Annales Univ. Sci. Budapest., Sect. Comp. 56 (2024) 227–247

CELL-ORIENTED PROGRAMMING

Zoltán Horváth, Zoltán Porkoláb, Dániel Balázs Rátai and Melinda Tóth (Budapest, Hungary)

Communicated by Péter Burcsi

(Received January 9, 2024; accepted June 18, 2024)

Abstract. In the currently existing distributed architectures, there are different components that are optimized to fulfill different roles (e.g. databases, message brokers, load-balancers) in the whole architecture. Distributed architectures are highly complex. Several competencies are needed to be able to create and maintain such an environment. Furthermore, these systems have strict limitations in performance optimization as well because the interaction between the components is limited to their interfaces.
The Object-oriented programming (OOP) paradigm uses encapsulation to bind together the data and the functions that manipulate the data. However, OOP does not encapsulate one very important factor, the process that executes the functions themselves. (They are executed simply on threads.) On the other hand, the actor model binds the process to the data, but it does not have the high-level programming features of the OOP paradigm. COP aims to encapsulate the process as well to the corresponding data and functions and bring the best from these two approaches. With this new programming approach, the COP aims to make it possible to create a homogeneous system where all the nodes can form one single huge logical computer, which makes the whole architecture much more simple. COP aims to make it possible to write code to a distributed system as easy as we would to one single computer and make distributed computing magnitudes more effective, reliable, safe, and easier to code.

Key words and phrases: Distributed system, object-oriented programming, adaptive system.
2010 Mathematics Subject Classification: 68M14.
Supported by the ÚNKP-23-3 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund. Project no. TKP2021-NVA-29 has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.
This work is a detailed version of a MaCS 2020 presentation.

1. Introduction

Most of today's organizations are moving aggressively to adopt more agile, efficient software delivery and IT management practices to meet customers' evolving expectations. While agile development teams are focused on speed and agility, the traditional mantra of IT operations is maintaining application stability, even if that means slowing down the development [26].

Autonomous teams can select the programming language best suited to their project. This may include older languages like Java and C++, or relatively new platforms and languages to support specific development use-cases or design requirements (e.g., Node.JS, Go, and Rust). In other cases development teams are eschewing traditional relational databases in favor of NoSQL or document style data stores. These could be advantageous in projects where data requirements are indeterminate or evolving and where speed and scalability is more critical than up-front logical design and data integrity. Examples include MongoDB, PostgreSQL, and Cassandra [26].

The variety of the necessary tools for solving a non-evident problem generates many other challenges. The team responsible for the delivery must have well-skilled people for the many different competencies needed for the various tools and languages, making the development of distributed systems very expensive. On the other hand, as complexity increases, the risk of having serious security vulnerabilities might increase exponentially [29].

Moreover, we see the world moving towards a hybrid computing architecture that includes large centralized cloud data centers, smaller regional edge data centers, and even smaller, local edge micro data center sites. This environment presents unique management challenges that, require a cloud-based software management architecture to thrive in this complex ecosystem [11].

Building a centralized cloud-computing based infrastructure can be already a highly complex problem to solve for many enterprises. Many architectural questions have to be made respectively to the requirements that the system needs to achieve. Often, huge teams, hundreds of developers or more are working on systems that can solve a single business problem. As 5G is increasing the need for edge computing, the task of a developer is becoming an even more complex problem. Therefore as designing, deploying, and maintaining modern distributed systems becomes more and more complex, the necessity of radically new, simple, and effective software solutions might increase drastically.

In this paper we propose a new programming approach, Cell-oriented Programming, aiming to simplify the development of complex distributed applications.

When we write a standalone application, we do not need to care about fault tolerance, consistency, atomic transactions [19], network throughput, latency,

containers, container orchestration, canary deployment, etc. Everything can synchronously run in the memory; we only need to care with the core logic of our application. The essence of the Cell-oriented Programming is what if we would not have to care about all those things that make distributed scalable computing so hard, and it would be as easy or close as easy as writing a standalone application.

Cell-oriented Programming is based on the entities called Cells. Cells have behavior and a state just like objects in OOP, but they are encapsulating the corresponding process as well.

Just like living biological cells, the Cells in COP have a survival strategy, and they are able to move between computing nodes or even spread themselves to several nodes at the same time. (It is hard to find a helpful analogy to a complex and abstract concept. In our case the cells are rather similar to unicellular amoebas, which can highly change their shapes. However, even this analogy would not be perfect, since the location of the cells is not concrete, they can exist at multiple locations at the same time. There are no such objects in nature, except in quantum physics, but that analogy would be confusing in many other ways. Therefore we are using the cell analogy despite the imperfection.) This way, the Cells can optimize performance and ensure fault tolerance at the same time. Furthermore, this way the architecture of the overall system adapts automatically respective to the load that the system gets. Therefore we have an automatically self-organized architecture with a bottom-up logic instead of a classical manually designed top-down approach. Developers do not need to care about all the architectural questions and decide about the different components that should be used. They can just code as they would write a standalone application, only in a bit different style than we got used to in the case of Object-oriented Programming.

This paper is structured as follows. In Section 2 we will give a high-level overview of the Cell-oriented Programming approach. The technical details of the possible implementation are discussed in Section 3. In Section 4 we compare the COP approach with the most relevant currently existing technologies and frameworks. In Section 5 we discuss some examples of how the current technologies were used to solve problems related to the COP approach. Finally, this paper concludes in Section 6.

