Applied Computer Science, vol. 14, no. 1, pp. 27–41
doi:10.23743/acs-2018-03

Submitted: 2018-01-06
Revised: 2018-03-05
Accepted: 2018-03-25

blockchain, cloud, datacenter, cluster

Maciej NABOŻNY[*]

ASYNCHRONOUS INFORMATION DISTRIBUTION AND CLUSTER STATE SYNCHRONIZATION

Abstract

This article describes issues related to information distribution and cluster state synchronization in decentralized environments with inconsistent network topology. The main objective of this study was to create a set of rules and functional requirements to build a fault-tolerant framework based on Blockchain for creating applications in decentralized and distributed environments, regardless of the underlying cluster's hardware.

1. INTRODUCTION

Proposed principles could be used as base for many systems, including IT systems, cluster management software, IoT systems and so on. Thus, this paper will discuss only theoretical aspects of distributing information in safe, reliable way.

To define system distributing information across environment, one should define at first basic, necessary components of such system. First, basic element is Node. This should be an individual entity capable of making decisions and interacting with other nodes in the environment. The way these nodes communicate with each other is a Connection. One connection enables one-way communication between two nodes. To provide bidirectional communication between two Nodes, it is required to provide double Connections.

All the above components together are called Clusters. A cluster is a set of connected Nodes and Connections between them, that form a partial or full mesh. This grid can be described using an oriented graph, in which each graph node is able to reach any other node on the cluster. A divided cluster (also described as a split-brain configuration) is a situation in which at least one node is unable to reach at least one other node in the cluster.

[*] cloudover.io ltd, 590 Kingston Road SW20–8DN London, United Kingdom,
+48 511 912 775, maciej.nabozny@cloudover.io

Notification is full information about the change of state of one Node in a Cluster. Notifications are generated by a node that changes its state and broadcasted to all other nodes in the cluster. Nodes can also forward received from neighbors notifications to other nodes, to provide notification routing and thus, redundant communication channels.

By state of the Node we can define the internal state of the resources driven by the Node's logic. Information can be understood as a description of such a state known by local and remote Nodes. Thus, the information about the Node is strongly related to its state, but at some point it may be contradictory.

2. PROBLEM FORMULATION

To create a solution for the described problem of distributing information and synchronizing state of nodes in entire cluster, without a centralized point, it is necessary to build a system that solves the following problems:

– How to handle the "split-brain" of cluster?
– How to handle rejoin of cluster?
– How to authenticate notifications?
– How to limit access to resources?
– How to trust newly attached nodes or objects to cluster?
– How to handle joining two existing clusters?
– How to handle conflicting updates of data after rejoin?
– How to avoid any additional communication channels to use only notifications for communication?
– How to track history of database and all changes over time?

The implementation, which covers all of the above issues, will define a fully functional, event-driven platform for creating distributed systems on the verge of Blockchain, public key infrastructure, and agent technology.

3. SOLUTION

3.1. Information schema

In most distributed systems the database is the central point of application cluster (Fernstrom, Narfelt & Ohlsson, 1992). Proposed approach assumes to keep local copy of database at each node and store information locally. Such organization of information may cause that each node may have different version of information, according to the state of cluster, time and its location among other nodes. Thus, by data synchronization we will define striving for make data sets the same on all nodes. In some cases it will be necessary to accept information inconsistency in different parts of cluster.

To organize the information into logical structures, in programming languages we could define classes and store information as objects (Lippman, 1996). The same idea was proposed here, to group information into objects inside database. Within this article we could redefine an object in context of described system. The object will be a set of named properties related to one logical kind, assigned to the one node. To store one property we should store:

- entity type,
- entity identifier,
- field name,
- field value.

In opposition to classes, proposed solution does not guarantee the schema of information. Once defined entity could have different fields at different nodes, dependently of its version or state. Such approach makes possible to upgrade whole system and change its information organization.

With such approach we could use key-value data store as the underlying database and store information about all objects in database in form:

– Key:	entityType:entityIdentifier:fieldName
– Value:	the value of field

Such underlying data schema makes possible to synchronize data at field level and manage privileges, on object level. Moreover, with above design one can make inclusion of objects, without any additional requirements of schema. Each included object gets a parent-object's identifier as prefix of this object's keys.

