electronics

MDPI

Article

Fundamentals of a Novel Debugging Mechanism for Orchestrated Cloud Infrastructures with Macrosteps and Active Control

Bence Ligetfalvi [1], Márk Emődi [1,2], József Kovács [1] and Róbert Lovas [1,*]

[1] Institute for Computer Science and Control (SZTAKI), Eötvös Loránd Research Network (ELKH), Kende u. 13-17, 1111 Budapest, Hungary; bence.ligetfalvi@sztaki.hu (B.L.); mark.emodi@sztaki.hu (M.E.); jozsef.kovacs@sztaki.hu (J.K.)

[2] Doctoral School of Applied Informatics and Applied Mathematics, Óbuda University, Bécsi út 96/B, 1034 Budapest, Hungary

* Correspondence: robert.lovas@sztaki.hu; Tel.: +36-1-279-6275

Abstract: In Infrastructure-as-a-Service (IaaS) clouds, the development process of a ready-to-use and reliable infrastructure might be a complex task due to the interconnected and dependent services that are deployed (and operated later on) in a concurrent way on virtual machines. Different timing conditions may change the overall initialisation method, which can lead to abnormal behaviour or failure in the non-deterministic environment. The overall motivation of our research is to improve the reliability of cloud-based infrastructures with minimal user interactions and significantly accelerate the time-consuming debugging process. This paper focuses on the behaviour of cloud-based infrastructures during their deployment phase and introduces the adaption of a replay, and active control enriched debugging technique, called macrostep, in the field of cloud orchestration in order to provide support for developers troubleshooting deployment-related errors. The fundamental macrostep mechanisms, including the generation of collective breakpoint sets as well as the traversal method for such consistent global states, have been combined with the Occopus cloud orchestrator and the Neo4J graph database. The paper describes the novel approach, the design choices as well as the implementation of the experimental debugger tool with a use case for validation purposes by providing some preliminary numerical results.

Keywords: cloud; IaaS; debugging; orchestration; replay; active control; troubleshooting; macrostep

Citation: Ligetfalvi, B.; Emődi, M.; Kovács, J.; Lovas, R. Fundamentals of a Novel Debugging Mechanism for Orchestrated Cloud Infrastructures with Macrosteps and Active Control. *Electronics* **2021**, *10*, 3108. https://doi.org/10.3390/electronics10243108

Academic Editor: George Angelos Papadopoulos

Received: 21 November 2021
Accepted: 10 December 2021
Published: 14 December 2021

1. Introduction

Infrastructure-as-a-service (IaaS) cloud computing systems allow automated construction and maintenance of virtual infrastructures [1] leveraging on the concept of virtual machines (VMs) as the fundamental building block.

1.1. Challenges

Thus, IaaS systems enable the creation, management and destruction of VMs, but the current IaaS solutions struggle with the management of multiple VMs—or focus particularly only on network management among multiple VMs. Recent research efforts were able to address and answer these issues with the cloud orchestrator concept [2,3]. On the other hand, complex service deployment and maintenance scenarios of such cloud orchestrators still pose new challenges since software engineers and testers must face (among others) the probe effect, the irreproducibility, the completeness problem, and also the large state-space to be somehow handled during the debugging/troubleshooting phase.

For instance, let us assume that a given orchestrated cloud deployment scenario always generates correct results on a particular cloud platform or on a set of cloud platforms in hybrid and federated clouds (where the software engineers originally developed and deployed their services) but often fails on other cloud platforms operated by other

IaaS providers. Mostly, the reason for this behaviour is the varying relative speeds of deployment tasks together with the untested race conditions in the non-deterministic environments. The different timing conditions might be occurring more frequently on cloud-based platforms than on dedicated clusters or traditional supercomputers because of the different implementation of the underlying operating systems/communications layers and the unpredictable network traffic, CPU loads or other dynamical changes. The above-described phenomenon can be very crucial because one cannot ensure that the cloud-based deployment always provisions (captures) the same computing nodes with almost the same timing conditions in case of (re)deployment or VM failure.

1.2. Motivation

Our research motivation is two-fold: (i) providing mechanisms for the development of reliable cloud-based research infrastructures, (ii) enabling less human interactions during automatised and accelerated debugging phase of orchestrated infrastructures. In this way, the development and operation costs of research infrastructures might be significantly reduced, and the cloud users may get a higher quality of service. This paper focuses on both motivations.

1.3. Solution

One feasible way to prove the cloud platform agnostic feature of complex deployment and maintenance strategies is to leverage on advanced systematic debugging/troubleshooting methods in order to find the timing or architecture dependent failures in the designed deployment description and orchestrator. For this purpose, our research team applied and extended the macrostep-based debugging [4] methodology that has been introduced originally for message-passing parallel programs and developed in the P-GRADE graphical programming environment [5].

The main essence of the macrostep methodology is to discover the different timing combinations among the concurrent behaviour of distributed events. While in the message-passing environment, the communication primitives were analysed by macrostep, in cloud orchestration, the parallel deployment of entities (software components on virtual machines) represent the source of possible concurrent behaviour. In the original macrostep technology, the central controller was able to enforce the program to be executed in a way to simulate or enforce all possible timing conditions, i.e., to execute the program through all possible execution paths to find the issues related to concurrent behaviour. The technique is based on applying (collective) breakpoints at certain points in the execution of concurrent activities performed by the various software components. Once the breakpoints are set, the execution of the components are controlled in a way to traverse the entire deployment process towards extreme, unexpected situations that have a low chance in normal conditions.

For orchestration, the goal is to apply this technique to investigate the behaviour of the deployment steps of a cloud orchestrator for particular cloud infrastructure, i.e., network of cloud virtual machines with installed software components and services interconnected to each other.

The first experimental prototype introduced in this paper has been elaborated based on the Occopus cloud orchestrator [6,7] framework. Occopus is an open-source cloud orchestrator software tool that has been designed and implemented by SZTAKI to support the research and experiments in the field of cloud orchestration. Similarly to other orchestration tools, Occopus applies the de-facto standard cloud-init tool to contextualise the new, launched virtual machines in order to start the services. The contextualisation itself is the place where the infrastructure deployment can be manipulated, i.e., artificially influenced in order to reach the desired (faulty) behaviour of the cloud infrastructure.

The paper is structured as follows. Section 2 first introduces related works on the field of debugging and troubleshooting on cloud orchestration. Macrostep-based debugging is detailed in Section 3 to understand the concept of collective breakpoint based troubleshoot-

ing. Then the architecture and operation of the prototype are described in Section 4. Section 5 is diving into the nature of manual and automatic debugging support, while Section 6 is about the visualisation feature of the prototype. Section 7 provides numerical results and details on our experiences using the new debugger tool. Finally, we conclude our results in Section 8.

2. Related Work

In the literature, some related attempts are described for HPC applications [8], but the typical example is the remote cloud debugging [9] feature for Windows Azure Cloud Services [10] that can be considered the most basic functionality of all distributed debuggers.

Sharma et al. [11] developed an endpoint-based monitoring solution that allows tenant-level monitoring. In addition, the system stores the history of the metric data for other uses, e.g., history evaluation or validation for SLA compliance. The solution offers monitoring customisation, extensibility and portability, but the use of the tool is limited to OpenStack-based cloud resources, and there is no active control mechanism offered for debugging purposes.

Baek et al. [12] created a monitoring solution where special loggers are inserted into the cloud components. The solution processes the logs and creates a resource graph from them. The resource graph presents the changes of the resource events and can be queried to define previous state changes. However, the tool's functionality is limited because it does not investigate the guest system in detail and need to inject the components into the system. It works only in an OpenStack environment without active control and other advanced debugging mechanisms.

Cloud debugger [13] developed by Google is able to follow the application states in real-time without the need of slowing down or stopping the application. It can take a snapshot from a desired state of the program, i.e., the call stack, and the variable values are searchable and verifiable. Another important feature is the so-called logpoints, which is able to create custom string and variable logging mechanisms in the code. Since it is a commercial product, the tool is available only for Google Cloud with a limited set of supported programming languages. Cloud debugger cannot provide advanced functionalities for handling multiple, orchestrated VMs.