2. The COP approach

In this section we describe the high level concept of Cell-oriented Programming. More technical details about the implementation can be found in Section 3.

Cells are the fundamental building blocks of Cell-oriented Programming. They can both process and store data, and they can also communicate with other Cells. But they can also grow and shrink, they can move, and they can hibernate themselves. Think about Cells like small living entities, like biological cells. In the real physical world living organisms like cells have a survival strategy. They want to avoid the potential dangers and try to survive in a constantly changing environment. We can think about the Cells of COP similarly. They try to survive in a continuously changing environment as well. They are trying to avoid losing the data that they store, not to be able to respond fast enough when they are asked or to use too much memory when they are not in use.

To avoid these threats, they can do different things. If their state was read too many times, they can grow to more and more computer nodes to be able to balance their load. If they are becoming rather write-intensive, they can shrink so that not so many nodes have to be involved in changing their state all the time. If their hosting node is about to run out of memory or computing sources, they can migrate to other safer nodes. If they are not used for a while, they can automatically hibernate themselves to avoid consuming memory, just keep themselves in the mass storage using the minimum necessary space. If some other Cells from another computing node are always calling them, they can move to that node to minimize the network traffic and process their task locally and magnitudes faster.

All these activities can be done automatically. We do not need to code this behavior into the Cells. They can have their own survival strategy. Therefore, we do not need to care about such architectural questions. There is no need to decide which process should run where, nor where the data should be stored, or design everything with a top-down approach. The architecture optimizes itself with a bottom-up logic. As a result, we got a huge single logical computer built from many physical computers. We do not need to care about all the complexity of the different computing nodes and environments.

Furthermore: client computers can also store and handle Cells; therefore, they also can be part of the huge collective logical computer. This way, a high performing, fault-tolerant, and cost-effective, architecture can be achieved, which balances automatically between cloud-computing, edge-computing, and P2P computing. Furthermore, this behavior can come out of the box; therefore, writing code in a COP way can be close as easy as writing a standalone single node application.

2.1. Nucleus & CellMemory

The cells have two basic components: the behavior (we will call it *"Nucleus"*) and the state (*"CellMemory"*). The *Nucleus* of the *Cell* contains the

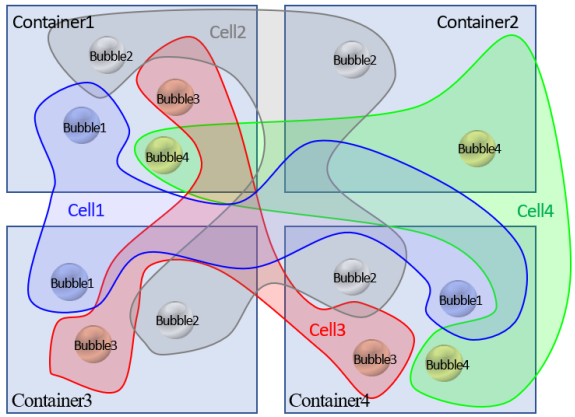

Figure 1. Structure of the Cells

methods, and functions that belong to a *Cell*, and the *CellMemory* contains the state of the *Cell*. The *Nucleus* is similar to a class in OOP and the *CellMemory* is similar to a concrete instance.

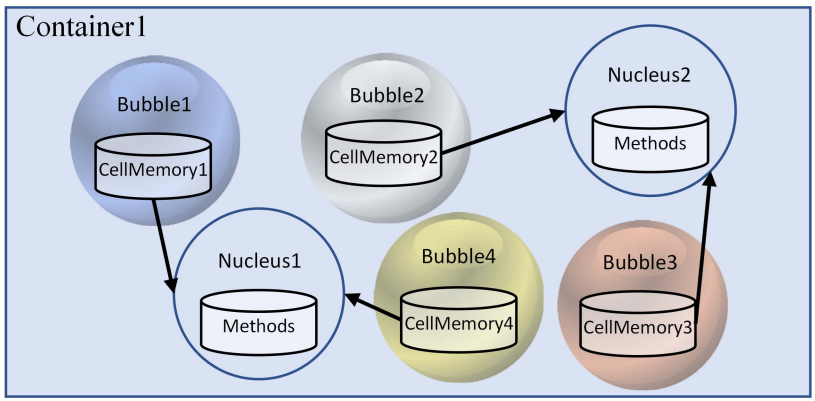

Figure 2. Nuclei and CellMemories

2.2. Bubbles

Cells are encapsulating the data with the corresponding methods, just like the Objects in the Object-oriented Programming. However, Cells are abstract entities. They do not exist physically on a particular computer node. They are spread across many different nodes. This gives them fault tolerance and scalability at the same time.

Despite the abstractness of the Cells, they still must exist somehow physically on the computing nodes. Therefore, we need to represent them physically on the different computer nodes. These representations we call *Bubbles*. Bubbles are the physical replications of a Cell on the different computer nodes. A Cell is built from Bubbles. The Bubbles of a Cell store the references of each other, they communicate, they synchronize themselves, and they can repair each other if there is a failure somewhere. In this way, Bubbles make the Cells fault-tolerant. Bubbles contain both the CellMemory and a reference to their Nucleus. Therefore they are not just storing a state, but they can make calculations on each node in parallel, ensuring that Cells are not only fault-tolerant, but also horizontally scalable.