3.2. Split-brain and re-join of cluster

The split brain problem (Davidson, Garcia-Molina & Skeen, 1985) covers situation, when communication between part of cluster or one node and rest of the cluster is temporarily interrupted. Such situation causes the data on particular nodes is not synchronized properly. This leads to situation where node or part of cluster is not up to date with its data, to the rest of cluster. Split brain situations are hard to handle in systems, especially in case when logics running in cluster has to take a decision based on its data.

Described in this chapter split brain is especially dedicated for handling by systems based on blockchain like described here instead of standard architectures and databases, where split brain and rejoin should result the consistent database. Also we won't discuss here the consensus algorithms used by blockchain, which were removed in proposed solution, what will be discussed later.

Most often, split-brain configuration could be caused by non-redundant hardware connecting parts of cluster:

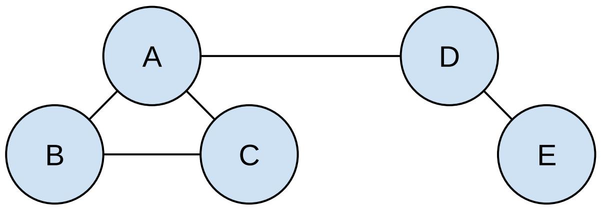

Fig. 1. A cluster with not fully redundant connections

It is possible to indicate four weak points shown in cluster described in Fig. 1. (regardless of the directionality of connections):

– A to B connection,
– D to E connection,
– A notification exchange (routing) from D,E nodes to B,C nodes,
– D notification exchange (routing) from E to A,B,C nodes.

Failure of any point from above list will result split-brain configuration. For example, failure of node A, shown in fig. 2 will split whole cluster to two, sep-arate clusters shown in fig. 3.

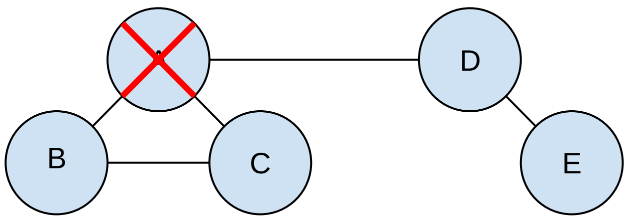

Fig. 2. Failure of node A

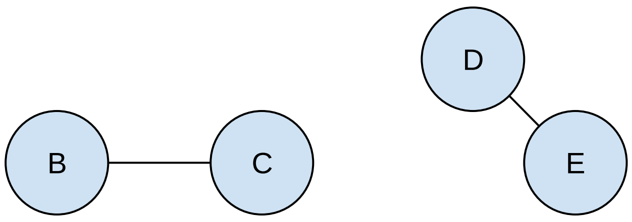

Fig. 3. Cluster topology with outage of node A

Let’s assume, that all information related to the cluster state will be available on all nodes till the split-brain, through local databases. In this way, notification exchange between nodes is running till outage of Node A. Moreover, each node after this failure will keep full copy of all information collected till split-brain.

Ongoing lifecycle of cluster could influe to the information stored on particular nodes and generate new notifications related to changes in node’s state. For example, changing the state of node B will create a change in local information.

This will create new notification. In cluster configuration shown in fig. 3, such notification will be shared by node B only to node C. No other possible route for updates is not available here.

Approach to store information locally, not in centralized database for cluster will prevent disconnecting nodes from data source in case of cluster connectivity failure. This could decrease performance of synchronization and propagation at node level, and decrease capacity of overall cluster.. Also duplication of data across all nodes could increase the costs of storing data and hardware in comparison to centralized system.

Handling information by centralized database results that any nodes temporarily detached from cluster make inconsistency quickly. This raises following questions:

– how should the online part of cluster react on changes on re-connected nodes,
– how should re-connected nodes react on changes in online part of cluster,
– how to handle conflicting updates of common parts (i.e. one of online node's changes state of disconnected node during split).

Proposed architecture is oriented to be data driven and to handle any changes of information by local Node's logics. It means, that changes in in information (broadcasted notifications) trigger logics. This is in opposite to: API > logics > database model, which is known from common architectures, where exposed API endpoint triggers execution of logics and then, logics updates central database. Event/Data driven architecture focused on reacting on data updates simplifies handling the rejoin. At the beginning of cluster's lifetime one of nodes should create the first notification about change in database. Each ongoing notification will be marked as successor of the previous one, by marking this fact on one of notification's internal fields. Thus, when all nodes are online, notifications are spreaded almost immediately. Such approach makes the chain of notifications, ordered by internal notification's fields. This idea was taken from the Blockchain (Nakamoto, 2008) database. The difference is that in some cases, chain could become a tree of notifications (or coexisting chains), when two or more nodes are concurrently changing database in the same time. The simple chain of notifications is shown in fig. 4. All notifications are ordered one, after another, by pointer to previous notification.