Smara et al. [14] proposed a fault detection method for clouds leveraging on the concept of the acceptance test. In their framework, the main aim is to construct fail-silent cloud components, which have the ability of self-fault detection. On the contrary, our approach can create a series of consistent global cuts (or states) for error checking and debugging in the distributed environments.

According to Zhang et al. [15] two main categories of cloud fault detection can be distinguished: rule-based or statistical detection. Rule-based detection methods are built on simple rulesets on the error message and record components, or basic decision trees can be built using multiple rules and queries. Our approach is to support both categories. However, our work focuses on the mechanism of how to traverse the state space where such rule-based or statistical approaches might be applied later on.

Quroush et al. [16] described a record and replay mechanism for cloud-based multi-tenant services that enables software developers to debug their application in the replay phase after a failure detected in the recording phase. Their approach has promising results, but there is no support for systematic traversing of the state space and is limited for script-based deployments.

Goossens et al. [17] also introduce a communication-centric debugging method, but their work covers the problem at the hardware level and based on transactions.

By combining the macrostep debugging (see Section 3) method and the features of Occopus cloud orchestrator, the presented work attempts to overcome the limitation of existing debugging solutions since even the most widely used cloud providers nor the state-of-the-are debuggers tools do not offer high level and advanced debugging facilities to their users that are similar to the described macrostep-based concept.

3. Macrostep-Based Debugging

3.1. Original Concept for Message-Passing Programs

During debugging parallel and distributed programs, several aspects have to be taken into account, one being that parallel programs show non-deterministic behaviour, making the reproduction of erroneous runs a difficult problem. This can be due to several factors, like differing relative CPU and/or memory speeds, operating system scheduling, network latency, etc. Another aspect is that sequential debugging methods, like breakpoint-to-breakpoint execution, cannot be applied to parallel programs.

Early debugging methods in parallel systems relied on the so-called "monitor & replay" approach [4]. In the first monitoring (or recording) phase, the monitoring tool collects as much data about the parallel program as required to reproduce the run in a deterministic way in the second replay phase. In this approach, replay is driven by the previously gathered program information. This approach does provide a solution for the debugging of parallel programs, but a new problem, the probe-effect is introduced. This means that the monitoring of the parallel program affects timings within it. Although one can mitigate the impact of the probe-effect by reducing the amount of collected information during monitoring, the effect cannot be completely eliminated itself.

Another approach for parallel program debugging is the so-called "control & replay" method (or active control), in which we make use of systematically generated test cases to exhaustively and completely test every possible timing condition in the parallel program. Replay is not performed according to collected data but according to the generated test cases. The most important part of this debugging approach is to find a suitable solution that can generate these test cases [4,18].

The previously mentioned macrostep-based debugging methodology utilises this "control & replay" approach by introducing the concept of collective breakpoints and macrosteps. An early implementation of this debugging method was DIWIDE (DIstributed WIndows DEbugger) [4]. Macrostep-based debugging builds on the following concepts: local breakpoints, collective breakpoints, macrosteps, the execution tree and meta-breakpoints.

Local breakpoints are implemented on the process level, and a process is halted when it hit a local breakpoint. A collective breakpoint is a set of local breakpoints, preferably covering each process. If a collective breakpoint contains local breakpoints from every process, then it is a complete one, otherwise it must be considered partial. If a collective breakpoint contains local breakpoints for all possible alternative paths for each process, then the collective breakpoint is strongly complete. A macrostep is the executed code region between two collective breakpoints. The original macrostep-debugging concept distinguishes pure and compound macrosteps, meaning that if communication-related code is found only as its last element, then the macrostep is pure, otherwise it is compound. By using strongly complete, pure collective breakpoints, the traditional breakpoint-to-breakpoint debugging methodology of sequential programs can be extended to parallel programs. While in sequential programs debugging is done in a step-by-step manner, in parallel programs, it can be achieved macrostep-by-macrostep [4].

One order of consecutive collective breakpoint hits is an execution path. The execution tree contains all execution paths, all possible timing conditions in the parallel program. It is built up of collective breakpoints and macrosteps, collective breakpoints being the nodes and macrosteps being the directed edges. The execution tree starts with the root node. In the original macrostep concept there are 3 distinct type of nodes: fork, alternative and deterministic. Deterministic nodes do not create new execution paths, however at alternative and fork nodes, it is possible to force the parallel program towards different execution paths. Breakpoints can be placed in the execution tree as well, in which case they are called meta-breakpoints, essentially meaning that the parallel program is steered to a specified node in the execution tree. Using an appropriate debugging tool that can generate suitable collective breakpoints [4], one can achieve the complete [18] and exhaustive testing [19] of parallel programs.

3.2. Macrostep-Based Debugging in Cloud Orchestration

IaaS systems can contain up to dozens or even thousands of VMs, and contextualisation is usually done in parallel to speed up deployment. It might be a challenging task to locate and analyse errors due to the inherently non-deterministic nature of cloud resources. Contextualisation processes may depend on each other, VMs differing in configuration, the actual load on physical resources used by the infrastructure (physical CPU, memory, storage, etc.), among others, can all influence timings during the deployment of a given infrastructure (or platform). Because of this non-deterministic behaviour, it is not unusual that erroneous situations cannot be reproduced reliably.

Traditional parallel systems and IaaS infrastructure deployments show many similarities, like potentially dependent processes running concurrently, errors related to wrong timing and problematic error reproduction. Thus it seems straightforward to apply the original macrostep debugging methodology (see Figure 1) to cloud infrastructure deployment. Similarly to the original concept, processes are running concurrently, which, in the case of IaaS systems, are the contextualisation processes of the VMs. By placing local breakpoints in each virtual machine's contextualisation process (see VM1 to VM4 in Figure 1), the debugger can effectively suspend them until an appropriate control signal is received. Then local breakpoints can be organised into collective breakpoints, each collective breakpoint (see M1 to M5 in Figure 1) containing one local breakpoint from every contextualisation process.

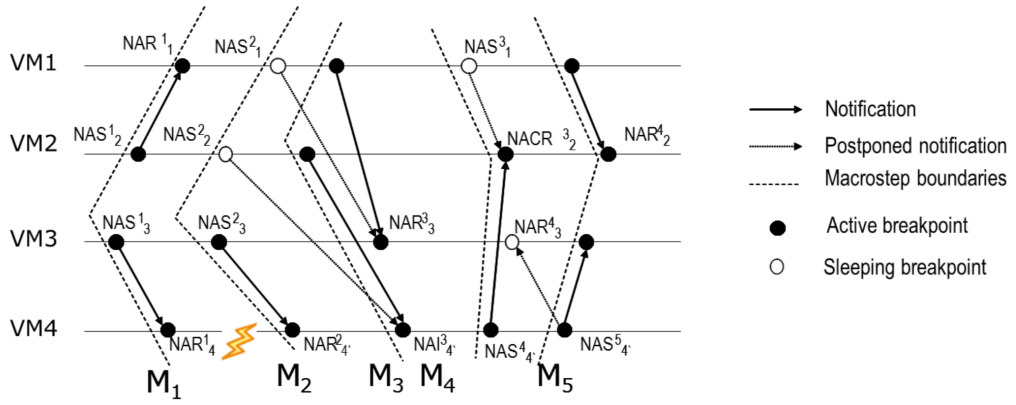

Figure 1. Macrostep concept in IaaS cloud infrastructures.

Reaching a collective breakpoint in this context essentially means the overall deployment of the infrastructure is temporarily halted, forming a consistent global cut (or state). At this point, each contextualisation process is waiting to proceed. If the debugger may choose from multiple contextualisation processes to continue, then the collective breakpoint is an alternative collective breakpoint. The next collective breakpoint can be reached by permitting one of the waiting processes to progress. If there is only one process waiting, then the collective breakpoint is deterministic.

The execution tree contains all execution paths, all possible timings that can occur during an infrastructure's deployment. The execution tree's root root node is the set of the first local breakpoints from each contextualisation process and is the first collective breakpoint for all execution paths. Nodes in the execution tree are connected by macrosteps, meaning that a macrostep is the executed contextualisation code between two collective breakpoints. In this way, IaaS system deployment can be carried out macrostep-by-macrostep, going from one collective breakpoint to another.

