified in both Dockerfile for each server and shared docker-compose.yaml of the entire project.

Next requirement for the proxy to work is to specify VIRTUAL_HOST environmental variables for each container in the docker-compose.yaml file. In our case, mediator server has VIRTUAL_HOST variable set to mediator-server.com and original server – has env variable set to original-server.com. Final requirement for the Docker network to work is to configure DNS to map our proxy hosts to the localhost, so that when request from Docker host machine is made to those hosts, they are redirected to nginx proxy address [7].

For testing purposes, original server is a simple todo application backend, which allows to add and get all the todo entries. SQLite database is used to avoid setting up separate Postgres container, but it is marked as an area for future improvement. Dockerised blockchain solution also shouldn't care about details of original server implementtation, we just need its RESTful API interface to operate with. Otherwise, it would break the interoperability of the proposed solution for different original servers, which are already present on customer side.

Mediator server on this stage has endpoint implemented, that catches all the incoming requests to that server and then redirects them to original server. Original server's host is currently hardcoded, but eventually it will be managed from environment configuration, as one of the possible solutions.

To launch whole project, docker-compose up command is used. To test implemented functionality, we send same /api/todo GET request to both hosts. So sent requests endpoints are: original-server.com/api/todo and mediator-server.com/api/todo. Responses to the client for these URLs have to be identical. Result can be seen on Fig. 2.

Docker Blockchain Network Setup. Next step is to introduce private blockchain network inside the existing docker solution network. For that another Docker image is used – ethereum-client/go. It will allow us to deploy ethereum nodes with necessary interfaces to communicate with them.

On this stage of demonstrative solution implementation Proof of Work consensus mechanism is used when setting up private blockchain network. In this consensus mechanism other nodes in the network have to perform cryptographic calculation to prove the new encrypted records in the ledger.

Proof of Work is not supported by the latest version of ethereum-client/go, so we have to use specific version, which supports PoW. In our case it is 1.10.1. Version with this consensus mechanism is selected because of the official documentation for latest version of the ethereum client providing instructions on how to deploy separate blockchain network with a tool different from Docker. Although that new tool is based on Docker and allows for simpler process of the network deployment, it doesn't provide effective ways to possibly separate process of creation of new nodes in different Docker networks, connecting to an overarching private blockchain network [8].

Proposed solution relies on being able to deploy blockchain nodes separately from the main private network but also purely in the scope of each separate Docker network belonging to each separate business owner of the node. Due to this reason and better documented process of setting up Proof of Work nodes in an autonomous Docker environment, older version and consensus mechanisms are chosen.

But it is clear that Proof of Authority consensus mechanism is able to improve blockchain network performance, as it won't be requiring cryptographic approvement calculation for each new record [10]. This would decrease latency time for getting responses to requests from the original client. So, moving to the PoA mechanism is considered a further improvement for this solution, which will be considered in the scope of a different research.

For setting up Ethereum nodes in the docker network we don't have to have any source code for them, but we need separate Dockerfiles for each type of the nodes required for the network.

To run private network for our solution these types of nodes are required:

- Bootnode, which acts as an entry point and a gateway to the private ethereum network. Its' port has to be exposed, so that we are able to connect via it, when deploying several other instances of our solution for different participants of the supply chain, that are integrating blockchain solution [11];
- JSON-RPC node, which is providing an interface for sending information to blockchain ledger via HTTP 1.1/2 request, in our case – through requests python library in Django framework. Each participant of the supply chain has separate node for each of them acting as both node allowing to connect to the blockchain network and a separate interface for API communication, not to overlap with other participants' nodes;
- Miner node. This node is required for a Proof of Work consensus mechanism, confirming writing operations in the ledger. It might be theoretically combined with JSON-RPC node, so that one node has two of those characteristics, separate testing will be conducted for that matter [12].