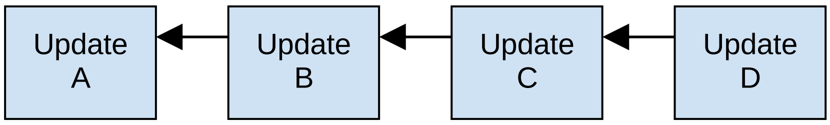

Fig. 4. Chain of notifications

In case of outage of node(s), new updates are not present in its chain. After rejoin, such node will receive only last new update.

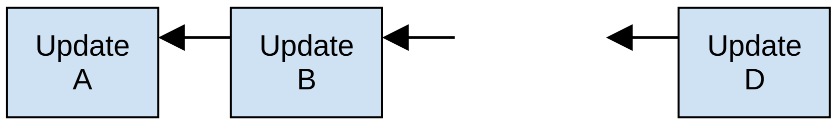

Fig. 5. Incomplete chain of updates, known by node disconnected after receiving update B

Shown in fig. 5 sample chain, represents notifications known by disconnected and then reconnected node. Latest received before outage notification, is notification B. After rejoin, this node will receive ongoing notification D, which points to notification C as its predecessor. Such notification is not known for rejoined node, so it should broadcast request for retransmission of notification C to all its neighbourship. In this way, rejoined node could track all changes made during its outage. Also, in case of new notifications of changes made during outage, this node could retransmit this changes to the rest of cluster. Proposed approach makes greater overhead in comparison to the central database, but makes easy to handle rejoin of any nodes and track historical state of database.

3.3. Handling join of new nodes and joining clusters

Described above way of handling rejoin of particular nodes after split brains applies to the joining new nodes and joining two or more clusters together too. Once new notification appears in node, or new cluster, all previous notifications should be fetched from older part of cluster as the notification's predecessors. There is no additional notification of join for existing cluster, thus all existing nodes could know notifications from new node or newly attached cluster immediately, after first change of information

3.4. Notification broadcasting

To provide better redundancy and failure tolerant cluster of nodes, it is recommended to use physical connections in redundant configuration. Additionally, algorithm responsible to broadcasting new notifications should use all possible connections to broadcast notification. Such approach makes cluster highly available and resistant for failure of particular connections between nodes. This approach could be optimized for better performance, but the it could guarantee lower availability level. Such approach is acceptable, until we need to handle connection failures as the primary functionality.

3.5. Handling conflicting notifications

Due to proposed architecture specification, many nodes could modify simultaneously one information in cluster. Most popular approach suggests to use locks and mutexes (Courtois, Heymans & Parnas, 1971) to avoid concurrency. Proposed solution shifts responsibility of handling such conflicts from database to the logics and notification ordering. Once order of notifications is known, there is no need to deal with concurrent modifications. Each node will process this notification, which is first in chain and modify information respectively to this order.

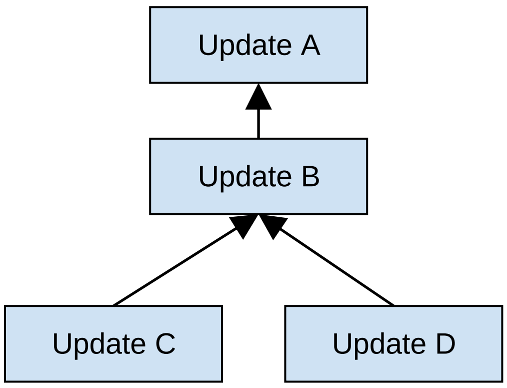

Fig. 6. Two conflicting updates, C and D

In situation of simultaneous notifications, with the same predecessor (fig. 6), logics on each node receives information about conflict. It is certain, that one of two notifications will arrive to node faster, so another will be marked as conflicting. The decision what to do should remain only to the logics at particular node, however simultaneous modification of one resource (i.e. setting value of single variable) is only fraction of such notifications. Most of split brains should produce branches in notification chain, without conflicts. Thus, the information schema in cluster should be designed in way to minimize risk of concurrent modification of the same resources.