3.3. Example for Orchestrated Infrastructure Deployment

An example (see Figure 2) for demonstrating the benefits of macrostep-based debugging would be an infrastructure containing two database-clients ("cl1" and "cl2") and one database-server ("srv"). Normally, timings can be correct in on deployment, and clients would be able to connect to the already existing database, but in another deployment, it is

possible that the database-server is far behind in contextualisation to handle any read-write request coming from the clients. With the macrostep debugging method, the debugger can systematically test each timing condition and find sufficient ones where clients are able to connect to an already existing database.

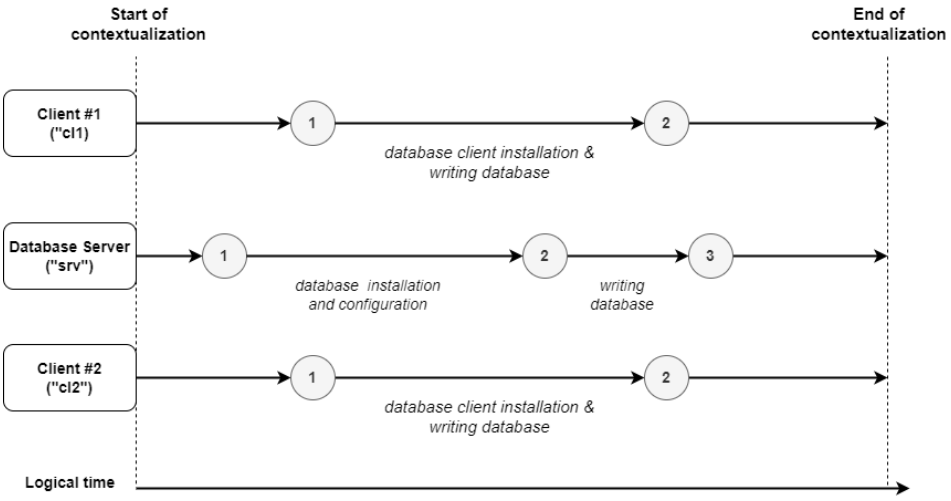

Figure 2. An example infrastructure containing three virtual machines.

A synthetic execution tree has been generated manually for illustration purposes in Figure 3 to understand the nature of an execution tree using this example infrastructure. It has the collection of all possible execution paths combined into a tree at the point of branches. In this simple tree, the infrastructure has two clients and one server. The clients contain two breakpoints, while the server contains four breakpoints. Each node represents a collective breakpoint and an n-tuple (in this case, the triplet) in the rectangles show which breakpoints the virtual machines are blocked in the order of client1, client2 and server. On the top, marked by a rectangle with thick line is considered as a root node. That is the initial point of every virtual machine waiting on their first breakpoints ("1-1-1"), i.e., on the first collective breakpoint, which can also be seen in Figure 1 denoted by M1. The final status for the entire infrastructure arrives when they are blocked on their final breakpoints, which is marked with the triplet of "2-2-3" on each leaves drawn by thin line rectangle back in Figure 3. Between the root and final collective breakpoints, we considered two more types of collective breakpoints in this example. They are the deterministic collective breakpoints marked by the dotted line and the alternative collective breakpoints marked by dashed line. The former means that there is only one path leading to the next collective breakpoint, while the latter represents a branch, i.e., more than one paths lead from that node to the next ones.

Looking at this execution tree, one may recognise that several nodes in the tree are identical regarding their status, i.e., these collective breakpoints represent the same consistent global status. For example, in Figure 3 the nodes "2-2-1" or "2-1-2" can be found more than once since these states can be reached through different execution paths. The most extreme example is the collective breakpoint is the one denoted by "2-2-3", which is the final one since every path leads to this collective breakpoint representing the final state. By merging the common nodes in this tree, a new way of visualisation of this graph could be created, where the graph starts with one root node and ends with one final node. This conversion is now skipped in this visualisation and the tree is kept just for the sake of understandability and for the sake of not losing information on which path a given node has been reached.

Using an appropriate tool, the macrostep-based debugging of orchestrated cloud infrastructures can be achieved. With this method, developers can systematically and exhaustively test every possible timing condition that can arise during the infrastructure's deployment.

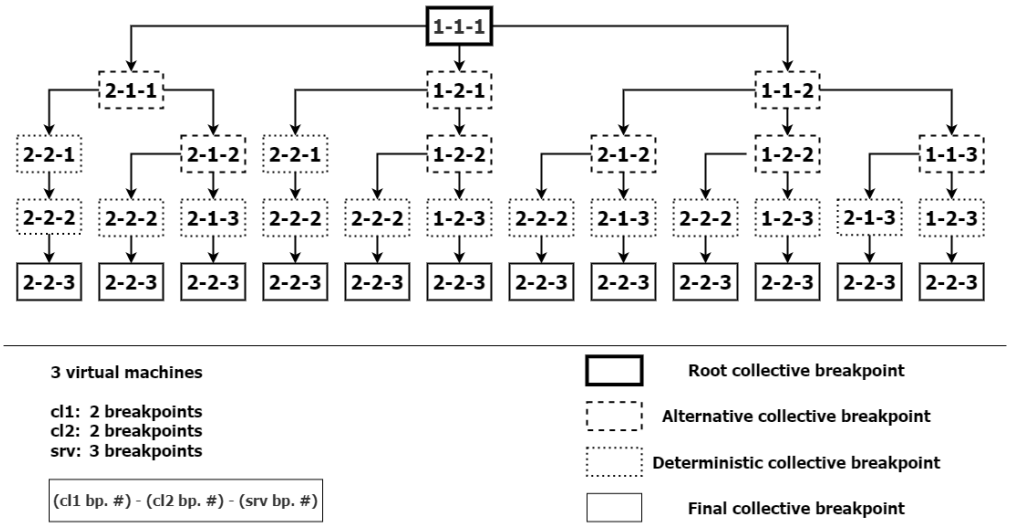

Figure 3. Complete execution tree of example infrastructure with 3 VMs and 7 local breakpoints.

4. Prototype: Integrated Debugger for Cloud Orchestrator Tool

In this section, we introduce an experimental system that utilises Occopus [7] for orchestrating cloud resources, Cloud-init [20] for VM contextualisation, Neo4j [21] graph database for storing and visualising the execution tree and the Macrostep Controller to coordinate the operation of the previous components. Figure 4 shows these components and their interactions.

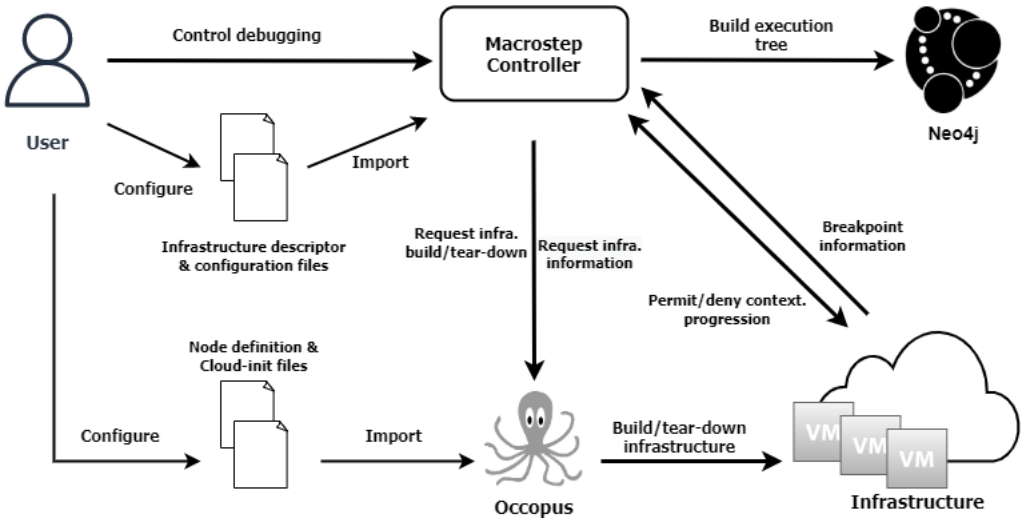

Figure 4. High-level architecture of the experimental prototype of integrated debugger system.

