Continuous Debugging of Microservices

Hong Zhu, Ian Bayley
School of Engineering, Computing and Mathematics
Oxford Brookes University
Oxford OX33 1HX, UK
hzhu@brookes.ac.uk, ibayley@brookes.ac.uk

Hongbo Wang
School of Computer and Communication Engineering
University of Science and Technology Beijing
Beijing 100083, China
foreverwhb@ustb.edu.cn

***Abstract*—Debugging is one of the most difficult tasks during the development of cloud-native applications for the microservices architecture. This paper proposes a continuous debugging facility to support the DevOps continuous development methodology. It has been implemented and integrated into the Integrated DevOps Environment CIDE for microservices written in the agent-oriented programming language CAOPLE. The paper also reports controlled experiments with the debug facility. Experiment data show that the overhead is less than 3% of the execution time on average.**

***Keywords*-Software-as-a-Service; Microservices; Cloud native applications; Integrated DevOps Environment; Debug facility; Continuous debugging**

I. Introduction

Microservices is a software architectural style in which a cloud-native application consists of a large number of services that are distributed over a cluster of computers, running in parallel, and interacting with each other through service requests and responses [1]. These services are small scale, of fine granularity, and each realises one function only. The instances of a service, called agents in the literature and hereafter in this paper, may be created or terminated dynamically in response to changes in demand or the failure of other agents. Systems in the microservices architectural style can therefore achieve elastic scalability, optimal performance and fault tolerance [2]. Furthermore, the system can evolve without interruption to its operation, because instances of obsolete services can be gradually removed and replaced with agents of new services. In this way, the microservices style enables continuous testing (CT), continuous integration (CI) and continuous deployment (CD), all of which are crucial to cloud-native applications [3]. For this reason, they have been widely adopted by industry for cloud-native applications [4].

However, microservices also pose grave challenges to software development and maintenance. In particular, when a vast number of services are running in parallel in a large cluster of computers, it is notoriously difficult to diagnose the causes of failures; see, for example, [5]. One approach to fault localisation is static analysis, which relies on data saved to files during execution such as log entries and analyse them after execution. In this paper, however, we focus instead on *dynamic debugging*. This is the process of investigating a piece of software through controlled executions of the program and focused observations of its dynamic behaviour, in order to find out the cause of an unexpected behaviour, or failure, by locating defects in the program code. It is widely used throughout traditional software development because of its effectiveness. However, according to Levine, it is *"is totally missing from the toolbox of microservice developers"* [5]. This paper proposes such a debugger facility.

The main contributions of this paper are as follows.

First, we introduce the notion of continuous debugging to complement the existing continuous development methodology and analysed the requirements of debugging microservices on debug tools.

Second, we propose a novel debugging facility that satisfies all the requirements of continuous debugging for microservices. It consists of tools to trace the execution of a selected agent(s), to take a state snapshot of the agent(s), and to control the execution of the agent(s). The debug facility can be integrated into a software development and operation environment to support the principle of DevOps. In particular, it enables programmers to directly debug microservices running in a cluster environment to realise the philosophy of "*you develop it, you operate it*".

Third, we demonstrate that the proposed debug facility is feasible by implementing it via modifying the CAVM virtual machine that runs microservice programs and by integrating it into the integrated DevOps environment CIDE.

Finally, we conducte controlled experiments to demonstrate the efficiency of the implementation of the debug facility. Our experiments show that the overhead can be less than 3% increase of execution time compared with an equivalent virtual machine without such a debug facility. We have also demonstrate that the use of the debug facility imposes minimal interference in the execution of the microservices. For the trace operation, the overhead is less than 5% on average and less than 14% in the worst case. The time needed to take a state snapshot is only a few milliseconds, and it is linear in both the number of variables in the microservices, the total volume of data held

in the variables, and the number of messages in the incoming queue. For most microservices, such debug operations will only take a few milliseconds.

The rest of this paper is organised as follows. Section II analyses the requirements for debugging microservices and introduces the notion of continuous debugging. Section III presents a continuous debug facility, which has designed and implemented for Agent-Oriented programming language CAOPLE and integrated to the Integrated DevOps Environment CIDE. Section IV reports the experiments carried out to evaluate the overhead of the debug facility. Section V concludes the paper with a comparison of related work.

II. Debugging Microservices

Interactive debugging is a process consisting of a linear sequence of interactions between a software developer and a piece of software. The developer issues commands to control the execution of the program and then observes its state. It is an integral part of the whole process of software development and operation. This section introduces the notion of *continuous debugging* in order to fit debugging into the methodology for developing cloud native applications in microservices architecture. We then analyse the specific requirements for debugging microservices.

A. Continuous Debugging

The DevOps methodology is the current best industrial practice for the development and operation of cloud-native applications with microservices, so it is important for a debug facility fitting into the DevOps methodology.

A key characteristic of the DevOps methodology is continuous development, which has implications for both individual components and the system as a whole. For components, continuity means that development activities proceed with minimal delay. For example, as soon as coding is finished, unit testing must begin, followed immediately by integration testing, then by deployment of the component to a stage environment, then user testing and so on. From this perspective, each component moves through the DevOps pipeline smoothly and continuously until it is delivered. For the system as a whole, continuity means that it is constantly changing and evolving as new components are added in, and old ones are modified and removed, simultaneously. System level releases and version updates are replaced by simultaneous evolutions of its components. While this is happening, the system must operate continuously without interruption.