The only problematic scenario is when two nodes modify the same resource in database. It is not possible to make decision in advance, so it is necessary to leave this decision to the logics.

3.6. Security of cluster and its data

In order to ensure a fully functional system design, it is necessary to solve the security problems. The proposed system will be an open database, like Blockchain (Nakamoto, 2008), with cryptography-based security. Thus, the permission to read any part of the information will be granted to each node connected to the cluster.

To secure write access to objects, one should use the asymmetric keys to provide signature-based authentication of each notification. Thus, to each object in cluster should be associated with its public and private key. Private key should be known only for the node, which creates object. Public key should be shared with all nodes in cluster with first notification, related to this object creation. This will allow each node in cluster to verify all ongoing notifications related to certain object. By storing list of authorized public keys related to object, we could also grant and revoke permissions to modify this object. Authorized objects changing information related to another object shall sign such notification with own private key, authorized in modified object.

The special case are object's fields which should be available for everyone. Then, the logics should accept notifications without matching signature or to recreate own, signed notification with the same information. Such approach could be used to allocate resources and leave verification to resource handler in cluster. If node's logics could accept such resource allocation, it could re-sign such notification with own, authorized key and re-share with cluster. Otherwise, notification won't be re-signed and it will be ignored by all other participants of cluster.

Once each object has public-private key pair assigned, it is possible to secure the access to object's data. It could be done by encrypting particular fields of object with provided public keys. By using mechanisms known from various encryption systems (i.e. dmcrypt ("DM-Crypt project", 2018) in Linux or GnuGPG ("GNU Privacy Guard project", 2018)), we could grant access to data for multiple nodes in cluster in the same way.

Above two assumptions make access for reading and writing data into the database complete. We could define how to grant access to modifying objects, share data in secure way and handle unsigned notifications.

To prevent spoofing new objects we could make object's ID related to its public key. The relation could be done with one-way hashing function, like SHA (*Secure Hash Signature Standard (SHS)180-2*, 2002) or other. Thus, once object is created with public-private key pair, the ID is the hash of this public key. With this assumption it is impossible to recreate the same object with the same ID and other key pair in reasonable period of time.

3.7. Trusting new objects and nodes

To make possible to authorize all new objects and nodes in cluster, it is necessary to provide some mechanisms against this problem. Thus, mechanisms known from X509 (Cooper, Santesson, Farrel, Boeyen & Housley, 2002), OpenPGP (Callas, Donnerhacke, Finney, Shaw & Thayer, 2007), or other standards could be implemented. Once the object's key is signed by some authority, it could be trusted. There is no other way to automate this process with assumption that in any moment of time, cluster's infrastructure could be splitted.

Otherwise one should have centralized service, verifying the authenticity of objects in database or doing it manually.

3.8. Tracking changes in database

Described assumptions of this system architecture makes tracking history of whole cluster trivial task. Once each node has all notifications, at each point of time we could recreate whole history of cluster's state and information related to it.

3.9. Database as the communication channel

In most IT systems, database acts as the data store. In most cases dedicated interface is responsible for communication with external services. Between interface and database we have layer of logics, which validates requests incoming from interfaces (API) and modifies the database.