4.1. Components

Occopus (see Figure 4), developed by our laboratory in SZTAKI, is a lightweight cloud-orchestrating tool that supports a wide variety of cloud providers, including public (e.g., Amazon Web Services [22], Azure [10]), private, community and hybrid ones as well. It supports infrastructure management during its full life cycle, including deployment, monitoring, scaling and shutdown.

To create an infrastructure, Occopus uses a so-called infrastructure descriptor that describes the infrastructure components at a higher level as well as dependencies among them. Infrastructure descriptors contain additional information, such as scaling parameters and user-defined variables. While the infrastructure descriptors describe what to create, node definition (one level below) describes how to create the nodes. An infra descriptor may refer to several node definitions that specify the nodes' various features realised by

virtual machines. These features are image id, flavor, network settings, etc. and many more cloud-related settings, as well as the way how the contextualisation should happen. Contextualisation is realised by cloud-init to perform changes on the newly created virtual machine. Furthermore, Occopus is able to utilise configuration manager (e.g., Chef [23], Puppet [24]) tools and supports various health-checking procedures (e.g., ping, port, URL, etc.).

Occopus has a REST API interface that makes it easier to build, maintain, scale and destroy infrastructures or nodes remotely. Alternatively, Occopus itself can be used as a CLI tool if needed. Upon the creation of an infrastructure, Occopus allocates unique identifiers both for the infrastructure and for the nodes comprising it. Occopus was chosen since it is open-source, easy to use and configure, has a rich feature-set and is compatible with most cloud platforms [6].

Cloud-init (see Figure 4) is a de facto industry standard tool for VM contextualisation, supported by a large set of cloud providers and major Unix distributions. It enables infrastructure developers to customise VMs to their requirements, including users, files, permissions, network settings, configuration managers, packages, disks and partitions. A Cloud-init script can contain different sections, where one of them is *runcmd*, that can be used to run user-defined, operating system specific code. Custom shell scripts can be placed in the *write_files* section and can be executed in the *runcmd* section if needed. Cloud-init is used widely in the industry, can be easily configured and is a versatile tool [20]. We will rely on its features when realising the macrostep technique.

Neo4j (see Figure 4) is a robust, powerful graph data platform that is suitable to store, handle and visualise large graphs (e.g., execution trees for macrostep). Since the number of local breakpoints and execution paths may increase dramatically in certain situations during macrostep execution, it was necessary to select a solution that could handle large sets of nodes (collective breakpoints), links and their related information. Neo4j can be used as a dedicated local/remote service or can be utilised as a hosted service in the macrostep architecture.

The Macrostep Controller (see Figure 4) is the heart of the system responsible for coordinating the entire debugging session, like performing manual/automatic debugging with active control or replaying. The controller instructs the orchestrator (Occopus) to deploy or destroy the entire infrastructure, decides when to execute the deployment of a particular virtual machine, builds the execution graph/tree by instructing the graph storage (Neo4j) and provides a user interface to receive the commands.

In order to understand the operation of the Macrostep Controller, we introduce the main components of this entity. Its internal architecture can be seen in Figure 5.

The heart of the Macrostep Controller is its core controller which performs the decision making and keeps the entire system alive, instructs all the other components. The request handler receives incoming user commands, i.e., loading infrastructure, setting breakpoints, continuing execution, etc. All these requests (commands) goes to the core controller, which then instructs the orchestrator handler to create/destroy infrastructure instance, i.e., to control the deployment of the infrastructure with its internal Occopus plugin in the current prototype. The breakpoint register is responsible for communicating to the virtual machines, i.e., receiving notifications on breakpoints being hit and instructing the VMs to continue their execution of deployment. The Macrostep Controller includes an internal database (implemented by SQLite [25] in the current prototype) for storing internal status of the debugging sessions. The execution tree handler performs the execution tree related read/write operations such as the creation or update of root and other nodes in the tree built by the execution paths. In the current implementation, a Neo4j graph database is used for this purpose.

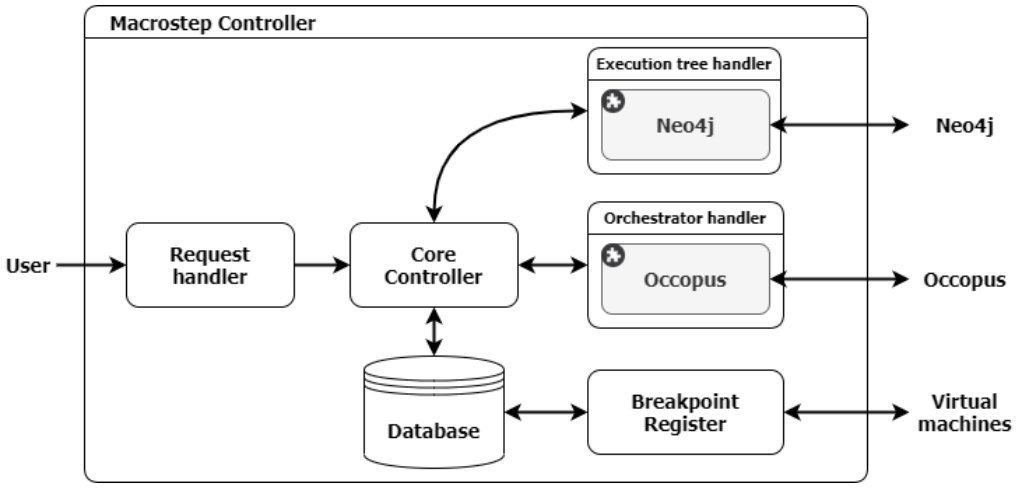

Figure 5. Internal architecture of Macrostep Controller.

4.2. Operation Mechanisms

In order to execute and debug the deployment of an infrastructure, a complete set of descriptors (infrastructure descriptor, node definition and cloud-init) is needed for the Occopus cloud orchestrator. We assume that Occopus is installed and properly configured and the cloud credentials are also configured for Occopus to access the cloud API through which the infrastructure is about to be deployed. Once we have a ready-to-use infrastructure and the Occopus orchestrator, the macrostep debugging may happen.

The Macrostep Controller requires an infrastructure-independent configuration file that contains the necessary information of the external components, e.g., endpoint of the Neo4j graph visualiser and endpoint of the Occopus cloud orchestrator. This information is read by the core controller, passed to the Execution tree handler (Neo4j plugin) and to the Orchestrator handler (Occopus plugin) to find their external tools. The breakpoint register is a component that keeps the connection with the virtual machines. On one side, it receives notifications on the breakpoint hits and registers them in the database, on the other side, it notifies virtual machines about the continuation of the execution whenever this decision is registered in the database by the core controller.

Since VMs contextualisation is implemented by the cloud-init file, we are instrumenting them to realise breakpoints. The current implementation does not support the automatic insertion of breakpoints, therefore manual preparation of the cloud-init files are needed. The macrostep local breakpoint is already realised by a method under the *write_files* section, so its invocation can be inserted at the boundary of the service installations. This special shell script (called *breakpoint script*) will temporarily halt contextualisation, collect information about the virtual machine's inner state, send it to the breakpoint register and periodically asks (pooling) for permission to continue contextualisation.

Arguments can be passed to the breakpoint script to better describe operations previously executed during contextualisation. The last call is expected to contain the argument 'last' or 'last_bp' to indicate for the Macrostep Controller that no more breakpoint exists and the deployment is finished. The breakpoint script creates a JSON [26] structure containing necessary keys and values. The unique ID of the VM along with the name of the infrastructure instance and the virtual machine are part of the JSON descriptor by default for identification. The JSON descriptor can also be extended with user-defined key-value pairs as well. When the script is invoked, it establishes a connection towards the breakpoint register and posts the JSON structure.

When the local breakpoints are set, the debug session can be started by launching the Macrostep Controller with the infrastructure name and its descriptor file as arguments. First, the Macrostep Controller instructs Occopus to create an infrastructure instance which as a return replies with the created infrastructure instance's unique ID.