2.3. CellContainer

Cells are stored in the so-called *CellContainers* indirectly. The Bubbles and the Nuclei that are building up the Cells and the Bubbles are stored in CellContainers directly. In the general case, each computer node has one CellContainer, but they may contain more for testing purposes or for optimizing to multicore CPU-s. All the operations or methods of any Cells can be triggered through a CellContainer. Therefore CellContainers are responsible for managing and encapsulating the corresponding state, behavior, and process of the Cells. The CellContainers are also responsible for communication over the network. The CellContainers together are building up a *CellContainer Grid*. CellContainers can take place on a cloud servers, edge servers, and client computing machines too. Therefore, cloud, edge, and P2P architectures are also achievable and a combination of these as well.

2.4. COP structure

COP is built on Cells. We have seen that Cells are built from Bubbles, and the Bubbles are stored in Containers. Figure 1. shows how this structure looks like. We can see a CellContainer Grid that stands from four Containers. We have four Cells altogether, and their Bubbles are spread across the Containers.

One might realize that Cells are abstract. They are just the sum of their Bubbles. However, it is important to distinguish the Bubbles from Cells because we cannot make any references to Bubbles. The developer who writes code in COP does not need to know anything about Bubbles. The developers are using the Cells, and the Bubbles are generated and organized automatically. Therefore, only Cells can be referenced and not Bubbles.

Figure 2. describes the inner structure of Container1 more detailed. The Bubbles are storing the state of the Cell in the CellMemory, and they have a reference to a Nucleus that contains the behavior, i.e. the methods of the Cell. It is shown that more Cells can use the same Nucleus. This is similar like in the

case of the objects and classes. More Objects can be instantiated from the same class. Here the Nucleus also represents the type of a Cell. However, Nuclei are also Cells. They are special Cells that contain the code of the methods which belong to their Cells. Their methods are automatically converted to in-memory functions when they are created or changed. The Nucleus Cells have a factory default Nucleus for handling these basic behaviors.

2.5. Encapsulation of the process

The OOP paradigm encapsulates the data with the corresponding functions. COP has the same encapsulation, but it encapsulates the corresponding process as well. This is necessary because COP is designed to execute in a distributed environment where the executing process is not as trivial as in the case of the OOP paradigm. OOP itself does not say anything about how the functions of an object should be executed. However, in general, threads are used for this purpose. Using threads for OOP is relatively simple, we have only a couple of threads maximum that are used. COP, on the other hand, can be executed on thousands of computing nodes, and managing the executing process is much more complicated.

In the actor model [12] the actor is the universal primitive of concurrent computation. It encapsulates the corresponding data, functions, and process. However, none of them can be reached from the outside. Actors can only interact through messages. Therefore the actor model does not provide a deterministic behavior, making the actor model based computation very complex. The actor model also misses the abstraction level of the OOP, which also implies limitations and added complexity [28].

To overcome those challenges we created a new concurrency model called the Traquest model [24, 25]. Discussing the Traquest model in more detail would be out of scope in this paper, but it can be reached in our previous paper. Here we only mention the core properties of the Traquest model to help the understanding of the COP programming approach.

Traquests are similar entities to promises [10], but it holds an extra reference for the nested Traquests, and it can form an atomic transaction through building Traquests-trees.

The Traquests should have transactional behavior for many reasons. If the developer has to care with all the variations when a Traquest fails that makes the code much more complicated. Also, in many use cases, ACID [19] properties are a must-have. The Traquest model provides a fully ACID concurrency model.

Every function in a Cell is executed in a Traquest, and every Cell can call other Cells only through Traquests. This way the Cells are encapsulating their corresponding processes, and they can be executed asynchronously, but still in

a deterministic way. Therefore creating algorithms in a COP style can be as straightforward and simple as writing a simple synchronous algorithm, but it can even be distributed to many computing nodes.

3. Coding in COP

There are many programming languages built for OOP, functional programming, procedural programming, or other paradigms. However, COP is a new programming approach. We wanted to avoid creating a new programming language and looked for a more optimal solution.

To only purely assemble a Cell, store its state in the memory, and execute the relating methods is relatively easy to solve. Cells could be built manually and injected into CellContainers. However, that would be very hard to code and maintain. Therefore, we used the TypeScript and TypeScript decorators [5], and built a library, called CellParser, which can convert Cells that are coded in a nearly traditional OOP style into Cell Nuclei. This way, we could avoid the need to create an entirely new programming language or a dialect for TypeScript. However, there are three minor conventions of what needs to be followed. Those conventions would not be necessary if a language was built for COP, but still, we could make the syntax easy to read and write, and the conventions are making a minor compromise. Figure 3. shows an example of how the code of a Cell looks like.

```
interface MainCellRemote extends CellRemote {
    hello(): Request<string>;
    sibling(): MainCellRemote;
}

@cell()
class MainCell extends Cell implements MainCellRemote {
    private _greet = "Hello world!";
    @request()
    hello() {
        return new Request<string>((respond) => {
            respond(this._greet);
        });
    }

    @reference(MainCell)
    sibling(): MainCellRemote {
        return cellRefer<MainCellRemote>(new Request((respond) => {
            this._greet += '!';
            respond({$cellID:this.$id,$brainID:'MainCell'});
        }))
    }
}
```

Figure 3. Cell source code example

Figure 3. shows the implementation of a very simple Cell, i.e. the Nucleus of a Cell. This Cell is a basic "Hello world!" example, with an extra functionality with a reference to itself to show how a reference to a Cell can be created. When the `hello()` method of the Cell is called, it answers with a greeting, but in a linked list we can call the siblings `hello()` method as well. Let us go through step by step on the code and see how this works exactly.