Starting state of the blockchain network is described in the genesis.json file, which is located in the ethereum-network folder, which is allocated for source files used to set up distributed network. So basic accounts and funds allocations are specified in genesis.json file. In the scope of current research, creating new network is implemented as a part of running this solution via docker-compose, but it is planned to improve it later on with separating the private blockchain network and making it external, so that propose dockerised solution only deploys one blockchain node to connect to the existing network to reduce load on the server and maintenance costs. Also, each bootnode has to generate a enode link, which has to be available to other members of

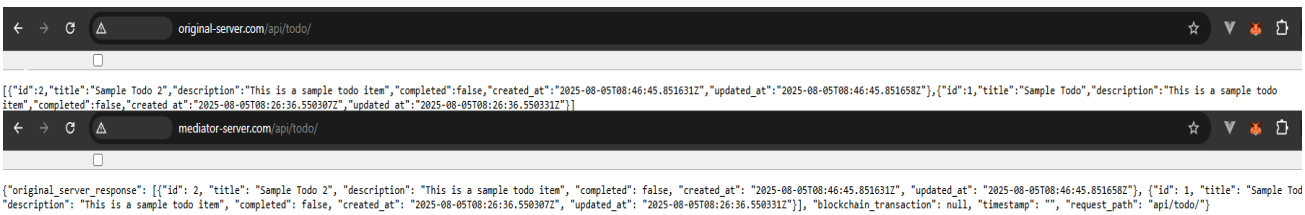

Fig. 2. Docker Running Proxy and Two Servers

the private blockchain network and those, who is going to join it. Having multiple enode is complicating the process and creating more risks of those links being exposed, allowing malicious actors to access private blockchain network and manipulate the data. Thus, it might be better to have one enode as a single source of truth and treat that enode as a secret link. Additional security mechanisms might be implemented to secure that enode link, such as cyphering and deciphering it with a secret/public keys, that each member of a supply chain has. But this security issue requires more investigation and is not covered in this paper. This is also considered as a future improvement.

Resulting images and containers running can be seen in Docker Desktop window on Fig. 3. Captured logs of demonstrative solution running can be seen on Fig. 4.

Interface configurations are specified in the general docker-compose.yaml file, where ports for bootnode and JSON-RPC interface node are specified. For containers with those nodes VIRTUAL_HOST also has to be specified to allow nginx-proxy container to send requests from mediator server and outside blockchain nodes to those.

Additionally, we can temporarily group all the blockchain nodes into a sub-network inside our Docker network to better organize them in our solution. It can be done by specifying new sub-network in the docker-compose.yaml file.

It will create necessary images first and then run new containers based of them. After necessary configuration is done, we can run our solution via docker-compose up command.

To check if servers and blockchain containers are running correctly we first look at the logs, which are being printed out in the terminal, where docker-compose up command is executed. Absence of errors there is the first signal of our solution working correctly on that stage.

Next step to check if blockchain is running correctly is to try getting network information, such as accounts, via API client, such as Echo API used in this project.

For that we are sending a POST request first to check connectivity of nodes. To see the list of peers available, we send request with raw JSON data and Content-Type header equal to “application/json”. Raw data has to look like this: {"jsonrpc": "2.0", "id": 1, "method": "admin_peers", "params": []}. As a result, we get a JSON response, body of which contains information enodes available for use.

localhost:8545 is used as an API URL, as we accessing it outside of docker container for now.

This response is a JSON-RPC reply from a Geth (Go Ethereum) node, specifically the result of calling something like admin_peers. It lists information about one connected peer on the Ethereum network. This JSON shows that your Geth node is connected to one peer at 172.16.254.2:30303, running Geth v1.10.1, supporting Ethereum protocols (eth/64–66, snap/1), and currently on a blockchain head with difficulty. It signals, that it is accessible both outside the docker environment, as we want it to be, and inside the docker network. Host will be geth-rpc-endpoint in that case as we are reaching out through RPC interface node.