The current theory and practice of continuous development consist of four elements: continuous testing, continuous integration, continuous deployment, and continuous delivery. Debugging is missing from the DevOps process and pipeline models. Debugging tools are not included in the suite of pipeline automation toolkits. Here, we define a fifth element: the notion of *continuous debugging*. For individual components, debugging activities should take place as soon as a failure occurs, whether it is detected by the system automatically or manually. This could be during coding, but it could also be just after a failure of unit testing or of integration testing in a stage environment, and of course, it should occur immediately after the component fails in a production environment. For the system as a whole, there should be no interruption to operation, as debugging should be applied to components in parallel with the development and operation activities on other components. For example, the completion of debugging for a component should trigger regression testing.

The most fundamental principle of DevOps is to integrate development and operation. To support this principle means integrating debugging tools into the development environment as well as the operation environment, and including it in the automated pipeline of the DevOps process. The integration should be seemless, in keeping with the fundamental change that DevOps brings to the ownership of programs and project [6], as summarised by the slogan "*you develop it, you operate it*".

B. Requirements for Debugging Microservices

In order to perform continuous debugging, the requirements are that debugging should be:

1) *remote*: the developer issues commands from their workstation to operate a microservice running on a separate remote machine, the one where the failure originally occurred. It can often be impossible to set up an environment that replicates the failure on the programmer's own workstation.
2) *parallel*: it is possible for the developer to interact, ie issue commands and observe states, simultaneously with multiple services. This is required because a failure behaviour normally exhibits itself in the interactions between services, and the cause is normally not just a single bug in the service under investigation, but a combination of fault(s) in other services. The multiple services may even be on different machines, as the environment is distributed.
3) *online*: it is possible to enter and exit debug mode freely, resuming the service's normal operation once debugging has finished. Often it is necessary to examine many services to discover which is faulty and we must be able to do this without affecting the normal operation of the service.
4) *non-intrusive*: it does not require instrumentation code to be inserted into the program and remain in the code during normal operation of the program. Such code would cause a significant overhead on system performance.
5) *isolated*: debugging one service does not affect the functional operation or performance of other microservices, so the impact on the system as a whole is min-

imised, which is particularly important in a production environment.

Clearly, the requirements given above should be sufficient to support debugging as soon as failure occurs without affecting the operation of the system as a whole.

C. Weakness of Existing Debug Facilities

All existing modern IDEs integrate a software development environment with debugging facilities that typically include the following functions:

- Setting breakpoints in the code where the execution will stop, so that the program state can be observed. The execution then be resumed when observations have been made.
- Executing the program step-by-step so these observations can be made after each step. The steps can be of different granularities: a machine instruction, a high-level language statement, a method call, etc.
- Inspecting the values of variables after the execution stops at a breakpoint or after a step.

However, the conventional debugging experience that these facilities provide does not meet the requirements set out in subsection II-B. It is not online because execution must start from the beginning. It is not remote because the debug tool runs on the same machine as the microservice in order to control execution. It is not isolated because when a breakpoint is set, all threads and processes that hit the breakpoint will be paused.

As far as we know, there is no existing debugging facility that meets the requirements of continuous debugging microservices running on a cluster of machines [5]. More discussions on related works is given in Section V-A.

III. The Proposed Debug Facility

In this section, we propose a new debug facility for continuous debugging of microservices. It has been implemented and integrated into the integrated DevOps environment CIDE [8] for developing microservices written in the service agent-oriented programming language CAOPLE [7]. We have also designed and implemented a command line interface, which also accepts scripts, so that the debug facility can be integrated with other tools in DevOps pipelines. The examples given in this section are from the current implementation.

The debug facility consists of three tools, described in the following three subsections; their implementation is briefly reported in subsection III-D. Each provides commands that the user/developer can issue to one or more selected agents to control their execution or display information about their state or behaviour. The agents need not be instances of the same microservice nor on the same machine.

A. Execution Trace

A trace is a sequence of the instructions that an agent executes in a particular period of time. The user can start tracing a set of selected agents by using the *Start Trace* command, and finish by using the *Stop Trace* command. During tracing, for each selected agent, the sequence of executed instructions is saved to a separate file on the machine where the agent is running. That recorded trace can be transmitted to the user's workstation by using the *Get Trace* command. Figure 1 shows an example of trace.

```
agent Id:3c32cf0b-2c4f-4d68-983b-9a79d9e1c348          @192.168.1.83
Caste name: ForallListSum
Start Time: 1550938026551 == 2019/02/23 16:07:06

Time    State   Line    PC      Operator (dest, src)
0       Body    37      39:     add (x, cnt)
11      Body    37      40:     mov (cnt, )
11      Body    38      41:     add (xIndex, 1)
11      Body    38      42:     mov (xIndex, )
11      Body    38      43:     jump (26, )
11      Body    35      26:     CopyToStack (myList, )
11      Body    35      27:     getListLength (, )
11      Body    35      28:     small (xIndex, )
11      Body    35      29:     jump_false (44, )
11      Body    35      30:     CopyToStack (myList, )
11      Body    35      31:     OptGetListIndex (xIndex, integer)
11      Body    35      32:     mov (x, )
11      Body    36      33:     toString (x, )
11      Body    36      34:     add ("x== ", )
```

Figure 1. An Example of Execution Trace

As shown in Figure 1, a trace begins with a header, containing the agent's universal unique ID, its caste name (i.e. the name of the microservice), the IP address where the agent is running, and the time when the trace started. Alongside each instruction executed, the trace records in readable format the time in milliseconds from the start of tracing, the line number of the source code statement from which the instruction was generated, and the program counter (PC) i.e. the address of the instruction.

B. State Snapshot

The execution state of an agent is characterised by the program counter, values of its variables, the contents of its stack, and the messages in its queue still to be processed. The *Get State* command snapshots the first three of these and sends them as a single package to the user's workstation for viewing and analysis. Figure 2 shows an example of such a state snapshot. The header is similar to the trace information. Variables are listed with their names, data types and values; JSON format is used for structured data types such as arrays or records.