We can simply use the `@cell()` decorator and extend the Cell class to convert a class to a Cell's Nucleus. The first convention we must follow is that a class that will be converted can have a constructor, but it cannot have any arguments. This is because the Nuclei are storing an initial state for the Cells. To be able to figure out this initial state, the CellParser calls the constructor of the class. Therefore, to be able to do that, only no-args constructors can be used. However, if we want to describe some initialization logic, we can create a so-called *request* function which can be responsible for that. We will talk about request functions later.

3.1. Encapsulation and data hiding

The first thing we need to notice on Figure 3. is that there is not just one class converted to a Cell but a class and an interface. Here we immediately arrived at the second convention what we need to follow, which would not be necessary if this would be a new programming language. Cells have data encapsulation, just like objects. However, they have very different rules for data hiding. In case of OOP, we have the visibility-scopes defined by the `private`, `public` and `protected` keywords. In case of the Cells, we would need `remote`, `private` and `protected` keywords and even the private and the protected visibility restrictions should act differently. The remote interfaces like the `MainCellRemote` interface are existing to be able to handle these modified visibility scopes.

Remote scope. The remote scope properties are the only ones that Cells can see from each-other. This is like the public scope of the objects, but there is a difference. The remote scope properties can only be functions that are returning *Traquests*.

We need the remote scope because Cells can exist on any nodes. It is not ensured that a Cell can call the other Cell synchronously. Even if the called Cell has an available Bubble locally, it still might be in a hibernated state, and in that case, it requires time to read it from the storage.

Therefore, Cells can only reach each other asynchronously. This gives much freedom because many things can be done between the calling of a Cell begins and the answer arrives. The remote scope methods have two different types. They can be request or reference methods.

Request functions. The request functions are asking the Cell to make some operations: calculating something, reading from its own state, or writing something to its state, and they can also call other Cell's remote functions. We need to add a `@request()` decorator to our request functions, and the CellParser will automatically consider the function as a request function.

Reference functions. In the OOP, we have the principle of composition. This means that objects can store references to other objects and like that it is possible to reach any directly or indirectly linked objects through the references between the objects. In COP we also have composition. However, because Cells are abstract entities and they can be physically anywhere, therefore referencing to a Cell works differently than just storing a pointer in the memory. The reference functions are similar to request functions, but they are restricted to return a Request with a CellReference object that addresses the targeted Cell.

In the example code the `sibling()` reference function, however, does not have a Traquest return type, but instead, it has a fully synchronous return type, which seemingly gives back the referred Cell itself. This is one of the features that make COP very easy to code. The asynchronous behavior of a reference function is hidden. It can be called like a normal synchronous function. Since there is always a request function at the end of the references, it was possible to achieve this. Therefore, no nested callbacks are needed. This makes the code very clean and easy to write.

To create a reference function, we need to add the `@reference()` decorator to the function. To hide the asynchronous behavior of the reference function we should use the `cellRefer()` function. This function requires an asynchronous Request with a CellReference object as an argument and returns with the same object, but it fakes the type, and it says that the return type is the remote interface that belongs to the referenced Cell. This way, TypeScript believes it is a standard object reference, and it can still recognize potential syntax errors compile time.

3.2. Proposed UML extension for COP

The Unified Modelling Language (UML) [14] has notations for OOP, which is very useful when we want to discuss how an object-oriented architecture works. We believe that it would be useful for COP as well, but UML has no components for that since COP is a new programming approach.

Therefore, we propose an extension to UML with new notations to describe the elements of COP. Of course, at this moment, these are not official components of the UML, these are just recommendations, but these new notations are very useful to be able to talk about COP. We will investigate in the future how to handle these with standard UML elements or extensions. Figure 4. shows an example, how these notations look like.

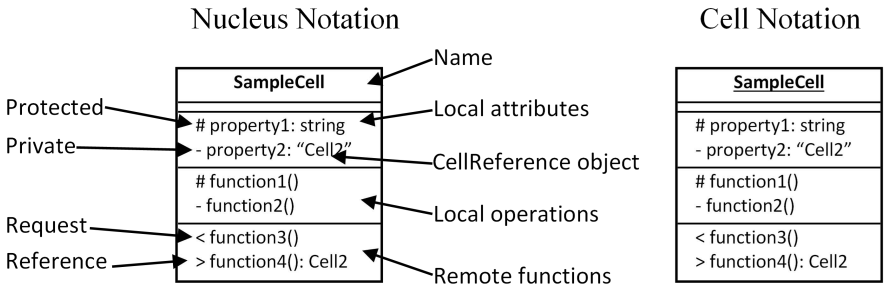

Figure 4. COP extensions for UML

Please remember, that even though in COP there are `protected` and `private` visibility scopes just like in OOP, but their meaning is different.