Example result of the communication via RPC interface is displayed on Fig. 5.

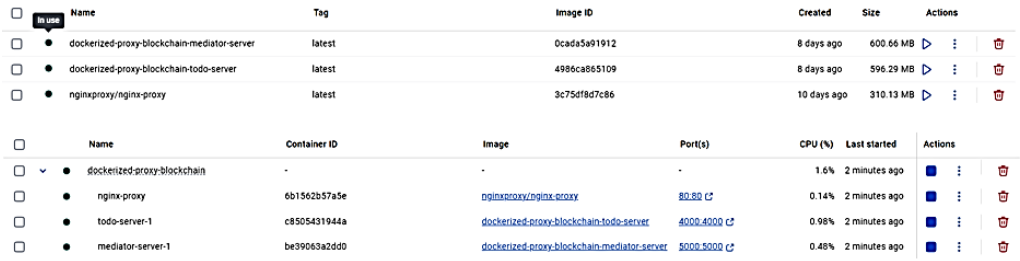

Fig. 3. Built Images and Running Containers

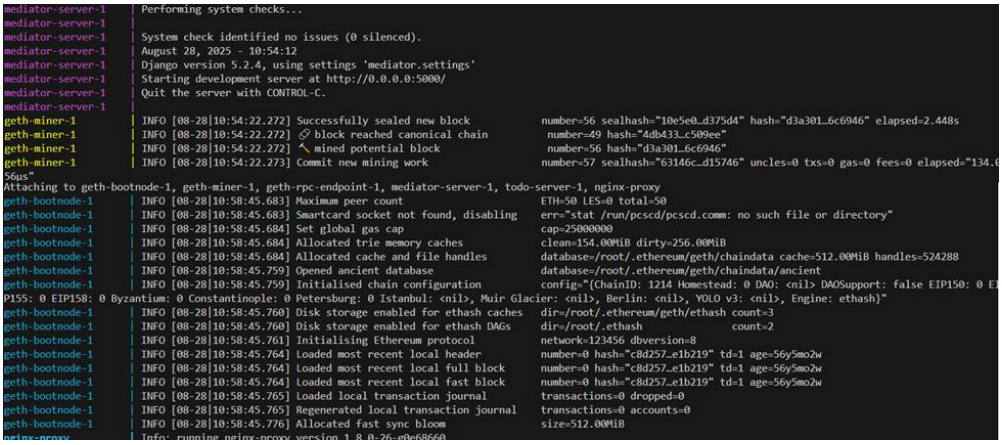

Fig. 4. Docker Solution Logs

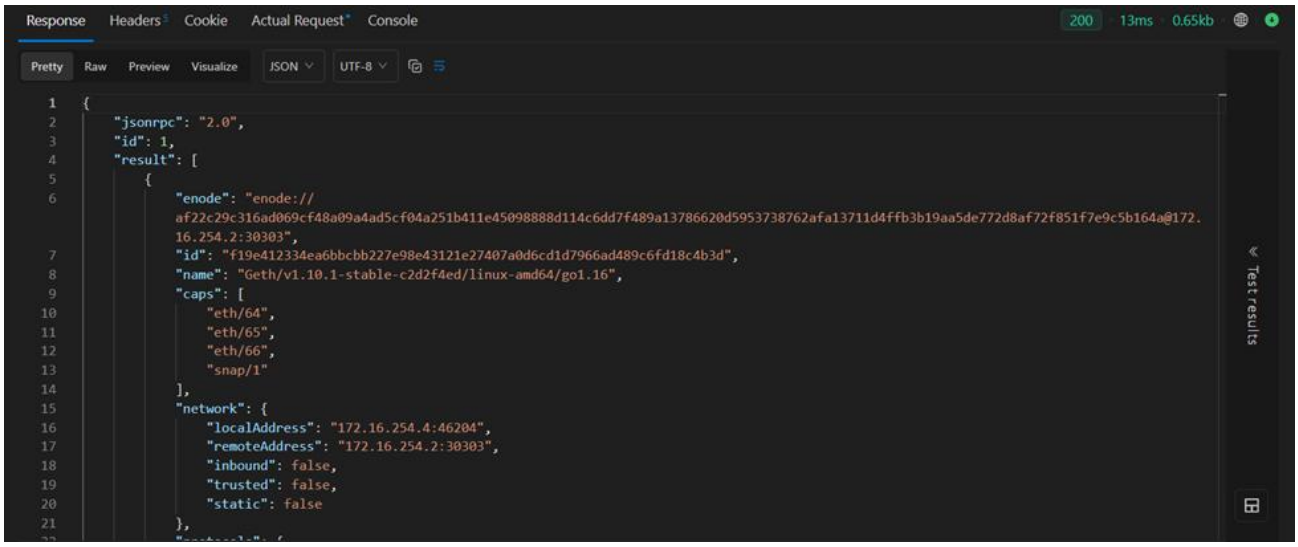

Fig. 5. Peer Connectivity Result

Area for Improvement. Showcased demonstrative proof of concept of the dockerized blockchain solution presents main idea of the proposed architecture and proves its implementation being possible, but it's still missing several key features, that are intended to be in the final solution. First, we deploy and access private blockchain network alongside deploying original and mediator servers. It's done for demonstration purposes, but goal is to make it possible to connect mediator server to the server outside of the docker network. It's not a difficult implementation, but experimental part of the implementation for that aspect will be done in a context of next research papers.

The more complicated topic is deploying, maintaining and connecting to the private blockchain network, that is shared between mediator servers' instances. Idea is to setup and run only RPC miner node alongside the mediator, which is able to connect to an outside private blockchain network. Ideal option is to have it check, if the mentioned network is available and connect to it only if it's present. Otherwise, system could set up missing network and provide necessary credentials for other systems to connect to it. But plausibility of such implementation is to be estimated, so this work is also planned for the further research.

Third aspect for improvement is implementation of custom smart contracts in Solidity language for customizing the way requests data is being saved and handled. It is yet to be researched, how those smart contracts can be shipped together with the solution, either as a part of initial setup or a separate script in mediator server, that will trigger deploying smart contracts from the solution to the private blockchain network.