The message queue can be obtained with the *Get Message Queue* command. Each agent has its own message queue because interactions between agents occur with asynchronous messaging in the form of service request events and service response events. Figure 3 shows an example message queue for an agent. Note that each message contains the caste name, agent ID, agent IP address, event type and parameters.

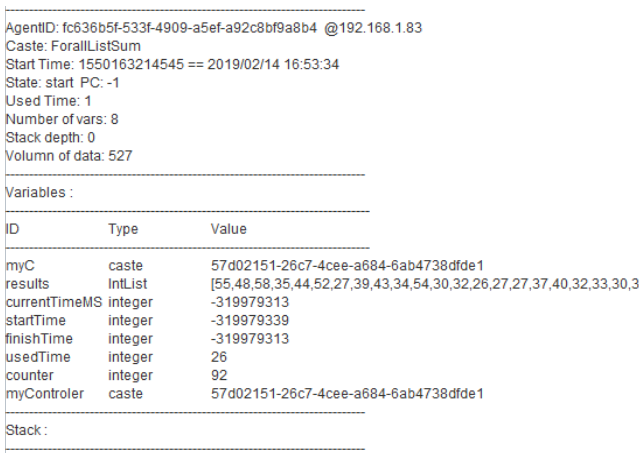


Figure 2. An Example of State Snapshot

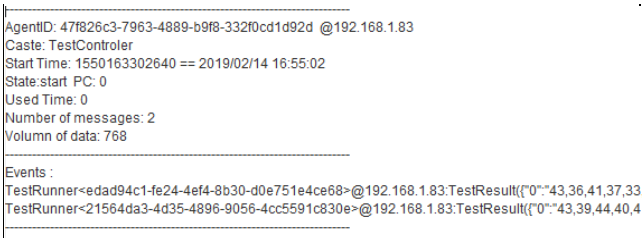


Figure 3. An Example of Message Queue Snapshot

Another way to obtain the state of an agent is to set a checkpoint, so that it saves the state information into a file when the program hits that specific point. Of course, multiple checkpoints can be set for one agent, and a checkpoint can be set on multiple agents. If an agent hits a checkpoint multiple times, a sequence of state snapshots will be recorded in one file. Each file holds the state of one agent.

The command *Add Checkpoint* adds one or many checkpoints to a set of selected agents; its parameter is a sequence of locations, which are either addresses of instructions in the object code of the agent or line numbers in its source code. The command *Clear Checkpoint* removes checkpoints from selected agents. Once checkpoints have been set, the *Start Checking* and the *Stop Checking* commands, respectively, start and stop the checkpointing. The recorded state snapshots can be transmitted to the user's workstation by issuing the *Get Checkpoints* command.

Figure 4 shows an example of such a checkpoint record. We can see that two state snapshots were taken at time moments 2991 and 3001 for a single checkpoint set at line 9 of the source code and instruction 11 of the object code. Each record shows the value of variables, contents of the stack and messages in the queue. Since there is a message in the queue from an agent of caste Peer at time moment 2991 but the queue is empty at time moment 3001, the message is processed in between these two time moments.

Figure 4. An Example of Checkpoint Records

C. Execution Control

The execution of a selected agent can be controlled by using the *Pause*, *Step Forward* and *Resume* commands. The *Pause* command temporarily halts execution of the selected agents and does so after the current instruction has completed to ensure that instructions are atomic. Once execution has been halted, the *Step Forward* command makes each selected agent execute one more instruction and halt again. The *Resume* command continues the normal executions of the selected agents.

Breakpoints can also be set for a set of selected agents. When the execution of an agent hits a breakpoint, it will halt. The *Run to Next Breakpoint* command will let each selected agent execute until it hits a breakpoint again. Like checkpoints, breakpoints can be specified with either source code line numbers or object code instruction addresses. Setting and removing breakpoints can be performed by using the *Add Breakpoint* and *Clear Breakpoint* commands on a number of selected agents, and each agent can have many breakpoints. Agents can have different breakpoints even if they are instances of the same caste.

Note that the *Get State* and *Get Message Queue* commands can be used both when the agent is executing and when it is paused. For example, using them before and after a *Step Forward* shows the effect of one individual instruction on the state of an agent.

D. Implementation of The Debug Facility

The debug facility has been fully implemented by modifying the CAVM virtual machine [7] and adding functions to receive the commands from users as service requests, execute the commands and respond with messages returned to the service requester.

The graphical user interface of the integrated DevOps environment CIDE [8] has been modified to include a set of buttons etc. for the user to issue commands to selected agents as service requests to the virtual machines where the

agents are executing; it then receives the messages from the modified CAVM and display the received data. Therefore, the debugging facility is integrated into the DevOps environment CIDE. Details of the implementation will be reported in a separate paper for reasons of space.

Figure 5 gives the GUI for the debug facility as a part of CIDE's runtime management of agents.

The debugging facility can also be invoked using command line instructions, making it possible to write shell scripts that integrate with the other monitoring and analysis tools in a DevOps pipeline. The format of instructions is as follows:

$\texttt{me} - \texttt{dbg}\ \langle\texttt{command}\rangle\ \{\texttt{agentID@IP}, \}^{+}[\texttt{parameters}]$

The debug commands are listed in Table I.