4. Comparison with existing technologies

To investigate how a suitable scalable backend is achievable, there are several architectures that should be examined. First, we present a use-case example, and next, we investigate the potential alternative technologies.

Exemplary is use case. However, COP is a general-purpose programming approach; it is worth examining it through a concrete use case example. COP can provide most of the benefits when facing problems that require complex real-time, high throughput, and low latency systems. Such a use can be a real-time IoT navigation service. IoT devices can be consumer drones, smartphones, smart watches, or any devices that can capture its location, send it to the cloud, and modify its behavior based on other IoT devices' location.

Let us say we need to build an IoT device control service (IDCS) where IoT devices located nearby each-other can locate themselves. Suppose that each device has to continuously send its own location to the IDCS, and the IDCS has to reply with the location of the 10 nearest devices. Suppose that each device can update its location 20 times each second, and the latency in getting the newest location of the 10 nearest devices should not be bigger than 200 ms. Suppose we use an octree [18] space partitioning algorithm for partitioning the location coordinates in the 3D space.

In COP, we can easily create Cells for the nodes in the octree and references to the other nodes. The neighboring cells will automatically be optimized for the same computing nodes, and most of the operations will happen in-memory time.

Multitier architectures. Probably the most standard solution for creating scalable backends is the multitier architecture [22]. In a multitier architecture, the applications must communicate with the databases to change or read the global state. Every read or write operation implies a message on the network. Therefore, no matter how efficient the database is, simply only communicating with the databases generates too much load. Pipelining [27] could be a possible solution in theory, but since we are using octree, many operations depend on each other. When we modify the tree structure of an octree, we need to read the properties of a node to decide if new nodes should be created or existing ones should be erased, and we need to do this in many iterations. If we need to reach the database in each iteration, the latency will be very high, and we have limitations in pipelining.

For the sake of simplicity let us use a quadtree example instead of octree. Suppose that we need to modify a node and therefore we need a function called `setNode` with the following signature: `setNode(address,properties)`. Suppose we have the following function call:
`setNode("3120",someProperty)`. This single function call will implicate the following operations:

1. Get the root node $\Longrightarrow$ 1 read operation
2. Get its 3^{rd} child.
3. Check if it is a leaf in the tree.
4. If yes
   - Create four children $\Longrightarrow$ 5 write operation
   - Set the neighbors $\Longrightarrow$ 4 read & 8 write operation
   - Recursively refresh the parents $\Longrightarrow \approx$ 4 read & 3 write operation
5. else
   - Get the 0th children and check if it is a leaf $\Longrightarrow$ 1 read operation
   - Change the node property $\Longrightarrow$ 1 write operation
   - Recursively refresh the parents $\Longrightarrow \approx$ 4 read & 3 write operation
   - Recursively check and erase homogeneous parents and modify the neighbors $\Longrightarrow \approx$ 6 read & 22 write operation

The above-mentioned quadtree structure can be visually represented by the Figure 5. for better understanding.

This is a simple example with around 15 read and 20 write operations per changing one single node. This means one device can easily generate more than

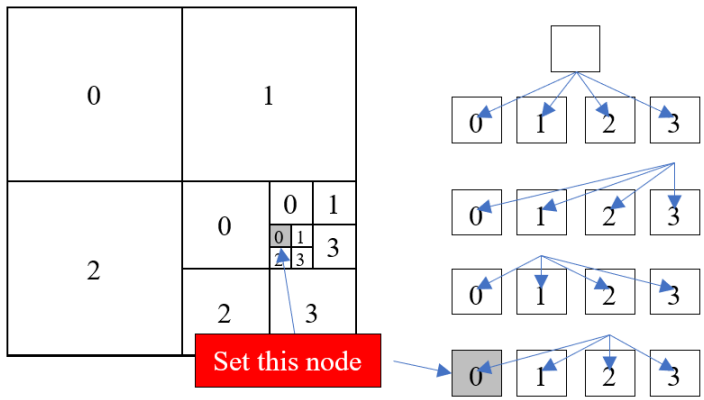

Figure 5. Modifying a quadtree

500 network operations per second to the database. It is no matter how many types of databases are investigated, the multitier-architectures will not solve the problem.

COP, on the other hand, optimizes its topology automatically. In most cases, all the operations will be made inside one single computing node; therefore, there is no need to communicate on the network each time. For the fault tolerance, the Bubbles have to be kept in synch of course, but those messages can be bulked together (we can assume around three separate network messages will be necessary only). Calculating with the 200 ms latency, this means approximately 15 messages are required instead of 500. The difference gets even more interesting when we calculate with 1000 devices. In this case, with a multitier architecture, we need 500 000 network messages per second. In comparison, COP will still implicate only 15 messages since only the operations for synchronizing the Bubbles are necessary, and they can be bulked together.

This optimization can be achieved thanks to two factors. On one hand, the survival strategy of the Cells optimizes the topology of the Bubbles, therefore minimizing the necessary number of the network interactions. And on the other hand the Traquest model has the buffering and lazy synchronization capabilities which allows to await the messages and send them in larger bulked packages.