Conclusions. The research presented in this paper demonstrates the feasibility of deploying a dockerised blockchain network as a modular extension to existing client–server architectures. By leveraging Docker's containerization, networking, and orchestration capabilities, it becomes possible to encapsulate blockchain nodes, mediator servers, and reverse proxies into an environment that is both reproducible and easily maintainable. The proof-of-concept shows how a mediator server, implemented in Django and connected to an Ethereum private network via JSON-RPC, can intercept client requests, record them on a blockchain ledger, and then transparently forward them to the original server. This validates the core idea that blockchain technologies can be integrated into existing infrastructures without fundamentally altering their design, instead introducing an intermediate layer that ensures data immutability and trust.

At the same time, the work highlights limitations and directions for further development. The current architecture relies on Proof of Work nodes within a Docker subnetwork, which simplifies experimentation but introduces scalability and performance constraints. Future iterations of the system could migrate to more efficient consensus mechanisms, such as Proof of Authority or Proof of Stake, to reduce latency and computational overhead. Similarly, persisting data using Docker volumes, refining external network connectivity, and introducing Solidity-based smart contracts would strengthen the robustness, adaptability, and business applicability of the solution.

Overall, the project delivers a working demonstration of how distributed ledger technologies can be containerized, orchestrated, and embedded into client–server ecosystems in a way that lowers deployment complexity while paving the path for more advanced features. It not only validates the technical foundation of such integration but also establishes a framework upon which future research and enterprise-grade blockchain applications can be built.