Table I
COMMAND LINE DEBUG INSTRUCTIONS

Command	Parameter	Meaning
pause		Pause the execution of the agent
resume		Resume the execution of the agent
step		Execute one more instruction
start_trace		Start tracing the agent
stop_trace		Stop tracing the agent
get_trace		Get recorded trace of the agent
get_state		Get the state of the agent
get_msg		Get the agent's messages in queue
set_ checkpoints	Points	Set checkpoints to an agent
clear_checkpoints		Remove the agent's checkpoints
start_check		Start to record the agent's state at checkpoints
stop_check		Stop recording the agent's state at checkpoints
get_checkpoints		Get recorded states of the agent at checkpoints
set_breakpoints	Points	Set breakpoints to the agent
clear_breakpoints		Remove all breakpoints of the agent
run_to_breakpoints		Execute the agent to the next breakpoint
clear	Ip Address	Remove debug data on the machine

The debug facility as implemented above enables debugging activities to be conducted on microservices remotely, online, and isolated; the microservices are agents executing in a distributed and parallel fashion. It does not require instrumentation of the code so it is non-intrusive. All the requirements of Section II-B are satisfied. The debug facility is integrated to the DevOps environment and the debugging commands can be scripted. In this way, it supports the continuous debugging and the integration principle of DevOps.

IV. EVALUATION

The implementation of the debug facility modifies the virtual machine CAVM, which forms the runtime environment of CAOPLE programs. Thus, it brings additional runtime overhead to the performance of the programs. Controlled experiments have been conducted to evaluate this overhead. This section reports the findings.

A. Experiment 1

Experiment 1 is designed to answer the following research question.

- **RQ1**: *In terms of performance, how does the system with the debug facility compare with the system without*?

To answer this question, a benchmark called BM1 of six programs was designed and coded, and run on the original CAVM (without debug facility) and the modified CAVM (with the debug facility). As shown in Table II, the benchmark combines numerical and text processing with invocations of library functions, system functions, and file operations. Each was executed 100 times consecutively and the average taken.

Table II
PROGRAMS IN BENCHMARK BM1

Program	Function	Main features
P1	Summation of a list of 400 integers from 0 to 399	Numerical calculation
P2	Generate 500 random numbers	Library function call
P3	Probes and displays the workload on the computer 100 times	System functions call
P4	Read 200 lines of text from a file and display them on the screen	File operations, Text processing
P5	Take 500 readings of the CPU usage and calculate the average	System function calls, Numerical calculation
P6	Calculate the average of 100 random numbers	Library function call, Numerical calculation

The experiment is repeated on two computer systems: a desktop computer and a server in a cluster; see Table III.

Table III
THE EXPERIMENT PLATFORM

Specification	Desktop PC	Server
OS	Windows Vista	Windows Server 2012 R2
No. of Nodes	1	16
No. of Cores	4	4
Memory Size	8 GB	32GB
Hard Drive Size	300GB	300GB
CPU	Intel Core 2 2.67GHz	Intel Xeon E3-1230v5 3.41GHz

Table IV shows the average execution time (column *Time*) for each benchmark program without the debug facility, the additional execution time (column *Diff*) when running the debug facility and the increase in running time as a percentage (column *Rate*).

Table IV
RESULTS OF EXPERIMENT 1

	Server			Desktop PC		
	Time (ms)	Diff	Rate %	Time (ms)	Diff	Rate %
P1	106.02	1.16	1.09	172.67	3.03	1.75
P2	131.01	3.44	2.63	171.12	3.99	2.33
P3	117.14	1.19	1.02	217.79	2.34	1.07
P4	14.6	0.52	3.56	79.44	1.34	1.69
P5	367.48	0.05	0.01	591.99	11.62	1.96
P6	6.18	0.42	6.80	11.28	0.27	2.42
Avg	123.74	1.13	2.52	207.38	3.77	1.87

We can see that the overhead of the debug facility is negligible. On average, it is 2.52% for the server cluster

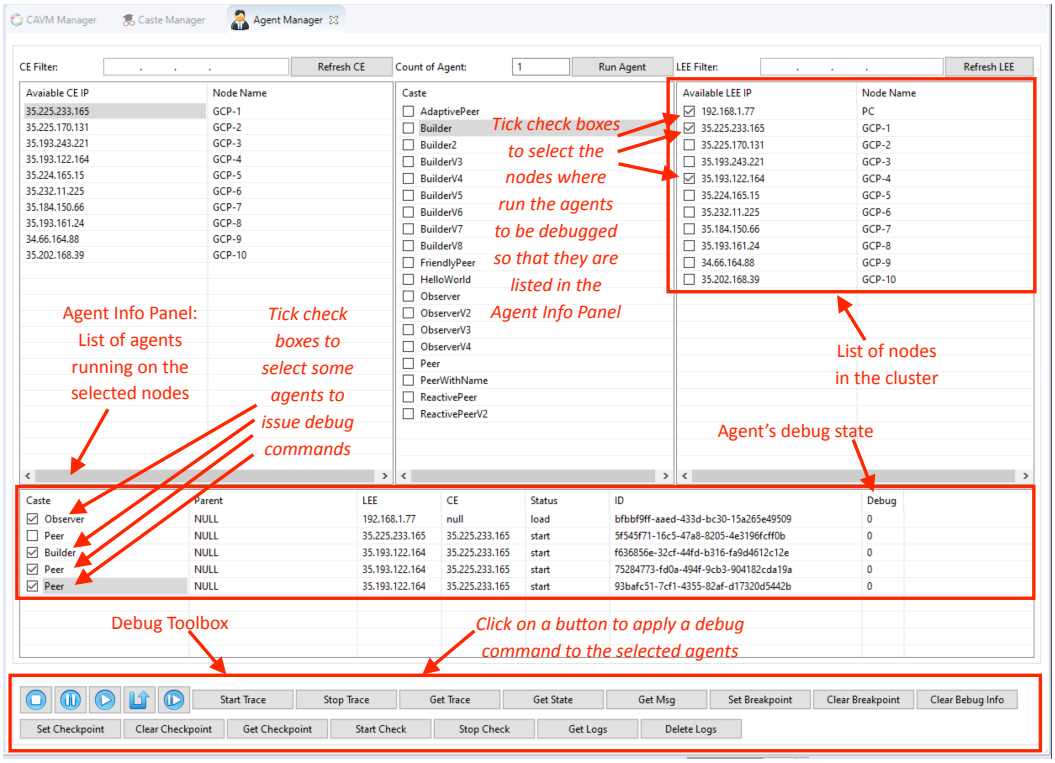


Figure 5. Graphical User Interface of the Debug Facility

and 1.87% for the desktop PC. The nondeterminism due to OS scheduling and JIC etc. has more effect.