Actor model. The actor model originated in 1973 [13]. The Actor model is a very widely used concept from programming to business process modeling. In programming, it refers to a very effective concurrency model where the so-called "Actors" behave as universal primitives of computation [12]. They can store a small piece from the global state, and they can also process a small piece of information. They are bound together with a message queue. Therefore they can communicate with each other through messages. They are extremely fast because if two Actors are on the same computer node, the whole messaging

process between them can be done in the local memory. They are also flexible to use and scalable since the Actors can take place on different nodes and move between them to optimize their performance.

Actors can be super-efficient, although there are disadvantages as well. It is tough to implement transactions with the Actor model in an efficient way [1]. Actors can call each other very fast, but they are just blindly consuming and generating messages. There is no mechanism for handling deadlocks or scenarios when Actors are infinitely sending each other messages in an infinite loop. It is very easy to write deadlocks, and it is hard to create a complex yet robust system [28] since mastering the actor model requires a steep learning curve.

The most popular Actor model implementations are aiming to reach fault tolerance by the so-called Supervisor Strategy [17]. However, Supervisor Strategy does not do anything with data loss coming from a failing node or data corruption. Therefore, it is rather a kind of distributed error handling mechanism for a non-blocking Actor based environment than a real solution for fault tolerance solution.

If any node of the quadtree in the IDCS gets lost, that means the whole branch is lost. Any inconsistency in the tree structure would have the same results. Therefore ACID properties is a required property for this use case.

Thanks to location transparency [15], Actors can move across the nodes to distribute the load and to optimize efficiency. However, when a particular Actor gets a massive load that a single server cannot handle, there is no solution since the Actors are atomic particles. They physically belong to one node at a given time.

For example, in our use case, the root node of the IDCS would get a massive load because all the IoT devices would read the root node at least once when they update their location or query the surrounding devices. If an actor represents this root node, there would be no chance for it to scale horizontally. Meanwhile, if we represent the root node with a Cell this horizontal scaling happens automatically thanks to the abstract location.

All in all, the Actor model is an amazing technology. However, it still has many limitations and it requires a relatively low-level coding and keeps many of the problems for the developer.

In the case of IDCS, the octree space partitioning nodes could be built from actors, and the representations of the IoT devices could be actors as well. However, persisting the unused nodes, and solving the fault tolerance issues, moving the actors between computing nodes should be done manually, making the overall system very complex.

Serverless. The serverless architecture [23] is a very interesting concept. FaaS services like the AWS Lambda or the Google Cloud Functions can make

the whole development and DevOps process much easier since they can scale automatically. But the functions do not store any state, and they are using external resources for the state storage just like the Business Logic Layer of the multitier architectures; therefore, they have the same limitations.

Stream processing. Stream processing technologies like Kafka are also interesting to investigate. On the Apace Kafka homepage, we can read the following: "Kafka is used for building real-time data pipelines and streaming apps. It is horizontally scalable, fault-tolerant, wicked fast, and runs in production in thousands of companies." [2]. It seems perfect in every way for IDCS, yet stream processing has its limitations. However, simple location data can easily be converted into a data stream; the octree structure of the IDCS makes it very hard to stream up octree nodes. The nodes are all individual records of data, and they have references to many directions. The clients are randomly moving amongst those directions, accessing them, and writing them. It is way more complex than, for example, a video stream or a message queue. Kafka uses so-called "Topics" to handle different streams at the same time [2]. For the IDCS, every octree node could be a Topic, but Topics were not designed for that. Topics are created to handle the streaming of even Gigabytes of data, not to have millions of them, which are handling a couple of bytes. Therefore, Kafka is apparently not the solution for this case either.

Batch processing There are powerful technologies for distributed batch processing too, like MapReduce [7] or Spark. However, as its name indicates, the problem is when we have real-time data. Even Spark Streaming only uses micro-batches. Batch processing cannot solve the problem, and in the case of the IDCS the data is generated absolutely in real-time by the users.

5. Further related work

Object-oriented programming. In the software development industry, it is a common understanding that **Object-oriented Programming** as a programming paradigm has four fundamental principles, which we also might call the four pillars of Object-oriented Programming [4]. However, when reviewing the body of work on OOP development, most authors simply suggest a set of concepts that characterize OOP, and move on with their research or discussion. Thus, they are either taking for granted that the concepts are known or implicitly acknowledging that a universal set of concepts does not exist. Several authors, asserting there is no clear definition of the essence of OOP, have called for the development of a consensus [3].

Since a comprehensive comparison of the OOP paradigm and the COP approach would be out of scope for this paper and might require further research we restrict hereby for the so called four pillars of OOP.

Abstraction. Nuclei are similar entities to Classes, and they can be defined in a way that they are entirely stateless. In this case, Nuclei practically behave as abstract Classes, which means abstraction is not a principle that could differentiate COP from OOP.

Inheritance. Inheritance expresses "is-a" and/or "has-a" relationship between two objects. Using inheritance in derived classes, we can reuse the code of existing super classes. In COP the same effect can be achieved by referencing the Nuclei on each other; therefore, inheritance is a non differentiating principle as well.

Polymorphism. It means one name many forms. Static polymorphism is achieved using method overloading and dynamic polymorphism using method overriding. Both hare similarly existing features of COP and OOP, and the viability of generics is still up to be researched in COP. Polymorphism, as a general concept, is not a differentiating factor either.

