RETORCH: Resource-aware End-to-end Test Orchestration

Cristian Augusto [0000-0001-6140-1375] [1], Jesús Morán [0000-0002-7544-3901] [1], Antonia Bertolino [0000-0001-8749-1356] [2], Claudio de la Riva [0000-0001-5592-9683] [1], Javier Tuya [0000-0002-1091-934X] [1]

[1] Computer Science Department, University of Oviedo, Gijón,
[2] ISTI-CNR, Consiglio Nazionale delle Ricerche, Pisa, Italy

[1]{augustocristian, moranjesus, claudio, tuya}@uniovi.es
[2] antonia.bertolino@isti.cnr.it

Abstract. Continuous integration practices introduce incremental changes in the code to both improve the quality and add new functionality. These changes can introduce faults that can be timely detected through continuous testing by automating the test cases and re-executing them at each code change. However, re-executing all test cases at each change may not be always feasible, especially for those test cases that make heavy use of resources thoroughly like End-to-End test cases that need a complex test infrastructure. This paper is focused on optimizing the usage of the resources employed during End-to-End testing (e.g., storage, memory, web servers or tables of a database, among others) through a resource-aware test orchestration technique in the context of continuous integration in the cloud. In order to optimize both the cost/usage of resources and the execution time, the approach proposes to i) identify the resources required by the End-to-End test cases, ii) group together those tests that need the same resources, iii) deploy the tests in both dependency isolated and elastic environments, and iv) schedule their parallel execution in several machines.

Keywords: Software testing, Continuous integration, Continuous testing, Testing in the cloud, End-to-End testing, Test orchestration.

1 Introduction

Continuous integration practices and methodologies are based on incremental changes of the code to improve quality or add new functionalities [1]. However, while introducing new features in the code, new faults can be introduced as well, which as a consequence can destabilize even code that was successfully tested in the past. To ensure that the modifications and the new code do not endanger the existing functionality, regression testing [2] is standard practice. In modern agile processes, though, in which new versions of software are continuously and frequently delivered within very short cycles, regression testing may face many challenges.

This is authors' version of paper published into the Proceedings of Quatic 2019.
The final authenticated version is available online at
https://doi.org/10.1007/978-3-030-29238-6_22

One emerging practice to shorten the validation of newly released version is *continuous testing* [3]: it consists of automating the test cases and re-executing them before any new release in the source code repository. However, a well-known problem is that as the number of tests increases, re-executing all of them at each frequent change may not be possible due to the extent of resources employed like the computational cost of execution, time required, or the number of virtual instances needed. As a solution to partially address this problem, many test minimization and prioritization techniques [2] have been proposed to identify a minimal subset of test cases or optimize their order of execution, respectively. The objective of these techniques is to look for a tradeoff between the probability of discovering the faults potentially added with a modification and the resources employed for regression testing. The prioritization techniques permute the execution order of the test cases aimed to firstly execute the relevant tests, but the whole execution of the test suite remains expensive unless the tester decides to execute only a subset of the more relevant tests through a minimization technique. The latter techniques reduce the usage of resources by not re-executing all tests, but they neither optimize the resources of the test executed nor alleviate the thoroughly use of resources in the whole test suite.

One of the testing stages that require a large amount of physical-logical-computational resources is End-to-End testing (from now onwards referred to as E2E), i.e., the test of the whole flow from start to end of the user interaction with the system. In E2E testing, the application of techniques such as parallelization, minimization or reduction may not be effective for cost reduction: this type of tests usually requires large amounts of resources in the broadest sense, including the high execution time, the cost of replicating resources, the resources to be made available, among others. In addition, the set-up of the testing environment acquires great importance. Thus, if this set-up requires a large amount of time compared with that employed in test execution, parallelizing the test cases in separate instances without a proper strategy would not solve the problem: for those tests sharing the usage of heavy resources, parallelization would be inefficient and a best solution would be to set up the test environment once and execute them in sequential way. Therefore, to optimize the cost of E2E testing, detecting the dependencies between the tests and the resources they require is a crucial aspect.

Moving the testing to the Cloud [4] is commonly acknowledged as a solution to reduce the cost of testing, especially to exploit the potential of unlimited resources and scalability delivered on demand. ElasTest [5] is an open source platform aimed to support Cloud testing and simplify the E2E test process. It avoids several testing dependence problems by providing dependency isolation through the containerized execution of the tests. This is done through the TJobs that are the tests together with the Docker containerized system under test (SUT,) customized to provide not only the production environment, but also utilities to execute, monitor and collect testing information.