B. Experiment 2

This experiment aims to investigate the impact of debug operations on the execution speed of the agent. Note that tracing an agent's execution will record every instruction executed by the agent, and thereby increase its execution time. Since the debug facility only affects the agent being debugged, the impact will be limited to the agent itself. For state snapshot operations, the agent's execution must be suspended while a state snapshot is carried out to ensure data integrity. The impact of that state snapshot is determined by the length of the time for which the agent must be suspended for it to happen. Therefore, we have the following research questions.

- **RQ2.1**: *When an agent is running with tracing activated, how much will its performance downgrade*?
- **RQ2.2**: *How long does it take to get an agent's state snapshot*?
- **RQ2.3**: *How long does it take to get the list of messages in an agent's message queue*?

The following experiments were designed and conducted to answer these questions.

1) Experiment 2.1: To answer research question RQ2.1, the same benchmark BM1 is used first with tracing enabled and then without. As before, each program in the benchmark is repeated 100 times and the average is calculated. The experiment is repeated on the same computer systems as in Experiment 1. Table V shows the average execution times without tracing in column Time, the increase in execution time in column Diff when tracing is switched on, and the percentage increase in column Rate. The results show that overhead of tracing is 4.21% for the server and 3.64% for the desktop PC.

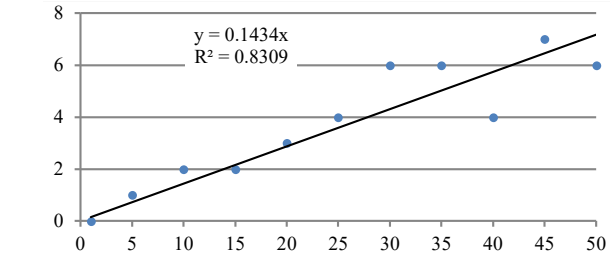
Table V
RESULTS OF EXPERIMENT 2.1

	Server			Desktop PC		
	Time (ms)	Diff	Ratio%	Time (ms)	Diff	Ratio%
P1	107.18	0.61	0.57	175.7	10.29	5.86
P2	134.45	3.04	2.26	175.11	14.21	8.11
P3	118.33	0.93	0.79	220.13	5.22	2.37
P4	15.12	0.99	6.55	80.78	0.05	0.06
P5	367.53	7.07	1.92	603.61	0.23	0.04
P6	6.6	0.87	13.18	11.55	0.62	5.39
		Average	4.21		*Average*	3.64

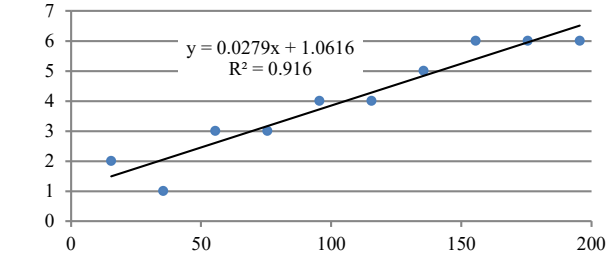
2) Experiment 2.2: To answer research question RQ2.2, a new benchmark BM2 was designed, in which each program is characterised by two parameters: the number of variables and the data size of each variable. Using BM2, the relationship between these two factors and the time taken to snapshot can be studied. When each code sample in BM2 was run, 10 state snapshots were taken, and the average

execution time was calculated. Table VI shows the execution times in column T, and the volumes of data in column V for various numbers of variables (Var).

Statistical analysis reveals that the execution time for taking a state snapshot is linear in both the number of variables and the volume of data. For example, Figure 6(a) shows the execution times as the number of variables varies from 1 to 50 with each variable being a list of 1000 integers. Figure 6(b) shows the execution times for taking a state snapshot as the total volume of data varies from 20 KB to 200KB when there are 50 variables.



(a) Time (ms) to take a state snapshot for x variables (x=1..50), where each variable is a list of 1000 integers.



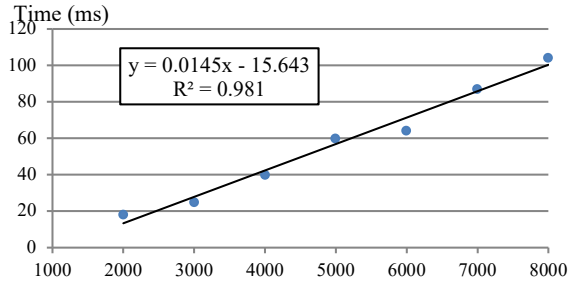
(b) Time (ms) to take a state snapshot of 50 variables, where the volume of data in each snapshot varies from 20 KB to 200KB.

Figure 6. Analysis of The Result of Experiment 2.2

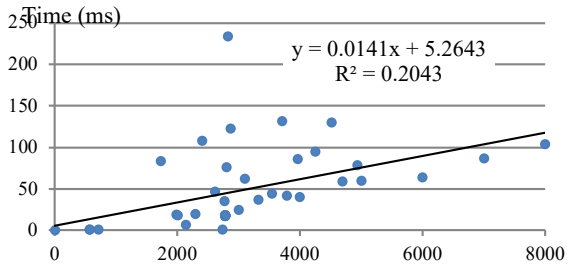
3) Experiment 2.3: To answer research question RQ2.3, a third benchmark BM3 was designed, in which each program is characterised by the length of message queue that the agent will have. When a program in BM3 was run, the *Get Message Queue* operation was performed 10 times and the average execution time calculated.