Encapsulation. In OOP encapsulation is the mechanism of hiding data implementation by restricting access to public methods. Instance variables are kept private and accessor methods are made public to achieve this. COP encapsulates Cell variables and methods just as OOP, but it also encapsulates the corresponding process. Therefore, COP must have completely different scopes for information hiding as well since Cells can not modify the internal behavior of an other Cell even if they share the exact same Nucleus and the a Cell can be called only at its remote scope.

Self-adaptive systems for IoT. As COP can optimize the location of the Bubbles, it becomes a completely self-adaptive architecture. Self-adaptive systems are highly used in the IoT sector and many other sectors already. However, in the literature, we can mostly find use-cases where the system can scale horizontally automatically adapt to the changes in the load, but the architecture itself can not adapt to the changed requirements, and it is still designed manually by a top-down approach. The article "Self-adaptive IoT Architectures" [21] investigates and compares several of such architectures but concludes no solutions that can automatically detect, change, and optimize the pattern of the architecture itself.

The authors in [9] highlight another important aspect of the self-adaptive systems. Namely that realising the adoption of the system also can incur costs and might reduce or eliminate the advantages of a self-adaptive architecture. At COP the costs of the adoption depend on the efficiency of the algorithm of the Cell survival strategy. Creating and optimizing this algorithm requires further research; however, the migration of the Bubbles can be grouped, and probably a highly optimal solution can be found later.

Self-adaptive microservice systems Microservice architecture is a very much used approach in the industry since the separation of concerns makes it possi-

ble to design robust and easy to maintain distributed applications. However, designing self-adaptive systems involves making design decisions about the environment while it is being observed and about the system itself, and then selecting adaptation mechanisms that are thereafter enacted. In the context of a microservice application, the design space for making self-adaptation decisions is even more complex due to the large number of runtime components and their independent and highly dynamic nature [20].

In the case of the COP approach, we do not need to face such problems. The separation of concerns happens on the design pattern level, and we do not need to deploy different services. Because the Nuclei are also Cells, they can be deployed on the fly separately. Therefore similarly to the microservice architectures we will not have a large monolith application, but still we can avoid the complexity of the microservice solutions.

Natural computing There are many nature-inspired models of computation. The article "P Colonies with a Bounded Number of Cells and Programs" [6] introduces a membrane computing method where "cells are represented by a collection of objects and rules for processing these objects, they are the basic computing agents in this formal model of computing". Our "COP" method is much less inspired literally by biological cells, their structure, and their behavior. We had an abstract and complex model and the we have found that "cells" might be the least imperfect analogy. However, COP can not be considered as a natural computing method as we first had the concept and later we searched for an explanatory methodology. Also COP can not be considered as a membrane computing method as it does not have any hierarchical or non-hierarchical membranes. In COP all the cells are on the same hierarchical levels. Using cells as an analogy that refers to a living entity is rather reflects a model similar to OOP, where the objects can have survival strategies to adapt to a changing environment.

6. Conclusion

It is well known that designing and developing a resilient, horizontally scalable distributed system is a highly complex task and will be even more complex in the future. We discussed that the complexity of the distributed systems is an enormous problem. It increases the cost and the risk of the development, it makes it harder to make changes and work in an agile way therefore. Complexity also results in unpredictable security vulnerabilities. Furthermore, as the 5G and edge computing is getting more and more widespread, the complexity of the distributed systems will increase even more.

The necessity of radically new, simple, and effective software solutions might increase drastically. Simplifying this process while keeping the performance goals or even improving it is crucial.

We have introduced a new way of creating distributed systems that are built on top of the context of cells. We named this new approach Cell-oriented programming (COP). COP is a concept that might bring a solution to the complexity issue of the distributed systems. Cells are similar entities to objects in OOP; however, cells are encapsulating the corresponding process as well, not only the corresponding functions and data. This difference has led to many conceptual changes and indicated a new programming approach for distributed computing.

Significant progress was accomplished to create the first working prototype of COP. A whole API was built to clean the concept in advance as much as possible, and also some of the components are working and have been already tested. It is easy to code with COP since it automatically handles all the scalability issues and gives an environment where we can code just like we would create a simple standalone application.

Meanwhile, COP has some extreme abilities. The performance could not be tested yet, since a full prototype implementation is necessary to measure the performance, but it is already possible to make some estimations. In a theoretical use case of the IDCS, where 1000 devices are using a given computing node at the same time, COP requires roughly 30,000 times fewer messages between the computing nodes than an architecture using databases, due to many factors

- the topology optimization,
- managing the global state with high probability right next to the location of the computing and not on separate nodes dedicated to the persistence,
- buffering the syncronization messages and ensuring full consistency with the Traquest model.

We published an article regarding the theoretical background of the Traquest model [25], the concurrency model for COP. It shows why such a performance improvement is possible in much more detail.

In general, COP will even be able to not only scale on servers but on client computers too, resulting in a lightning fast, easy to code and cost-effective cloud/edge/peer-to-peer hybrid system. COP can eliminate the boundaries between cloud, edge, and peer-to-peer computing by creating a bottom-up self-organizing architecture that automatically decides about the location where the information should be stored and processed without any manual intervention.

With further research and clarification, COP might even go beyond the limitations of the Object-oriented Programming paradigm for distributed systems and might have a chance to become an entirely new programming paradigm of its own. In this paper, we showed the basics of this new concept to serve as a foundation to establish this new programming approach direction.