Containerization has provided new advantages in the virtualization field, reducing the amount of both resources and time required to deploy a service in an isolated environment. The SUT instantiation can take advantage of the containerization in order to be deployed several times in the same machine, avoiding common problems like dependencies. However, in the current version, the ElasTest containerized execution presents the problem that it needs the instantiation of the resources required for each

container causing underusage of those resources. Our proposal is intended to reduce the number of resources used in the containerized execution of the test during E2E testing and it could be integrated into the ElasTest platform to support resource-aware Cloud testing orchestration. We call the approach RETORCH (**R**esource-aware **E**2E **T**est **ORCH**estration).

We present the concept behind the RETORCH approach and show an example of its application to a demonstrator from the ElasTest project. Precisely, this paper includes the following contributions:

1. A motivation and description of the RETORCH framework through which to perform the E2E resource identification, the test grouping and scheduling.
2. An illustrative example scenario of RETORCH usage.

The rest of the paper is organized as follows. Section 2 describes the related work. Section 3 introduces the orchestration approach proposed and defines its terminology. Section 4 describes a working example related to a teaching online service (Full Teaching application using the OpenVidu Streaming Engine). Finally, the conclusions and future work are in Section 5.

2 Related work

RETORCH is motivated by the Multi Objective Regression Test Optimization approach presented by Harman in [6]. He discussed several cost and value-based objectives for testing, supporting the new point of view that testing optimization should be performed by considering in combined way the several different types of resources needed.

Despite the recent advances in the efficient and effective use of resources during testing, there remain several open challenges [7] to be addressed when performing test prioritization, selection, and minimization. Several authors have studied approaches to optimize these techniques considering both cost and rate of fault detection [8–10]. Our proposal has some aspects in common with prioritization techniques [2], as we schedule the test cases aiming at improving one metric of the tests contained on the suite (in our case resource efficiency)

To make test suite prioritization, test dependencies are an important aspect to take into account. Some authors have proposed techniques and tools to detect dependencies between test cases. Bell et al. [11] provide a dependency detection tool (Electrictest), and compare it to other state of the art tools, getting a similar fault detection rate with lower slowdown. Gyori et al. [12] introduce formally the concept of test Pollution problem and present a technique (called POLDET) that detects in execution time the "polluting" tests. Another technique that addresses the test dependence problem is in [13], where Gambi et al. provide an evolution of their prior approach (ElectricTest), and test it empirically with good results: they discover the known dependences and find another one that previous tests and tools did not discover.

Test resource optimization has been widely treated in the literature. In [14] Gambi et al. provide a solution that allows the developer to choose between different resource

optimization parameters, as time, cost or a mix of both. In [15] Sundar et al. address the test execution optimization problem considering the cost, and present the concept of test plan derivation. Test plan derivation is composed by three processes of test resources partition that redistribute the tests to ease test parallelization. In another work by Yu et al. [16], the resource optimization problem is addressed by performing a clustering of test cases with resources. Unlike those works, we understand as a resource a wide range of terms beyond time or cost, and aim to optimize the testing depending on several aspects. A similar aim is pursued by García et al. [17], who propose to orchestrate test cases (essentially to find a proper selection and sequencing) depending on the outcome of test execution (verdict-driven) or on the produced output (data-driven).

In the field of Cloud services, several approaches have been proposed to face similar problems, e.g. Esfahani et al. in [18], expose the Cloud build infrastructure of Microsoft (so-called CloudBuild). This infrastructure presents similar dependences issues, which they address by extracting dependence graphs and deriving the dependencies on them automatically. We propose a future line of work that aims at a similar automated detection of the test resources into the containers.

3 Approach

The RETORCH framework aims at optimizing the cost/usage of resources orchestrating the E2E test cases in different machines based on the resources needed to execute each test. Fig.1 depicts the core concept of the orchestration starting from the E2E test