Occopus starts building the infrastructure and the VMs begin their contextualisation according to the Cloud-init files. When a VM hits a local breakpoint during contextualisation the previously described breakpoint script is executed, halting contextualisation until permission is granted. The Macrostep Controller waits until the infrastructure reaches *root state* which means that every virtual machine has reached its first breakpoint. This is considered as the root collective breakpoint, i.e., the root node in the execution tree. At this point, the entire infrastructure deployment is halted, and every VM is waiting for permission to continue. The Macrostep Controller registers a root collective breakpoint for the infrastructure in the Neo4j database.

Depending on the type of debug session, either the user or the Macrostep Controller issues permission to one of the waiting VMs to continue its contextualisation. The selected VM continues its deployment, reaches/hits the next breakpoint, the breakpoint script is executed again and the VM blocks again. The infrastructure at this point has reached another collective breakpoint.

As the contextualisation of the virtual machines are developing, the execution tree is built by adding more and more nodes. Each node in the execution tree stores several properties, which are as follows:

1. name of the infrastructure
2. collective breakpoint ID
3. previous collective breakpoint ID
4. process states
5. collected data
6. node type
7. instance IDs
8. exhausted flag

Collective breakpoints (2) in the tree are associated with a unique ID upon creation. Additionally, each node refers to its predecessor breakpoint (3). The Macrostep Controller uses these IDs to keep track of the infrastructure instance's current position in the execution tree.

The process states (4) property defines the current local breakpoint number for each process where they are blocked at the moment. It is stored by using the process names as keys and the values are the list of breakpoint numbers for each instance of the process named in the key. For example, the property with this value {"client": [1,2], "server": [2]} means the client has two instances where the first instance has been blocked at its breakpoint number one, while the other instance is blocked at its breakpoint number 2. The server has one instance blocked at breakpoint number 2. The next property in the list, called *collected data* (5) stores every information collected and received from the virtual machines at the time of hitting the breakpoint.

The Macrostep Controller is responsible for determining the type of a node (6) in the execution tree: "alternative" if there are more than one unfinished contextualisation processes or "deterministic" if there is only one such process. In case a collective breakpoint is the last collective breakpoint of an execution path, meaning there are no viable contextualisation processes to continue, then the collective breakpoint will be flagged as "*final*".

Infrastructure instances that traversed a node in the execution tree will be listed in the node's instance IDs (7) properties. The exhausted flag (8) is primarily used for alternative breakpoints, indicating the case when every execution path starting from that node has been traversed accordingly if the root's exhausted flag is set to true.

Whenever a collective breakpoint is hit, the Macrostep Controller performs the creation and linking of appropriate nodes in the execution tree as needed. After a collective breakpoint has been created in the Neo4j database, a continue command is issued to one of the waiting, unfinished virtual machines either by the user or the Macrostep Controller depending on the working mode.

This whole process is continued until all VMs have finished their contextualisation. At each collective breakpoint the Macrostep Controller checks the infrastructure's execution tree, updating or creating and attaching new collective breakpoints if necessary. The Macrostep Controller eventually detects when infrastructure deployment is over since every VM has finished its contextualisation process, and then it instructs Occopus to destroy the infrastructure instances. This overall procedure represents one execution path in the execution tree, and the execution tree is built incrementally, i.e., another execution path can be traversed in the next turn.

5. Major Operations Modes for Debugging Purposes

5.1. Manual Debugging Mode

Generally, a debug session, either manual or automatic or replay, starts with the creation of an infrastructure instance by Occopus, and the Macrostep Controller waits until all the infrastructure instances reach the root state. After the root state has been reached, the Macrostep Controller registers the collective root breakpoint of the infrastructure in the Neo4j database.

During manual debugging the user is responsible for choosing among the virtual machines of the infrastructure. At each collective breakpoint, the user is prompted for selection, after which permission is granted to the chosen virtual machine and it continues its contextualisation process. Eventually, the infrastructure instance will reach a new collective breakpoint and the Macrostep Controller checks if such a collective breakpoint exists in the execution tree or not. If not, then the Macrostep Controller registers a collective breakpoint. At each collective breakpoint, the Macrostep Controller checks if every process has finished its contextualisation. If not, then the user is prompted again to choose from the available virtual machines. This is repeated until all virtual machines have finished their contextualisation, after which the Macrostep Controller instructs Occopus to destroy the infrastructure instance.

The steps described above represent the manual debugging of one execution path. The user can start a new manual debug session and explore another execution path.

5.2. Automatic Debugging Mode

During automatic debugging, the Macrostep Controller itself decides which virtual machine will continue its deployment at each consistent global state. After an automatic debug session is initiated, the Macrostep Controller issues the creation of an infrastructure instance and it waits until the instance reaches the root state.

To select a virtual machine, the Macrostep Controller creates an ordered list of the VMs using alphabetical ordering by (node) name and by (node) ID. From this list, the Macrostep Controller picks the first virtual machine that has not yet finished its deployment and continues it. This selection policy is applied at each consistent global state until the infrastructure instance has finished its deployment, after which the Macrostep Controller instructs Occopus to terminate the instance.

At this point, the Macrostep Controller backtracks in the execution tree to the first non-exhausted, alternative collective breakpoint and initiates a replay session (see the following subsection) targeting to reach this collective breakpoint. The new infrastructure instance will eventually reach the targeted collective breakpoint and the Macrostep Controller continues on a new, non-explored execution path, selecting the appropriate virtual machine. An alternative collective breakpoint will be flagged as exhausted if every alternative path from that collective breakpoint is explored.

Automatic debugging essentially means the depth-first traversal of the execution tree, which continues until the collective root breakpoint itself is flagged as exhausted.

5.3. Replay Mode

The replay mechanism is implemented through meta-breakpoints, meaning that the user defines a collective breakpoint ID at the start of the session. The aim of the Macrostep

Controller is to coordinate the execution of virtual machines breakpoint-by-breakpoint to reach the collective breakpoint defined by the user. Initially, it checks if the collective breakpoint exists in the infrastructure's execution tree, and then the Macrostep Controller instructs Occopus to create the infrastructure instance. During the coordination of the execution of VMs, the execution path is followed towards the target one. At each collective breakpoint hit, it is checked if the targeted collective breakpoint has been reached already. If not, then the Macrostep Controller calculates which virtual machine's contextualisation process is needed to be continued to reach the next consistent global state leading towards the target collective breakpoint.

6. Visualization and Query Functionalities for the Execution Tree

As mentioned in Section 4, the current prototype integrates the Neo4j graph database and visualiser software. In the macrostep prototype, we utilise its capability to store and visualise execution trees and their related information. For visualisation purposes, Neo4j includes a web interface accessible by web-browsers, which can be used to run database queries and display their results. Neo4j uses the Cypher Query Language [27]. Users can inspect execution trees created during the macrostep debug sessions by running Cypher queries through this web interface.

In our example in Figure 6, in the upper left corner such CQL query and its result (on the left) can be seen.

The query (see Listing 1) has been constructed to display the execution tree generated for the infrastructure called *app_mariadb*. On the left side, the screenshot of the execution tree is in Figure 6. The graph is built by nodes (denoted by circles) representing collective breakpoints and by edges (denoted by arrows) representing links between two consecutive collective breakpoints. In the example, the query resulted in a graph where thirty-five nodes and thirty-four edges satisfied the conditions.