The experiment was conducted in two scenarios: an *ideal scenario* in which there were no other agents running on the machine and a non-ideal *workload scenario* in which there were a number of other agents running on the same machine at a nearly saturated workload.

Figure 7(a) and (b) show the distributions of the times taken to take a snapshot of the message queues in the two scenarios. In the ideal scenario, snapshotting was not interrupted by other agents (i.e. threads and processes) running on the same machine. The execution time was linear in the number of messages in the queue. In the workload scenario, the time to snapshot was affected in this way but statistically the same linear increase pattern can be observed.



(a) In the ideal scenario



(b) In the workload scenario

Figure 7. Times to snapshot a message queue

C. Conclusions of the Experiments

From the experimental data, we can draw the following conclusions.

- The overhead of the debug facility is less than 3% on average, so it should not be noticeable to the user.
- The overhead of the trace function is less than 5% on average. In the worst case, it was 13.18% but this was as a replacement for traditional step-by-step interactive debugging so the increase in execution time is acceptable.
- The overhead of taking a snapshot of an agent's state is negligible. Even if there are 50 variables each holding 1000 integers (200KB) the time taken is only a few milliseconds (6 ms). The execution time is linear in both the number of variables and the total size of the data.
- The execution time to take a snapshot of the message queue is linear in the number of messages in the queue in the ideal scenario when no other concurrent thread or process interrupts the snapshot operation. In a heavy workload scenario, when a snapshot can be interrupted, it is still linear in the number of messages in the queue, statistically speaking.

Therefore, the overhead of the debug facility as a whole is acceptable.

V. Conclusion

A. Related Work

Debug facilities are an integral part of modern integrated software development environments (IDEs) like Eclipse, NetBeans, IntelliJ, XCode, Visual Studio, etc. They provide

Table VI
RESULTS OF EXPERIMENT 2.2

Var=2		Var=6		Var =11		Var = 16		Var = 21		Var = 26		Var = 31		Var = 36		Var = 41		Var = 46		Var = 51	
V	T	V	T	V	T	V	T	V	T	V	T	V	T	V	T	V	T	V	T	V	T
326	0	1554	1	3090	0	4630	0	6170	0	7710	3	9250	1	10790	1	12330	0	13870	1	15.41	2
726	1	3554	1	7090	0	10630	1	14170	0	17710	3	21250	1	24790	1	28330	1	31870	5	35.41	1
1126	4	5554	1	11090	1	16630	2	22170	1	27710	1	33250	2	38790	1	44330	2	49870	2	55.41	3
1526	0	7554	0	15090	1	22630	1	30170	1	37710	2	45250	3	52790	3	60330	2	67870	5	75.41	3
1926	1	9554	1	19090	1	28630	1	38170	2	47710	2	57250	3	66790	3	76330	35	85870	3	95.41	4
2326	1	11554	1	23090	2	34630	18	46170	2	57710	2	69250	3	80790	4	92330	3	103870	5	115.41	4
2726	1	13554	1	27090	1	40630	2	54170	2	67710	4	81250	3	94790	5	108330	6	121870	6	135.41	5
3126	0	15554	1	31090	3	46630	2	62170	4	77710	3	93250	3	108790	9	124330	4	139870	6	155.41	6
3526	3	17554	3	35090	1	52630	2	70170	4	87710	3	105250	4	122790	4	140330	6	157870	6	175.41	6
3928	0	19560	1	39101	2	58646	2	78191	3	97736	4	117281	6	136826	6	156371	4	175916	7	195.461	6

the functions for setting breakpoints, stepping through the program and inspecting the memory state. A typical example is the GNU Project Debugger GBD, designed for offline and local debugging. Debug facilities and tools have also been developed for various programming languages, such as *dlv* for Go, *ptvsd* for Python, etc. As discussed in Section II-C, they do not meet the all requirements of debugging microservices, nor do they support continuous debugging as a part of the DevOps pipeline. As Levine pointed out, the debugger *"is totally missing from the toolbox of microservice developers"* [5].

The closest near matches are works on debugging the programs running on supercomputers [9], the *BigDebugger* for debugging MapReduce applications for Big Data analysis [11], and two more recent developments: Cloud Debugger for Google's Cloud Platform for microservices [14] and Squash [17]. They are discussed below.

To enable the debugging of parallel programs running on supercomputer architectures, Jin et al. [9] developed a code library for launching a debug facility simultaneously on the nodes in a supercomputer from the front-end, and sending data collected back to the developer's workstation. The main function of the library code is the communication between the front-end and back-end in supercomputer systems. A similar work is reported in [10] addressing the same problem.

In the context of Big Data analysis applications, Gulzar et al. [11] developed a debug facility called *BigDebug* to extend the Spark system. It provides a set of debug primitives to enable debugging of the MapReduce type of computation on Hadoop clusters. Their debug facility includes the following functions.

- Simulated breakpoint, which enables the user to inspect intermediate results at a given "breakpoint" and then resume the execution. This creates the illusion of a breakpoint, even though the program is still running on the cloud in the background.
- Guarded watchpoint, which enables the user to query a subset of data matching a given guard condition.
- Data trace, which enables the user to trace forward and backward through the processing of an individual data record to identify the origin of the final or intermediate output.
- Crash culprit and remediation, which sends all required information to the driver when the crash occurs so that the user can determine the culprit and take actions to fix the code and then carry on the computation.