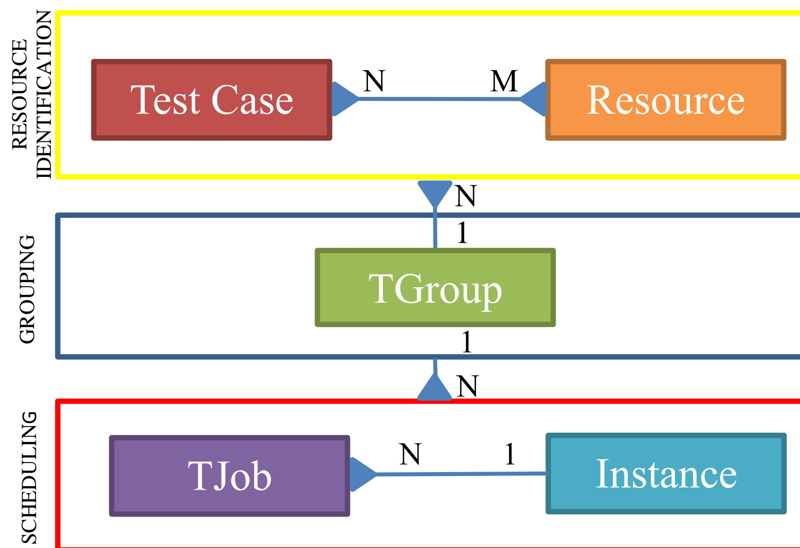

Fig. 1. Scheme of the RETORCH approach

cases to their execution in several machines/instances grouping those tests that use homogeneous resources in order to optimize both resources and execution time.

As first step, the resources used by each test case are identified to detect those tests that require the same resources (Resource identification). According to the resources identified, some tests can be executed together while others cannot because of incompatibilities in their allocated resources or in the way in which they access these resources. Then, those test cases that can be executed together are grouped to arrange their execution and reuse their resources to optimize their cost (Grouping). These groups of tests are called TGroups. Test cases that belong to different TGroups can be executed independently because the resources they employ are different. Finally, each TGroup may be split and allocated in several instances (Scheduling) to optimize both the cost/usage of resources and the test execution time. The test cases of these TGroups are split in several subsets of test cases, which are called TJobs, that contain not only the code of the test, but also the environment with the dependencies isolated in a container that allows easy deployment of the test in a cloud instance.

In the following subsections the above key concepts are detailed. Subsection 3.1 below defines the resources and their attributes. These resources can be used in different ways by the E2E test cases considering the access mode (Subsection 3.2) and the properties (Subsection 3.3). Finally, the processes that orchestrate the E2E test cases are defined in Subsection 3.4.

3.1 Definition of Resources and Test Jobs/Groups

The core of RETORCH is based on three main concepts defined below: the resources required by the test cases, the groups of test cases that use homogeneous resources (TGroups) and a partitioning of these TGroups so to isolate the dependencies in elastic environment and optimize resource usage through scheduling (TJobs).

1. **Resource**: Physical, logical and/or computational entities required by the execution of one or more test cases, for example, a web server requested by a test case or a table of database queried by the test case. The resources are characterized by the following attributes:

    a. **Elasticity**: A resource is elastic when it can be instantiated and made available for the tests cases on the fly (e.g. a database running in a container, a software simulator). Conversely, a resource is not elastic when only a fixed maximum number of instances is available (e.g. a sensor, a camera, a hardware emulator).
    b. **Hierarchy/partitioning:** A resource may contain sub-resources or partitions that also are resources (e.g. a database may be partitioned into several tables or sets, of tables).
    c. **Sharing:** Shared resources may be used simultaneously by more than one test case without interfering into the test result.
    d. **Lifecycle:** All resources have a lifecycle composed of three phases: set up of resource, test execution using the resource and disposal of the resource. In the Set-up phase, the resources are deployed and initialized according to the test data (e.g. initial load of the database, configuration data). Once the resources are ready, the

tests use these resources in the test execution phase according to the proposal. After finishing the test execution, the resources are disposed and released, making them available for other test cases.

2. **TGroup (Test Group)**: is a set of test cases that use homogeneous resources and can be deployed together in the same environment. For example, a TGroup can contain the test cases that query the same database with the same initial load and without modifying the information. These test cases can use the same database set-up in the same instance. In contrast, two test cases that both query and modify the database information can cause wrong test execution, and then these two test cases must be in different TGroups and deployed in different instances/environments. Each TGroup settles the environment needed by the test execution in the whole system or also considering scaffolding and test harness through the mocks, stubs or other simulators that can alleviate the cost of resources that are not mainly needed for the tests of the TGroup. The test cases of the TGroup can be also divided to not only optimize the cost/usage of resources but also the execution time through a distributed scheduling.
3. **TJob (Test Job):** is a partitioning of a TGroup containing several test cases inside a Docker container that also deploys the system under test isolating the dependencies and is customized to provide utilities to execute, monitor and collect testing information. Then one TGroup can be split into several TJobs and these can be scheduled them in different instances of the cloud, so to reduce the execution time by executing more TJobs in parallel thanks to the isolation of dependencies.

Example 1: Air Traffic Management (ATM). When testing the operations that an air traffic controller makes to manage their assigned flights, we need a resource that is the Control Working Position (CWP), which is itself a complex system, so as, it is not elastic. The CWP may become a shared resource if we partition the flight area into hierarchical clusters of sectors, provided that each test case will manage only flights belonging to a cluster. Moreover, when testing a transfer of flights between controllers will need two CWP, either exclusive or shared. This resource also has his own lifecycle, with a set-up (prepare all the CWP and flight plans), a test phase and finally a release and disposal.