Listing 1: query the tree

```
match (n:Collective_BP {app_name: "app_mariadb"})
return n
```

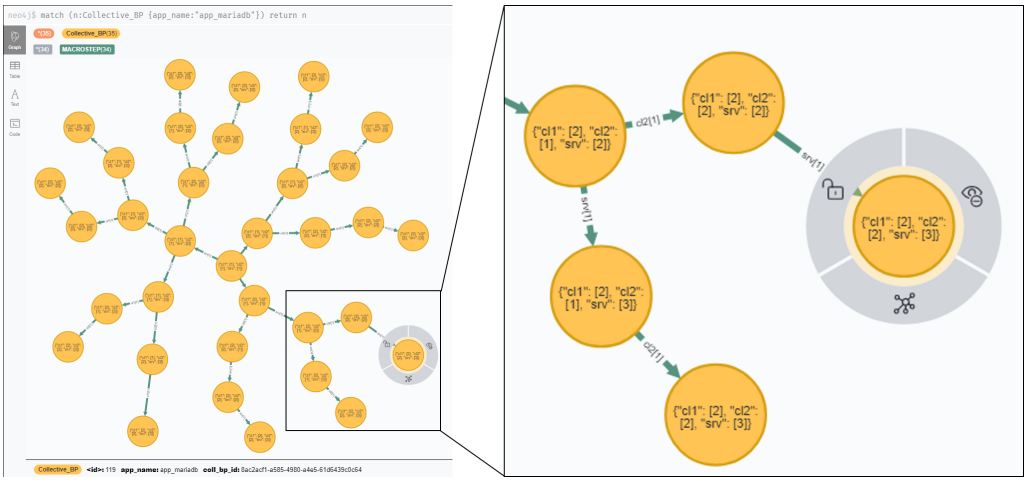

Figure 6. An execution tree and its subtree (right side) represented in Neo4j.

Beyond showing the result as a graph, Neo4j offers three more formats for displaying the results. They are (i) Table, (ii) Text and (iii) Code. Users can choose between them on the left-hand side graphical menu. For our use case, the graph display mode is the most comfortable option and as such the nodes of the tree can be rearranged on the canvas to make it easier for overviewing.

Querying larger execution trees may result in nodes displayed with small-sized text on them becoming blurry. Users can zoom in and zoom out to mitigate this, while navigation up/down or left/right can be achieved with a "click-and-drag" style. Selecting a node or edge will highlight it and display its properties and values on the status bar of the page. Scrolling can be used to navigate up/down between properties.

Selected nodes are highlighted by a grey circle around the node that is divided into three slices (see the right side of Figure 6). The lock symbol can be used to unlock the node from its current position in case we previously moved it, which will result in the node "floating" freely. The eye symbol is used to dismiss it from the current display, while the graph symbol can be used to expand or collapse child relationships.

The web interface of Neo4j offers customisation options as well. For example, changing the colour and size of collective breakpoints, as well as adding a caption that is displayed on the nodes. The latter one is utilised by the macrostep debugger to add collective breakpoint related information on top of the nodes, as it is shown on the right side of Figure 6.

The code (see Listing 2) lists the name of the nodes in quotes (e.g., "cl1") and the breakpoint in squared brackets indicates at which local breakpoint is the node (i.e., the virtual machine) blocked currently. For more detailed information, the status bar enables to open up a textual description showing all the details associated and stored for a given collective breakpoint. Finally, the caption on the edges also shows information related to the changes that happened during the execution between the source collective breakpoint and the collective target breakpoint.

Listing 2: collective breakpoint properties

```
{"cl1": [2], "cl2":[2], "srv":[3]}
```

The code example (see Listing 3) means that the first instance named "srv" was executing until it reached its next local breakpoint, i.e., the next collective breakpoint. These codes help the developer to assign a certain collective breakpoint with the location in the deployment script when looking for unexpected behaviour during the deployment orchestration.

Listing 3: edge properties

```
srv[1]
```

7. Benchmarking and Experiences

In order to collect experiences about the behaviour of the Macrostep Controller as well as about the macrostep methodology in the deployment phase of cloud infrastructures, an illustrative and scalable use case has been selected, and we performed several measurements on it. The client-server application has been identified as a proper use case as it is simple enough but contains dependencies and scalable components (VMs) at the same time. The test configuration consisted of Occopus v1.8, Amazon's EC2 service with t2.micro instances, Ubuntu 18.04 (ami-0e342d72b12109f91) operating system and MariaDB version 10.1.48 as the database server and database client.

The application contains one server and an arbitrary number of clients depending on the existence of the server. The server in our use case is a database server, while the clients can read/write the database on the server. In the server contextualisation, we inserted 3 breakpoints, while the client contains only 2 breakpoints at the beginning and ending of its contextualisation.

The first set of measurements aimed to investigate the cost of applying the macrostep methodology on our use case, i.e., how much time it is required for the controller to discover the entire state space of an application depending on its size. The correlation between the time required to traverse all the execution paths of the application and between its size can be seen in Figure 7.

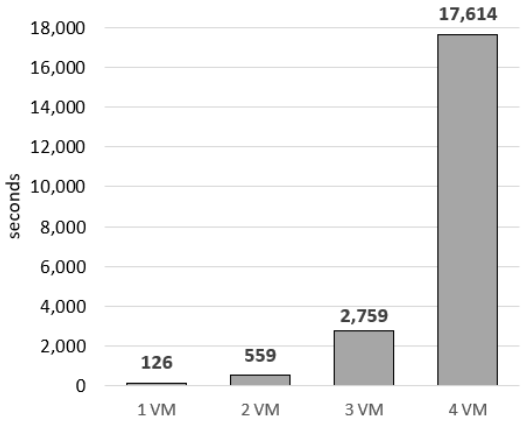

Figure 7. Total time in seconds to traverse the execution tree during automatic debugging.

There are four different setups we measured and investigated: the server without any client (1VM), the server with one (2VM), two (3VM) or three (4VM) clients. The total number of execution paths are 1, 3, 12, 60 for each setup, respectively. The columns indicate the overall time required to traverse the entire execution tree of the particular setup. The result shows that with the growing number of virtual machines, the overall execution time increases exponentially. This is in line with our expectation since upscaling increases the alternate paths at each branch. Even with this simple use case, it can be demonstrated how quick the complexity increases with the growing number of virtual machines.

Traversing the entire execution tree requires a significant amount of time even for a simple infrastructure. However, with the help of some optimisation (see Section 8), the full-state space searching could be driven to scan only the execution paths which are of interest. To see the cost of traversing one execution path, we investigated the previous measurements and extracted the timestamp of launching the virtual machine, the start and finish timestamps of the contextualisation. Based on the timestamps, we calculated the period for booting all virtual machines and hitting their first breakpoints at the beginning of the cloud-init process, i.e., the time interval necessary for traversing the *root node* of the execution tree (see "Root" in Figure 8) and for macrosteping through an execution path ("Macrosteps" in Figure 8). Based on the information, the average time spent for running an execution path is also shown in the figure for each of the four setups.

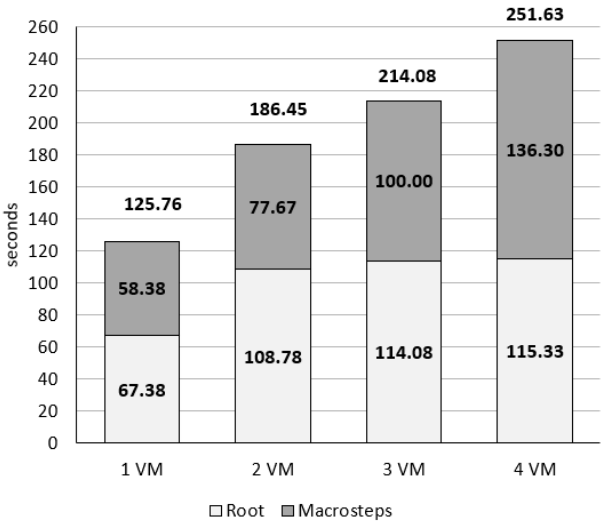

Figure 8. Average time in seconds to traverse one execution path with 1 to 4 VMs and macrosteps.

Furthermore, in Figure 8 various situations can be observed. Root time increases once new dependency appears among the virtual machines, such as between the first client and

the server. More clients do not increase the root time since they are executed in parallel during the experiment. With the growing number of virtual machines, the time required for running an execution path increases nearly linear since the number of steps in an execution path increases.

Digging even more into the details, we tried to investigate how the execution path is combined by the time of the different steps. The boundaries of the macrosteps are visualised in Figure 9 leveraging on the collected timestamps. There are 12 execution paths altogether in a 3VM setup and each path contains 4 macrosteps. During the contextualisation, the most time-consuming activity is installing of the database (typically 50 s), while other steps consume much less time. Depending on the order of steps, the server installation time appears in different macrosteps throughout the execution path.