References

[1] **Akka,** Transactors (scala) (2018)
https://doc.akka.io/docs/akka/1.2/scala/transactors.html

[2] **Apache Software Foundation,** Apace kafka (2017)
https://kafka.apache.org/

[3] **Armstrong, D.,** The quarks of object-oriented development, *Communications of the ACM* (2006)
https://www.researchgate.net/publication/220425366

[4] **Chandel, M.,** What are four basic principles of object oriented programming? Medium (2018)
https://medium.com/@cancerian0684/what-are-four-basic-principles-of-object-oriented-programming-645af8b43727

[5] **Cherny, B.,** *Programming TypeScript*, O'Reilly Media Inc., 2019.

[6] **Csuhaj-Varjú, E., M. Margenstern and G. Vaszil,** P colonies with a bounded number of cells and programs, *Membrane Computing. 7th International Workshop, WMC 2006, Leiden, The Netherlands*, (2006)

[7] **Dean, J. and S. Ghemawat,** MapReduce: simplified data processing on large clusters, *Communications of the ACM*, (2008)

[8] **Dénes, T.,** Real face of János Bolyai, *Notices of the American Mathematical Society* (2011)
http://www.ams.org/notices/201101/rtx110100041p.pdf

[9] **Donckt, J.V.D., D. Weyns, M.U. Iftikhar and R.K. Singh,** Cost-benefit analysis at runtime for self-adaptive systems applied to an internet of things application, *ENASE 2018: Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering*, (2018)

[10] **Friedman, D. and D. Wise,** The impact of applicative programming on multiprocessing, *International Conference on Parallel Processing*, (1976)

[11] **Gillott, I.,** The role of edge computing in 5g, *Data Economy Magazine* (2018)
https://data-economy.com/the-role-of-edge-computing-in-5g/

[12] **Hewitt, C.,** Actor model of computation, *ArXiv* (2015)
https://arxiv.org/vc/arxiv/papers/1008/1008.1459v8.pdf

[13] **Hewitt, C., P. Bishop and R. Steiger,** A universal modular actor formalism for artificial intelligence, *IJCAI* (1973)
https://www.ijcai.org/Proceedings/73/Papers/027B.pdf

[14] **Jakimi, A. and M. El Koutbi,** An object-oriented approach to UML scenarios engineering and code generation, *International Journal of Computer Theory and Engineering*, (2009)

[15] **Kim, W.Y. and G. Agha,** Efficient support of location transparency in concurrent object-oriented programming languages, *Supercomputing'95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, (1995)

[16] **Lightbend Inc.,** Akka quickstart with java (2018) https://developer.lightbend.com/guides/akka-quickstart-java/

[17] **Lightbend Inc.,** Fault tolerance (2018) https://doc.akka.io/docs/akka/2.5.3/java/fault-tolerance.html

[18] **Meagher, D.,** Geometric modeling using octree encoding, *Computer graphics and image processing* (1982)

[19] **Medjahed, B., M. Ouzzani and A. Elmagarmid,** Generalization of ACID properties, *Purdue e-Pubs* (2009)

[20] **Mendonça, N.C., P. Jamshidi, D. Garlan and C. Pahl,** Developing self-adaptive microservice systems: Challenges and directions, *IEEE Software* (2019)

[21] **Muccini, H., R. Spalazzese, M.T. Moghaddam and M. Sharaf,** Self-adaptive iot architectures: an emergency handling case study, *ECSA '18: Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings* (2018)

[22] **Pacifici, G., B. Urgaonka, P. Shenoy, M. Spreitzer and A. Tantawi,** An analytical model for multi-tier internet services and its applications, *SIGMETRICS* (2005) http://lass.cs.umass.edu/papers/pdf/SIGMETRICS05.pdf

[23] **Pekkala, A.,** Migrating a web application to serverless architecture (2019)

[24] **Rátai, D.B., Z. Horváth, Z. Porkoláb and M. Tóth,** Traquest model - a novel model for ACID concurrent computations, *The 12th Conference of PhD Students in Computer Science - Proceedings* (2020)

[25] **Rátai, D.B., Z. Horváth, Z. Porkoláb and M. Tóth,** Traquest Model, *Acta Cybernetica*, **25(2)** (2021), 435–468. https://doi.org/10.14232/actacyb.288765

[26] **Ravichandran, A., K. Taylor and P. Waterhouse,** *DevOps for Digital Leaders*, Springer Science+Business Media LLC (2016)

[27] **Redis Labs Ltd.,** Redis documentation - Using pipelining to speedup Redis queries (2020) https://redis.io/topics/pipelining

[28] **Vuckovic, J.,** What's wrong with the actor model (2015) https://jaksa.wordpress.com/2015/10/13/whats-wrong-with-the-actor-model/

[29] **Yovel, Y.,** Complexity is the real vulnerability (2015) https://www.informationsecuritybuzz.com/articles/complexity-is-the-real-vulnerability/

Z. Horváth, Z. Porkoláb, D.B. Rátai and M. Tóth
Eötvös Loránd University
Faculty of Informatics
Budapest
Hungary
hz@inf.elte.hu
gsd@inf.elte.hu
danielratai@inf.elte.hu
toth_m@inf.elte.hu