References

1. Жержерунов П. Ю., Шматко О. В. Designing the architecture and software components of the dockerised blockchain mediator. *Системний аналіз, управління та інформаційні технології*. 2025. №. 1 (13), P. 101–105. DOI: https://doi.org/10.20998/2079-0023.2025.01.15.
2. Gervais A., Karame G., Wust K., Glykantiz V., Ritzdorf H., Capkun S. On the Security and Performance of Proof of Work Blockchains. *CCS '16: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016. P. 3–16. DOI: https://doi.org/10.1145/2976749.2978341.

3. *Docker*. URL: https://www.docker.com/what-docker (дата звернення: 21.08.2025).
4. Rad B. B., Bhatti H. J., Ahmadi M. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*. 2017. №. 17 (3). P. 228–235. URL: https://www.researchgate.net/publication/318816158_An_Introduction_to_Docker_and_Analysis_of_its_Performance (дата звернення: 21.08.2025).
5. *Dockerfile reference. Docker Docs*. URL: https://docs.docker.com/reference/dockerfile/ (дата звернення: 21.08.2025).
6. *nginx-proxy*. URL: https://github.com/nginx-proxy/nginx-proxy (дата звернення: 21.08.2025).
7. *docker-gen.* URL: *https://github.com/nginx-proxy/docker-gen* (дата звернення: 21.08.2025).
8. *Kurtosis. Go Ethereum*. URL: https://ethereum.org/docs/fundamentals/kurtosis/ (дата звернення: 21.08.2025).
9. *Docker Hub container image library: App containerization*. URL: https://hub.docker.com/ (дата звернення: 21.08.2025).
10. Joshi S. Feasibility of proof of authority as a consensus protocol model. *arXiv*. 2021. URL: https://arxiv.org/abs/2109.02480 (дата звернення: 21.08.2025)
11. Luo J. Unveiling Ethereum's P2P network: The role of chain and client diversity. *arXiv*. 2025. URL: https://arxiv.org/abs/2501.16236 (дата звернення: 21.08.2025)
12. *Command-line options. Go Ethereum*. URL: https://geth.ethereum.org/docs/fundamentals/command-line-options (дата звернення: 21.08.2025).

References (transliterated)

1. Zherzherunov P. Y., Shmatko O. V. Designing the architecture and software components of the dockerised blockchain mediator. *Systemnyi analiz, upravlinnia ta informatsiini tekhnolohii*. 2025, no. 1 (13), pp. 101–105. DOI: https://doi.org/10.20998/2079-0023.2025.01.15.
2. Gervais A., Karame G., Wust K., Glykantiz V., Ritzdorf H., Capkun S. On the Security and Performance of Proof of Work Blockchains. *CCS '16: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 3–16. DOI: https://doi.org/10.1145/2976749.2978341.
3. *Docker*. Available at: https://www.docker.com/what-docker (access date: 21.08.2025).
4. Rad B. B., Bhatti H. J., Ahmadi M. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*. 2017, no. 17 (3), pp. 228–235. Available at: https://www.researchgate.net/publication/318816158_An_Introduction_to_Docker_and_Analysis_of_its_Performance (access date: 21.08.2025).
5. *Dockerfile reference. Docker Docs*. Available at: https://docs.docker.com/reference/dockerfile/ (access date: 21.08.2025).
6. *nginx-proxy*. Available at: https://github.com/nginx-proxy/nginx-proxy (access date: 21.08.2025).
7. *docker-gen.* Available at: *https://github.com/nginx-proxy/docker-gen* (access date: 21.08.2025).
8. *Kurtosis. Go Ethereum*. Available at: https://ethereum.org/docs/fundamentals/kurtosis/ (access date: 21.08.2025).
9. *Docker Hub container image library: App containerization*. Available at: https://hub.docker.com/ (access date: 21.08.2025).
10. Joshi S. Feasibility of proof of authority as a consensus protocol model. *arXiv*. 2021. Available at: https://arxiv.org/abs/2109.02480 (access date: 21.08.2025)
11. Luo J. Unveiling Ethereum's P2P network: The role of chain and client diversity. *arXiv*. 2025. Available at: https://arxiv.org/abs/2501.16236 (access date: 21.08.2025)
12. *Command-line options. Go Ethereum*. Available at: https://geth.ethereum.org/docs/fundamentals/command-line-options (access date: 21.08.2025).