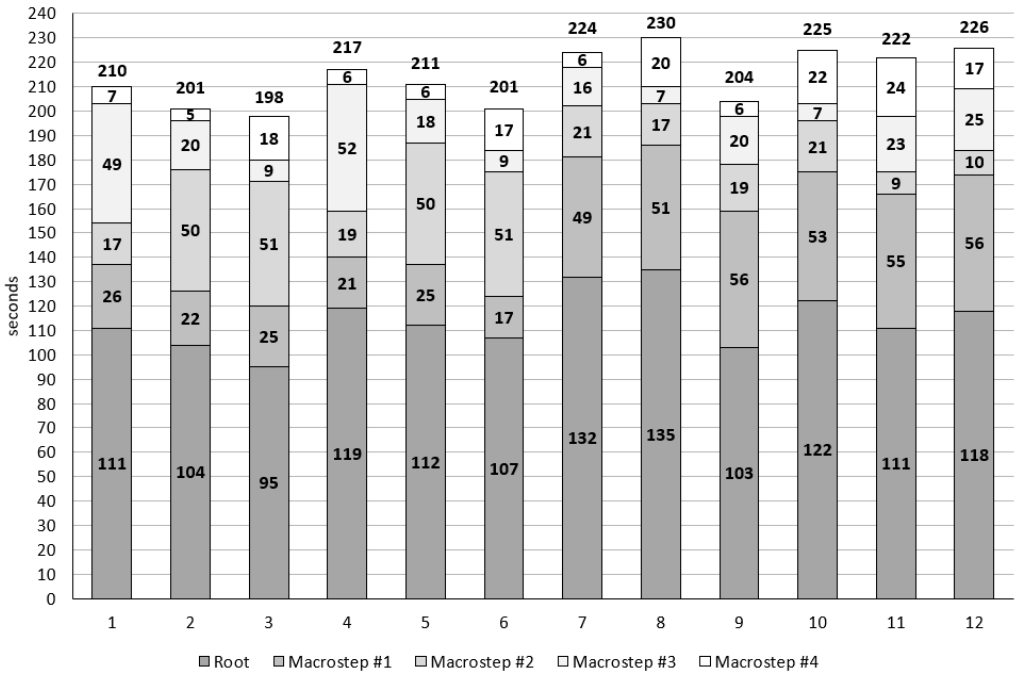

Figure 9. Macrostep periods in each execution path of the 3VM setup.

The previous experiment showed the average execution time of each macrostep in each execution paths, however these paths may contain invalid ones (considering the situation when a client is deployed before the database installation). The aim of the macrostep debugger is to find these invalid/faulty ones among the entire state space. To estimate the chance to find execution paths that are not covered during normal execution, we performed the fourth experiment. In this experiment, we inactivated the breakpoints and collected information only about the order of their hits during so-called free-runs where no active control over the execution happened. By doing this, we managed to identify the number of execution paths covered during normal execution.

The outcome of experiment four is visualised in Figure 10. In the horizontal axis, the executions are enlisted each after another, while the vertical axis shows the average number of distinct traversed execution paths until that execution. In this experiment the maximum number of distinct paths is 12 as we ran the 3VM setup. As the executions were repeated and all-together 24 runs were performed, the number of execution paths discovered increases linearly until the maximum is found in case of macrostep based execution, i.e., with active control. In case of free-run (no macrostep control, only monitoring) approximately half of the paths have been covered during the 24 executions. The number of runs could be increased, however the tendency shows that the cost increases dramatically to find one more additional path in the execution tree. This experiment clearly shows the advantage of macrostep against the random execution for finding execution paths of an application.

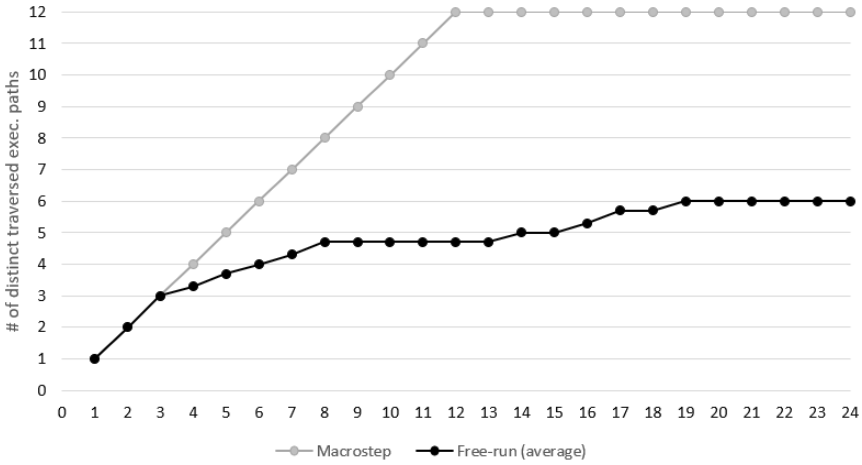

Figure 10. Overall number of covered execution paths during automatic debugging and free-running.

8. Conclusions and Future Work

The paper introduces the first steps of our work on applying the macrostep debugging technique for cloud orchestration. The idea is based on the parallel and concurrent way of establishing and deploying services when contextualising virtual machines. We successfully identified the theoretical background including processes, breakpoints, collective breakpoints, and execution trees in the context of cloud orchestration. In order to validate our work, a prototype is developed relying on the Occopus orchestrator and the Neo4j graph database tools. We introduced the components and operation of the prototype as well as the manual-/auto-debugging, replaying, visualisation and query functionalities of the tool. The validation of the experimental framework has been started with use case and measurements with numerical results. Our mechanism is able to handle the non-deterministic behaviour of the cloud environment in terms of the unpredictable relative speed of resources (including virtual machines, networks, and I/O devices). Besides the reproducible errors, the described mechanism allow us to explore the state space systematically and help in the detection of more possible erroneous situations with higher coverage as illustrated in this paper.

The impact of our results is expected to be wide since cloud computing has become a cornerstone of a large variety of research and innovation activities: the European Open Science Cloud (EOSC) [28] initiative and the Hungarian ELKH Cloud [29] are two prominent examples. The Occopus orchestration tool is available and supported by the major open-source clouds, and also as a part of commercially supported MiCADO [30] framework. Since commercial clouds are also supported by Occopus, the possible impact of our results is even wider, e.g., Occopus has been applied in the manufacturing sector as well [31]. Another way of impact is related to the reference architecture concept [32] where the reliability and portability might be improved of such complex architectures by using our approach.

Our research is the first step towards a novel debugging framework. Regarding the future work, we plan to automate the instrumentation, i.e., the insertion of the local breakpoints into the cloud-init files of the nodes. We would also like to improve the prototype further and make it more universal by handling and supporting other cloud orchestrators (such as Terraform). More use cases must be evaluated to start categorising the possible errors. The planned parallel version of execution tree traversing will significantly improve the usability of the debugger tool in complex use cases.

Moreover, the automatic correctness evaluation of consistent global states [19] at each macrostep is in our research plan together with the further elaboration of modelling and steering mechanisms [18] towards possible suspicious or erroneous situations in the generated execution tree. This smart functionality will rely on graph-based machine learning. Later, the maintenance phase of cloud infrastructures is to be studied. Finally,

the formerly defined formal background for macrostep has to be adapted [33] in the new scenario along with experimental large scale validation.