Compared to our debug facility, the overhead of BigDebug is much higher: up to 24% for recording tracing, 19% for crash monitoring, and 9% for watchpoint [11]. Moreover, BigDebug is only applicable to the MapReduce type of dataflow computations. It is not applicable for microservices.

Tracing has been widely used in practice for debugging concurrent systems and software running on parallel hardware architectures such as multiple core systems [12]. Google's Dapper provides context-based tracing in particular. It relies on the homogeneous infrastructure of common RPC libraries to minimise the instrumentation burden. Its data model (call graph) and architecture has become the de facto standard for trace collection. Zipkin, created at Twitter, is an open source clone of Dapper. Zipkin and its derivatives including Amonon's X-Ray are in widespread use. However, as Alvaro [13] pointed out, despite the fact that distributed systems are a mature research area in academia and are ubiquitous in industry, the art of debugging distributed systems is still in its infancy.

A recent development in industry is Google's *Cloud Debugger* [14]. It has the features of remote online debugging, which is called real-time debugging in [14]. It provides two debugging tools: *snapshot* and *logpoints*. The former gets the values of selected variables when the execution of an instance hits a snapshot location in the code, and sends the value to the user's workstation. The latter generates a log entry in the target log system when the execution of an instance hits a logpoint location in the code. Both of them are similar to our checkpointing functions, but there are at least three important differences: Cloud Debugger does not collect information about message queues, which is vital for debugging microservices. Its snapshots and logpoints are applied to "all instances of the app", rather than just the

selected instances, as in our approaches. Its snapshots require the user to specify which variables are required, whereas we obtain values of all valid variables, including dynamically-created variables.

Looking more closely at the last of these differences, although selecting just some of variables can reduce the amount of data transferred to the user's workstation, this is offset by the longer computation time required to set the snapshot, which takes about 40 seconds [14]. In our approach, the equivalent task of setting a checkpoint can be done instantly. Moreover, the user may not be able to know the names of variables if they do not have the source code and even if they did, they would not be able to specify dynamically generated variables and compiler-generated internal variables. Finally, user-specified variables may not be in scope for some locations of the program.

It is worth noting, moreover, that Cloud Debugger does not have any tool for control the executions of the application under debug. Finally, in order to reduce the overhead, Cloud Debugger restricts the time period for which a snapshot or logpoint is effective and enables the user to set conditions for the functions. In spite of this, Cloud Debugger's overhead is still higher than ours. It causes an additional latency of each service request about 10ms on average [14]. With less than 10ms, our debugger can get the values of more than 50 variables and 200KB of data (equivalent to 50,000 integers); see experiment data in Figure 6 of Section IV-B2.

Squash is an open source project of *solo.io* that provides an interface between an existing IDE and the software running on remote machine for testing microservices using existing debug facilities like *dlv* for Go, *ptvsd* for Python and GNU Project Debugger *gdb* [17], [18]. Currently, it supports *VS Code*, *Intellij* and *Eclipse* IDEs for microservices running on Kubernetes and OpenShift platforms and written in programming languages (microservices frameworks) Go, Java, JavaScripts (Nodejs) and Python [18]. It completely relies on existing debuggers to perform debugging activities, but provides no new debugging facilities. Therefore, it only meets some of the requirements of debugging microservices that we recognised in Section II-B; for example, it does not enable online debugging microservices.

In the wider context of site reliability engineering, the current trend in industry is away from monitoring to towards *observability engineering* (or *observability* for short), which consists of four pillars: *monitoring*, *alert and visualisation*, *distributed system tracing* and *log aggregation and analysis* [20]. It gives better support for debugging than monitoring does by providing more information, especially the "*highly granular insights into the behaviour of systems*". However, as Shridharan pointed out [15], observability is not debugging, which is characterised as *"an iterative process which involves introspection of the various observations and facts reported by the system, making the right deductions and testing whether the theory holds water*". Thus, observability is not sufficient for debugging microservices.

Both observability techniques and debug facilities provide a means of observations for microservices. However, there are subtle differences. For example, considering the observability as a system quality attribute like usability, efficiency, maintainability, testability, etc., Baron Schwartz defines observability as "*a measure of how well internal states of a system can be inferred from knowledge of its external outputs*" [16]. Indeed, existing observability techniques treat microservices as a black box and only observe external features. In contrast, when debugging, one does not only observes a system's external outputs, but perhaps more importantly, its internal states too. Moreover, controllability as a mathematical dual of observability in control theory is equally important for debugging but completely missing in the observability tool box. Many debugging facilities, including ours, not only contain tools for observation on system's behaviour and state, but also tools to control the execution of the system.

There are many tools available for each of the four pillars of observability. These tools are increasingly integrated together, as with Google Cloud operations suite and its predecessor Stackdriver [19], and they help to diagnose failure and their causes. For example, through telemetry visualisation and alert tools, one can recognise whether a failure occurred in the system, and predict whether a failure could occur or is in progress. Through message tracing and log aggregation and analysis tools, one can identify which machine and which microservice caused a problem. Observability tools are widely used in industry to locate bugs in a microservice system [21], and are becoming an active research subject. For example, Shang et al. [22] proposed a data mining approach to the analysis of log files to identify failures of deployment of big data analysis applications to Hadoop clusters. Tong et al. [23] also employed data mining techniques to analyse log files of cloud platforms to pinpoint bug-induced software failures. They can identify which processes cause the system failure, but not where exactly this happened in the code. It is highly desirable to integrate debugging tools into the microservice development toolbox. For example, Google Cloud operations suite has recently included the debugging tool Cloud Debugger [19].