Received 19.11.2025

УДК 004.72

П. Ю. ЖЕРЖЕРУНОВ, студент, Національний технічний університет «Харківський політехнічний інститут», м. Харків, Україна, e-mail: Pavlo.Zherzherunov@cs.khpi.edu.ua, ORCID: https://orcid.org/0009-0005-7240-9395
О. В. ШМАТКО, доктор філософії (PhD), доцент, Національний технічний університет «Харківський політехнічний інститут», доцент кафедри програмної інженерії та інтелектуальних технологій управління, м. Харків, Україна, e-mail: oleksandr.shmatko@khpi.edu.ua, ORCID: https://orcid.org/0000-0002-2426-900X

АРХІТЕКТУРНИЙ ПІДХІД ДО ЗАХИСТУ ДАНИХ У РОЗПОДІЛЕНІЙ СИСТЕМІ УПРАВЛІННЯ ЛАНЦЮГОМ ПОСТАЧАННЯ З ВИКОРИСТАННЯМ БЛОКЧЕЙН-ВУЗЛІВ

Рішення на основі блокчейну, реалізоване за допомогою Docker, може покращити поточний низький рівень впровадження розподілених технологій у малих та середніх підприємствах. Це можна зробити шляхом проектування та впровадження середовища, яке успадковує простоту розгортання та масштабованість контейнерних систем із безпекою та прозорістю розподілених додатків. У цій статті описано практичне впровадження рішення на основі блокчейну, розробленого як демонстраційну реалізацію для існуючої клієнт–серверної архітектури. Це рішення використовує контейнери Docker для спрощення налаштування та розгортання приватної мережі блокчейну, сервер-посередника та зворотній проксі-сервер. Впровадження цієї системи в невеликому масштабі демонструє можливість інтеграції технології блокчейну в існуючі бізнес-процеси без фундаментальних архітектурних змін і підтверджує проблеми розгортання та обслуговування, які зазвичай супроводжують розподілені системи, що використовують приватний блокчейн. Обговорювана реалізація є демонстрацією того, що розроблена архітектура є потенційно відтворюваним і легко підтримуваним середовищем для реєстрації та перевірки даних за допомогою незмінного реєстру в меншому масштабі. Доказ концепції успішно підтверджує основну ідею. Реалізація показує, як сервер-посередник перехоплює запити клієнтів, записує їх у приватний блокчейн Ethereum через інтерфейс JSON-RPC, а потім пересилає їх на оригінальний сервер. Це підтверджує здатність рішення впровадити надійний проміжний рівень для незмінності даних. Проект демонструє робочу структуру для вбудовування технологій розподіленого реєстру в екосистеми клієнт–сервер. Хоча поточний механізм консенсусу Proof of Work має обмеження щодо масштабованості, архітектура забезпечує міцну основу для майбутніх досліджень, включаючи перехід на більш ефективні механізми консенсусу та інтеграцію смарт-контрактів.

Ключові слова: докеризована архітектура блокчейну, управління ланцюгами поставок, контейнеризовані вузли блокчейну, малі та середні підприємства, ланцюг поставок, захист даних у розподіленій системі, блокчейн проксі, алгоритми хешування, ethereum.

Повні імена авторів / Author's full names

Автор 1 / Author 1: Жержерунов Павло Юрійович / Zherzherunov Pavlo Yuriiovych
Автор 2 / Author 2: Шматко Олександр Віталійович / Shmatko Olexandr Vitaliiovych