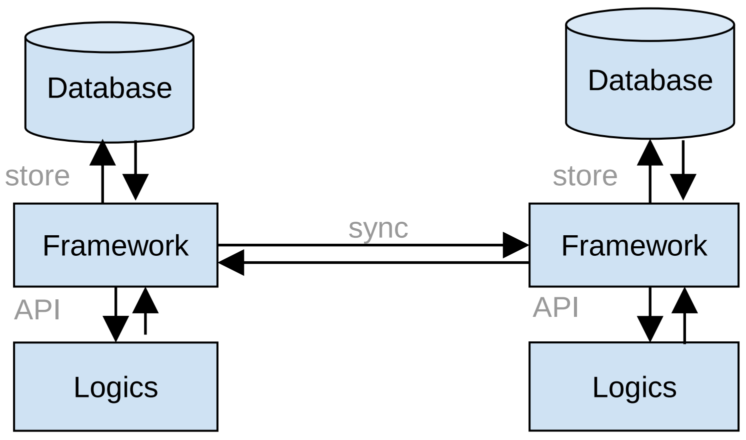

Fig. 7. Architecture of system using notifications as the communication channel

In proposed architecture it is recommend to use fact of information change – the notifications – as the communication channel (fig. 7). The mentioned communication channel in fact is not a database, but the framework intermediating communication with the data store. As mentioned in previous chapters, each node has logics, responsible for validating incoming notifications and making decisions about applying them to local information. From the other side, once state is changed at one node, the notification is broadcasted to other ones. Thus, database updates, together with described authorization mechanisms are an alternative to creating interfaces for communications. For example, changing state of one object in database will create notification of this fact broadcasted to all neighbors. If implementation of this concept provides broadcasting such updates to all neighbors, and all neighbors will retransmit it to own neighbors, the very strong redundancy will be provided. Moreover, using only proposed system as communication channel for all purposes, creates redundant and fault tolerant channel to communication. However still, it is highly independent on the cluster's physical networking. Handling updates in described in this paper way (chapter 3.A) also defines how to split-brains and re-joins of cluster will be handled.

4. EXISTING TECHNOLOGIES

4.1. IP network and ISO/OSI models

First technology, worth to compare with is the IP networking (in 4 or 6 versions). Basic principle is to provide connectivity between hosts (in our case nodes) and deliver messages in form of packets (notifications). With dynamic routing we could provide redundant paths inside IP networks and make it highly resistant for failures of single points.

However at this point our similarities are end. Of course we could extend IP networking with additional layers, protocols and tools to provide more similar way of communication, but, without implementing application layer of ISO/OSI we cannot do:

– security and authentication – exception is the IPSec protocol, but requires a lot of preconfiguration, what is in opposite to next point,
– autodiscovery and autoconfiguration,
– data store,
– data recreation/database history tracking,
– no default redundancy – configuring redundant connections requires additional routing configuration in whole network.

Thus, for purposes of the discussed architecture, IP network was insufficient solution, however it was used as the backing technology for connectivity. Main difficulty is total lack of data store, combined with cryptographic security layer, which is required in such systems. However some ideas known from IP networks was implemented in proposed architecture.

4.2. Blockchain (including Ethereum, Bitcoin network and other)

Most similar solution to proposed, which has the most common points with described here system is the general Blockchain technology (Nakamoto, 2008). It covers most requirements, mentioned in introduction, but has one, major difference. Blockchain was designed as the base for cryptocurrencies, so its main purpose is to securely make transactions between participants without any centralized point. One of most unique features of blockchain technology is the way how this database confirms the transactions across the network, known as consensus algorithm. This mechanism is used to create one, final version of database each ten minutes (approx.) by voting. Nodes across network (known as “miners”) are used to calculate hashe of block of transactions and thus make cryptographic confirmation of transactions in time. In Bitcoin’s network such hash should start with certain amount of zeros at beginning, by calculating SHA of block joined concatenated with random data, resulting proper hash. In Bitcoin this is called proof of work, due to generating hash requires a lot of computational work. In other decentralized systems other mechanisms are used (i.e. proof of stake). This mechanism is most important part of Blockchain and other decentralized systems, what makes it hard to forge.

Ethereum (Eyal, 2017; Tschorsch & Scheuermann, 2016; Wood, 2014) has the similar approach to the verification of smart contracts as the Bitcoin network. The difference is that miners in Ethereum network validate small programs (called smart contracts), instead of validating transactions. Thus, implementing Ethereum as the consistent, asynchronous information distribution system is not possible without large modifications of protocol. Additionally, both technologies (Ethereum and Bitcoin) are based on the same principles of underlying Blockchain. Moreover, the Blockchain’s architecture has multiple levels (participants, ledger services, miners). Discussed here clusters should be simple, with flat architecture of nodes with the same privileges.

Above approach makes the Blockchain a great solution for currency purposes. In distributed environment nobody can undo the transaction. Also nobody can forge recent transactions. In context of described system, Blockchain’s certainty is a problem. In case of split brains and rejoins we want to deal both parts of cluster, with all changes created by them. Finally, in proposed solution each node decides what part of data and updates is trustworth and which part should be ignored.

4.3. Agent systems

Agent systems, in traditional, scientific approach implement the agents, which are something more than objects. Agents have logics (Zhang & Zhang, 2004) and are able to take decisions based on incoming data, which could be incomplete or incorrect. Most agent systems could take decisions with a certain margin of error.