B. Future Work

It is interesting to study the effectiveness of the proposed debug facility in practice, for example, through empirical studies with professional software developers and using a benchmark like DbgBench [24]. It is also worth noting that in recent years there has been a rapid growth in research on automated debugging; see [25] for a recent survey of research on the topic. The execution traces (also called *execution profile* in the literature) are the main input to such automated debugging algorithms. It may be interesting to investigate how to link our trace tool to such automated

debugging tools. The implementation of our debug facility is through modification of the language's virtual machine (i.e. the runtime environment). The approach should be applicable for other languages that are implemented by using language virtual machines such as Java, Python, etc. It will be interesting to explore how to implement a similar debug facility for Java and Python.

REFERENCES

[1] J. Lewis and M. Fowler, *Microservices*, Online at http://martinfowler.com/articles/microservices.html, 25 Mar. 2014. Last access:27 Jun.2020.

[2] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, OReilly, Feb. 2015.

[3] L. Krause, *Microservices: Patterns and Applications*, Microservicesbook.io, April 2015.

[4] H. Zhu and I. Bayley, If Docker Is The Answer, What Is The Question? A Case for Software Engineering Paradigm Shift Towards Service Agent Orientation, in *Proc. of SOSE 2018*, pp152-163, Mar. 2018.

[5] I. Levine, *Squash: Microservices Debugger*. Online at https://medium.com/solo-io/squash-microservices-debugger-5023e27533de, 5 Mar. 2018. Last access:25 Jun.2020.

[6] S. Sharma, and B. Coyne, *DevOps for the Dummies*, 2nd IBM Limited Edition. John Wiley & Sons, 2015.

[7] C. Xu, H. Zhu, I. Bayley, D. Lightfoot, M. Green, and P. Marshall, CAOPLE: A Programming Language for Microservices SaaS, in *Proc. of SOSE 2016*, pp42-52, Apr. 2016.

[8] D. Liu, H. Zhu, C. Xu, I. Bayley, D. Lightfoot, M. Green, P. Marshall, CIDE: An Integrated Development Environment for Microservices, in *Proc. of SCC 2016*, pp808-812, Jun.2016.

[9] C. Jin, D. Abramson, M. N. Dinh, A. Gontarek, R. Moench and L. DeRose, A Scalable Parallel Debugging Library with Pluggable Communication Protocols, in *Proc. of CCGrid 2012*, pp252-259, May 2012.

[10] J. Zhang, Z. Luan, W. Li, H. Yang, J. Ni, Y. Huang and D. Qian, CDebugger: A Scalable Parallel Debugger with Dynamic Communication Topology Configuration, in *Proc. of CSC 2011*, pp228-234, Dec. 2011.

[11] M. A. Gulzar, M. Interlandi, S. Yoo, S. D. Tetali, T. Condie, T. Millstein, and M. Kim, BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark, In *Proc. of ICSE 2016*, pp784-795, May 2016.

[12] A. Spear, M. Levy, and M. Desnoyers, Using Tracing to Solve the Multicore System Debug Problem, *IEEE Computer* 45(12), pp60-64, Dec. 2012.

[13] P. Alvaro, OK, But Why? Tracing and Debugging Distributed Systems, *Comm. of ACM*, 60(7), pp46-47, Jul. 2017.

[14] Google, "Cloud Debugger", Online at https://cloud.google.com/debugger, Last access: 6 Jun. 2020.

[15] C. Sridharan, *Monitoring and Observability*, Online at https://medium.com/@copyconstruct/monitoring-and- observability-8417d1952e1c, 4 Sept. 2017. Last access: 6 June 2020.

[16] B. Schwartz, *Monitoring Isn't Observability*, Online at https://orangematter.solarwinds.com/2017/09/14/monitoring-isnt-observability/, 14 Sept. 2017. Last access: 6 Jun. 2020.

[17] D. Rettori, *Squash: The Definitive Cloud-Native Debugging Tool*, Online at https://itnext.io/squash-the-definitive- cloud-native-debugging-tool-89614650cc94, 8 Mar., 2019. Last access: 10 Jun. 2020.

[18] Solo.io, *Debug Your Microservice Applications from Your Terminal or IDE While They Run in Kubernetes*, Squash Official Website. Online at https://squash.solo.io. Last access: 25 Jun. 2020.

[19] Google Cloud Platform, *Google Cloud Operations Suite Documentation*, Online at https://cloud.google.com/stackdriver/docs. Last access: 25 Jun. 2020.

[20] A. Asta, *Observability at Twitter: Technical Overview*, Part 1, Online at https://blog.twitter.com/engineering/en_us/a/2016/observability-at-twitter-technical-overview-part-i.html, 18 Mar. 2016; Part 2, Online at https://blog.twitter.com/engineering/en_us/a/2016/observability-at-twitter-technical-overview-part-ii.html, 22 Mar. 2016. Last access: 8 Jun. 2020.

[21] V. Tarcic, *The DevOps 2.0 Toolkit: Automating the Continuous Deployment Pipeline with Containerized Microservices*, Lean Pub, 2016.

[22] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan and P. Martin, Assisting Developers of Big Data Analytics Applications when Deploying on Hadoop Clouds, In *Proc. of ICSE 2013*, pp402-411, May 2013.

[23] T. Jia, Y. Li, Ho. Tang, Z. Wu, An Approach to Pinpointing Bug-Induced Failure in Logs of Open Cloud Platforms, in *Proc. of CLOUD 2016*, pp294-302, Jun. 2016.

[24] M. Bohme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, How Developers Debug Software - the DbgBench Dataset, in *Proc. of ICSE-C 2017*, pp244-246, May 2017.

[25] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, A Survey on Software Fault Localization, *IEEE Transactions on Software Engineering* 42(8), pp707-740, Aug. 2016.