Author Contributions: Conceptualization, J.K.; Formal analysis, R.L.; Investigation, J.K. and R.L.; Methodology, J.K. and R.L.; Software, B.L. and M.E.; Supervision, J.K. and R.L.; Validation, R.L.; Visualization, B.L. and M.E.; Writing—original draft, B.L.; Writing—review & editing, M.E., J.K. and R.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was partially funded by the National Research, Development and Innovation Office (NKFIH) under OTKA Grant Agreement No. K 132838, the Eötvös Loránd Research Network Secretariat under grant agreement no. KÖ-40/2020, and the ÚNKP-20-5 New National Excellence Program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation Fund. The presented work of R. Lovas was also supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences. M. Emődi thankfully acknowledges the support of the Doctoral School of Applied Informatics and Applied Mathematics, Óbuda University.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Bhardwaj, S.; Jain, L.; Jain, S. Cloud computing: A study of infrastructure as a service (IaaS). *Int. J. Eng. Inf. Technol.* **2010**, *2*, 60–63.
2. Caballer, M.; Blanquer, I.; Molto, G.; de Alfonso, C. Dynamic management of virtual infrastructures. *J. Grid Comput.* **2015**, *13*, 53–70. [CrossRef]
3. Dukaric, R.; Juric, M.B. Towards a unified taxonomy and architecture of cloud frameworks. *Future Gener. Comput. Syst.* **2013**, *29*, 1196–1210. [CrossRef]
4. Kacsuk, P.; Lovas, R.; Kovács, J. Systematic Debugging of Parallel Programs in DIWIDE Based on Collective Breakpoints and Macrosteps. In *Euro-Par'99 Parallel Processing*; Springer: Berlin/Heidelberg, Germany, 1999; Voume 1685, pp. 90–97.
5. Kacsuk, P.; Dozsa, G.; Kovacs, J.; Lovas, R.; Podhorszki, N.; Balaton, Z.; Gombas, G. P-GRADE: A Grid Programming Environment. *J. Grid Comput.* **2003**, *1*, 171–197. [CrossRef]
6. Kovács, J.; Kacsuk, P. Occopus: A Multi-Cloud Orchestrator to Deploy and Manage Complex Scientific Infrastructures. *J. Grid Comput.* **2018**, *16*, 19–37. [CrossRef]
7. Occopus. Available online: https://occopus.readthedocs.io/en/latest/ (accessed on 23 October 2021).
8. Zhang, J.; Luan, Z.; Li, W.; Yang, H.; Ni, J.; Huang, Y.; Qian, D. CDebugger: A scalable parallel debugger with dynamic communication topology configuration. In Proceedings of the 2011 International Conference on Cloud and Service Computing, Hong Kong, China, 12–14 December 2011; pp. 228–234.
9. Cai, J.; Fei, J.; Liu, X.P.; Wang, H.; Wu, Y.R.; Zhong, S.Q. Remote Debugging in a Cloud Computing Environment. US Patent 9,244,817, 26 January 2016
10. Microsoft Azure. Available online: https://azure.microsoft.com/ (accessed on 23 October 2021).
11. Sharma, P.; Chatterjee, S.; Sharma, D. CloudView: Enabling tenants to monitor and control their cloud instantiations. In Proceedings of the 2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), Ghent, Belgium, 27–31 May 2013; pp. 443–449.
12. Baek, H.; Srivastava, A.; Van der Merwe, J. Cloudsight: A tenant-oriented transparency framework for cross-layer cloud troubleshooting. In Proceedings of the 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), Madrid, Spain, 14–17 May 2017; pp. 268–273.
13. Cloud Debugger. Available online: https://cloud.google.com/debugger (accessed on 23 October 2021).
14. Smara, M.; Aliouat, M.; Pathan, A.S.K.; Aliouat, Z. Acceptance test for fault detection in component-based cloud computing and systems. *Future Gener. Comput. Syst.* May 2017, *70*, 74–93. [CrossRef]
15. Zhang, P.; Shu, S.; Zhou, M. An online fault detection model and strategies based on SVM-grid in clouds. *IEEE/CAA J. Autom. Sin.* **2018**, *5*, 445–456. [CrossRef]
16. Sheikh Quroush, M.S.; Ovatman, T. A Record/Replay Debugger for Service Development on the Cloud. In *Cloud Computing and Services Science*; Muñoz, V.M., Ferguson, D., Helfert, M., Pahl, C., Eds.; Springer International Publishing: Cham, Switzerland, 2019; pp. 64–76.
17. Goossens, K.; Vermeulen, B.; Steeden, R.V.; Bennebroek, M. Transaction-Based Communication-Centric Debug. In Proceedings of the First International Symposium on Networks-on-Chip (NOCS'07), Princeton, NJ, USA, 7–9 May 2007; pp. 95–106.
18. Lovas, R.; Vécsei, B. Integration of Formal Verification and Debugging Methods in P-GRADE Environment. In *Distributed and Parallel Systems: Cluster and Grid Computing*; Juhász, Z., Kacsuk, P., Kranzlmüller, D., Eds.; Springer: Boston, MA, USA, 2005; pp. 83–92.
19. Kovacs, J.; Kusper, G.; Lovas, R.; Schreiner, W. Integrating Temporal Assertions into a Parallel Debugger. In *Euro-Par 2002 Parallel Processing*; Monien, B., Feldmann, R., Eds.; Springer: Berlin/Heidelberg, Germany, 2002; pp. 113–120.

20. Cloud-Init: The Standard for Customising Cloud Instances. Available online: https://cloud-init.io/ (accessed on 23 October 2021).
21. Webber, J. A programmatic introduction to neo4j. In Proceedings of the the 3rd Annual Conference on Systems, Programming, and Applications, Software for Humanity, Tucson, AZ, USA, 19–26 October 2012; pp. 217–218.
22. Amazon Web Services. Available online: https://aws.amazon.com/ (accessed on 30 October 2021).
23. Luchian, E.; Filip, C.; Rus, A.B.; Ivanciu, I.A.; Dobrota, V. Automation of the infrastructure and services for an openstack deployment using chef tool. In Proceedings of the the 2016 15th RoEduNet Conference: Networking in Education and Research, Bucharest, Romania, 7–9 September 2016; pp. 1–5.
24. Sobeslav, V.; Komarek, A. OpenSource Automation in Cloud Computing. In Proceedings of the 4th International Conference on Computer Engineering and Networks, Shanghai, China, 19–20 July 2014; Wong, W.E., Ed.; Springer International Publishing: Cham, Switzerland, 2015; pp. 805–812.
25. Owens, M. *The Definitive Guide to SQLite*; Apress: Berkeley, CA, USA, 2006.
26. Pezoa, F.; Reutter, J.L.; Suarez, F.; Ugarte, M.; Vrgoč, D. Foundations of JSON schema. In Proceedings of the 25th International Conference on World Wide Web, Montréal, QC, Canada, 11–15 April 2016; pp. 263–273.
27. Francis, N.; Green, A.; Guagliardo, P.; Libkin, L.; Lindaaker, T.; Marsault, V.; Plantikow, S.; Rydberg, M.; Selmer, P.; Taylor, A. Cypher: An evolving query language for property graphs. In Proceedings of the 2018 International Conference on Management of Data, Houston, TX, USA, 10–15 June 2018; pp. 1433–1445.
28. Almeida, A.; Borges, M.; Roque, L. The European open science cloud: A new challenge for Europe. In Proceedings of the 5th International Conference on Technological Ecosystems for Enhancing Multiculturality, Cádiz, Spain, 18–20 October 2017.
29. ELKH Cloud Portal. Available online: https://science-cloud.hu/en (accessed on 9 November 2021).
30. Ullah, A.; Dagdeviren, H.; Ariyattu, R.C.; DesLauriers, J.; Kiss, T.; Bowden, J. MiCADO-Edge: Towards an Application-level Orchestrator for the Cloud-to-Edge Computing Continuum. *J. Grid Comput.* **2021**, *19*, 47. [CrossRef]
31. Taylor, S.J.E.; Anagnostou, A.; Abubakar, N.T.; Kiss, T.; DesLauriers, J.; Terstyanszky, G.; Kacsuk, P.; Kovacs, J.; Kite, S.; Pattison, G.; et al. Innovations in Simulation: Experiences With Cloud-Based Simulation Experimentation. In Proceedings of the 2020 Winter Simulation Conference (WSC), Orlando, FL, USA, 14–18 December 2020; pp. 3164–3175. [CrossRef]
32. Nagy, E.; Lovas, R.; Pintye, I.; Hajnal, A,; Kacsuk, P. Cloud-agnostic architectures for machine learning based on Apache Spark. *Adv. Eng. Softw.* **2021**, *159*, 103029
33. Lovas, R.; Kacsuk, P. Correctness debugging of message passing programs using model verification techniques. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2007; Voume 4757, pp. 335–343.