Thus, described in this paper system is most familiar with the agent systems, but extends it by adding some principles dedicated to drive clusters of nodes and handles various problems not present in agent systems.

5. RESULTS

Described functional requirements and ideas were implemented as stand-alone framework for creating distributed applications in form of library. The library could be found at the project's website ("Dinemic project", 2017) and Github repository ("Dinemic code repositories", 2017). Developed project is in the beta version at the moment of writing this article and is being tested to provide best stability and security, using all described in this paper ideas.

Created framework is designed to hide all complexity of technology, cryptography and networking from developer and provides Event Driven Development approach to creating distributed applications. Except the described functional assumptions, the features of solutions are:

- proxy to the underlying database and full ORM for C++ language – objects created through framework could store data in local data store automatically,
- automatic broadcasting notifications of changes in database over network – all updates on local database are sent to all neighbors over network,
- digital signatures of notifications, data encryption and control of incoming notifications by event handlers – before notifications are received and processed by framework, application can define its own event listeners, to take own actions before or after creating, updating or deleting objects. Event handlers also can prevent unauthorized changes in whole database, objects or particular fields of objects. All behavior could be defined by each application in cluster, what can result in very different database contents on various network nodes.

For applications using dinemic framework there is dedicated application skeleton, which guides developer to use it in proper way. This way allows to easily design applications as event driven. The base class for application – DApp class can be used to create basic models of applications and define necessary models within. Methods of this class (launch and oneshot) are dedicated to handle application installation, short actions (i.e. read or update database contents) and to remove objects when application is removed from node. Models created within the dinemic framework should use inheritance of DModel and DField classes to handle database readings and modifications. Additional class DAction is the interface to creating own handlers of changes in database across whole network. All updates incoming to node can trigger execution of DAction instances defined by user. In this way, application is able to define how to act when database changed fields of certain object at other node. Due to complexity of overall system, usage of al above classes in accordance with documentation is highly recommended. A wider description of framework API remains in documentation.

Implementation of whole framework is done in C++11 language. Beyond the ORM and described event handling mechanisms, the additional two types of configurable drivers were used. First one is the storage driver, which is used to store data in local node's data store. This allows to use on each node different database or key-value store to keep data. Available at this moment storages are: MemoryDriver, which stores data in node's RAM and RedisDriver, which uses Redis server and its key-value store. This driver guarantees more data safety in comparison with MemoryDriver. Second configurable driver is synchronization driver, which should used by each node in the cluster to communicate changes in local database. Synchronization driver monitors all changes in database made by local application and listens for updates generated by other nodes. Its role is also to handle all cryptography aspects - validation of digital signatures, chain verification and notification generation.

Beyond the development of framework itself, strong emphasis were put on aspects of security. During development, a lot of tests were made to confirm framework assumptions and cover all described in this paper functional requirements. Tests are designed to cover all units of framework and also to test its behavior in various network configurations. Additionally, performance tests were made, however publication of them is purposeless, due to the size of cluster, logical design, used hardware and driver set has very high impact on results. Author was not able to find any referral method to compare it with existing system.

6. CONCLUSIONS

Described in this paper assumptions of asynchronous, consistent information distribution for decentralized systems could be treated as the extended idea of agent system combined with the Public Key Infrastructure ideas and cryptography. However mentioned here problems and solutions will allow to create fully functional systems, without any single point of failure known from large scale systems. Described solutions could be applied to the particular applications in Computer Science context, to create the IoT-like systems, or to the whole computing clusters to provide consistent communication between them. The appliance of this solutions depends on needs and shown here assumptions are main guidelines to create decentralized, safe solutions.

In the comparison to the most popular in recent time decentralized system – Blockchain (and similar), this solution abandons one of it principles – the confirmation of transactions by proof of work mechanisms ("Proof of work explanation, Bitcoin project documentation", 2008). In described solution, most important principle was to authenticate origin of changes in whole cluster and trusting them absolutely. Main purpose of that was to trust modifications made by own applications in cluster and prevent unauthorized changes.

Purpose of creating this concept is to provide tool for creating distributed computing clouds, without any centralized point, but still with all basic functionality: storage, networking and virtualization. The main disadvantage of proposed solution and approach to handling management of cluster state is the large overhead of computations on each of cluster nodes. For systems handling up to thousands of virtual machines the amount of changes in central database is not so high to be problematic for described system. However for applications generating large volumes of data and changing data very frequently, this approach might be too slow and complex. In such cases, more appropriate might be using standard architectures, like microservices with strong emphasis on hardware redundancy (Balalaie, Heydarnoori & Jamshidi, 2016; Viennot, Lecuyer, Bell, Geambasu & Nieh, 2015).

ACKNOWLEDGEMENTS

All research and related work were funded and supported by cloudover.io ltd. company. Intellectual property for tools and libraries developed during research remain with cloudover.io ltd. company. The author would like to thank to Roman Krzanowski and Krzysztof Boryczko for theirs help, the Polish Linux Users Group ("Polish Linux Users Group", 2000) for promoting the project and research and the Crypto@Cracow Meetup ("Meetup Group Crypto@Cracow", 2016) group for help.

REFERENCES

Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2016). Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. In *IEEE Software* (*33*(3), pp. 42–52). USA: IEEE. doi:10.1109/MS.2016.64

Callas, J., Donnerhacke, L., Finney, H., Shaw, D., & Thayer, R. (2007, November). OpenPGP Message Format. Retrieved from https://tools.ietf.org/pdf/rfc4880.pdf

Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., & Polk, W. (2008, May). Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Retrieved from https://www.rfc-editor.org/rfc/pdfrfc/rfc5280.txt.pdf

Courtois, P. J., Heymans, F., & Parnas, D. L. (1971). Concurrent control with readers and writers. *Communications of the ACM*, *14*(10), 667-668. doi:10.1145/362759.362813

Davidson, S., Garcia-Molina, H., & Skeen, D. (1985). Consistency In A Partitioned Network: A Survey. *ACM Computing Surveys*, *17*(3), 341–370. doi:10.1145/5505.5508

Dinemic code repositories, (n.d.). Retrieved February 1, 2018, from https://github.com/cloudOver/libdinemic

Dinemic project, (n.d.). Retrieved February 1, 2018, from https://dinemic.io

DM-Crypt project, (n.d.). Retrieved February 1, 2018, from http://www.saout.de/misc/dm-crypt

Eyal, I. (2017). Blockchain Technology: Transforming Libertarian Cryptocurrency Dreams to Finance and Banking Realities. In *Computer* (*50*(9), pp. 38–49). USA: IEEE. doi:10.1109/MC.2017.3571042

Federal Information Processing Standards. (2002). *Secure Hash Signature Standard (SHS) (FIPS PUB 180-2).*

Fernstrom, C., Narfelt, K.-H., & Ohlsson, L. (1992). Software factory principles, architecture, and experiments. In *IEEE Software* (*9*(2), 36–44). USA: IEEE. doi: 10.1109/52.120600

GNU Privacy Guard project, (n.d.). Retrieved February 1, 2018, from https://www.gnupg.org/

Lippman, S. B. (1996). *Inside the C++ Object Model*, 1st edition. USA: Addison-Wesley Professional.

Meetup group Crypto@Cracow. (2016). Retrieved February 1, 2018, from https://www.meetup.com/pl-PL/Crypto-Cracow/

Nakamoto, S. (2008, October). Bitcoin: A Peer-to-Peer Electronic Cash System. Retrieved from https://bitcoin.org/bitcoin.pdf

Polish Linux Users Group. (2000). Retrieved February 1, 2018, from https://linux.org.pl

Proof of work explanation, Bitcoin project documentation. (2008). Retrieved February 1, 2018, from https://en.bitcoin.it/wiki/Proof_of_work

Tschorsch, F., & Scheuermann, B. (2016). Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies. In *IEEE Communications Surveys & Tutorials IEEE.* (*18*(3), pp. 2084–2123). USA: IEEE. doi: 10.1109/COMST.2016.2535718

Viennot, N., Lecuyer, M., Bell, J., Geambasu, R., & Nieh, J. (2015). Synapse: a microservices architecture for heterogeneous-database web applications. In *EuroSys'15, Proceedings of the Tenth European Conference on Computer Systems*, *Article No. 21.* USA, New York: ACM. doi:10.1145/2741948.2741975

Wood, G. (2014). Ethereum: a secure decentralised generalised transaction ledger. Retrieved from http://gavwood.com/Paper.pdf

Zhang, Z., & Zhang, Ch. (2004). Basics of Agents and Multi-agent Systems. In *Agent-Based Hybrid Intelligent Systems*. *Lecture Notes in Computer Science* (pp. 29–33). Berlin: Springer.