3.2 Resource Access Modes

In order to group the test cases in TGroups and then schedule them in TJobs, RETORCH considers how the test cases must use the resources. Each test case can perform different kind of operations over the allocated resources. These operations can have two properties: **safety** and **idempotency.** Safe operations are those whose execution does not modify the resource, for example, a SELECT operation in a database query because it does not change the information of the database and does not introduce dependencies between test cases. Idempotent operations are those that can be performed several times consecutively producing the same result.

Different test cases may have different usage patterns when sharing the same resource. Each pair of test case and resource is associated according to an access mode

that determines if the operations performed during the test execution modify the resource or not, and how. The access modes are enumerated below:

- **Read-only:** the test case performs both safe and idempotent operations allowing other test cases read the resource at the same time (e.g. a test case that queries the master tables of a database without any change, allows that other test cases query the same resource).
- **Read-write:** the test case performs operations that are neither safe nor idempotent. Then, other test cases may not use this resource simultaneously to avoid unexpected erroneous executions (e.g., a communication channel, if we are not doing performance-related testing).
- **Write-only:** the test case performs operations that are neither safe nor idempotent similar to those "read-write", but allows that more than one test case update the resource simultaneously, restricting reads to only assertions that check the expected results (e.g., a centralized log system may be used simultaneously by several test cases, provided that, if we need to check the logs, there is a mechanism that allows identifying the logs produced by each test case).
- **Dynamic:** the test case performs operations that are safe but not idempotent. The resource is partitioned on the fly allowing that each test case create and access each partition independently from other test cases (e.g., when testing several test cases that issue orders, more than one test can place an order at the same time, but in dynamic access each test case must only use the orders that it has created).
- **No access**: This access mode is banally safe because the operations of the test case do not make use of any resource (e.g., when using a simple mock that does not requires any resource).

Example 2: In ATM, the Flight plans in an Air Traffic simulator are usually a shared dynamic resource, because they are created on fly as needed when the test is performed. On the other hand, the operation logs that are kept for legal requirements, are a write-only resource because they do not use it for anything other than saving the different usage traceback.

3.3 Resource Properties

The test cases can be grouped according to their access modes in the resources, but the test orchestration also considers other properties that allow the deployment and proper monitorization of both test cases and resources needed. Each pair of test case and resource is characterized by the following properties:

- **Allocated:** Location of each resource must be known to make possible their identification (e.g., the instances over where is deployed). Allocation is crucial when an effective use and measure of the resource performance during testing is considered.
- **Measurable:** Each resource must have indicators to allow measuring how many of them are deployed and their performance (e.g. RAM, processor usage or heartbeat latency received by a sensor network).

- **Traceability**: Each resource must be always traceable, allowing to know its state at every time of the test execution (e.g. ready, running, disposing of, or testing over it).
- **Elasticity Cost:** The elasticity cost measures the expenses incurred during the resource life cycle. This cost may be a combination of money, time, processing power, memory, energy, among others.
- **Test Environment:** The resources and test cases must be deployed in an environment that isolates the dependencies and avoids wrong executions/accesses with a properly set up. The whole environment can be encapsulated in a Docker container to be easily deployed in a cloud instance.

3.4 Processes

The behavior of RETORCH is organized around three different processes, namely Resource Identification, Grouping and Scheduling. Resource identification provides a first view of the resources required by the test cases. Then, a grouping is performed for later, optimizing the execution of those tests in a proper scheduling process. These processes, schematically represented in Fig. 2 are described below:

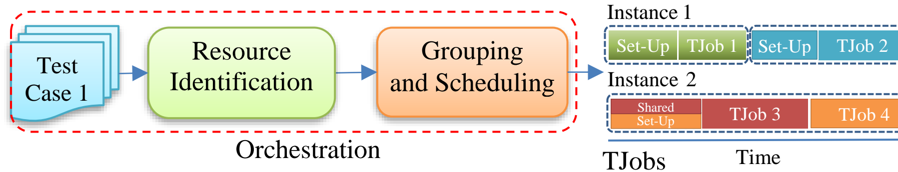

Fig. 2. Scheme of the main RETORCH processes

1. **Resource Identification:** In this process the resources that each test case needs to be executed properly are identified. To determine how the test case uses the resource, each association of a resource and test case is labeled with an access mode and the properties required by the test (Subsections 3.2 and 3.3). With the information about the resources, we can calculate the elasticity cost of the resources.
2. **Grouping:** To arrange the test cases according the resource usage, several groups are formed. The goal is to arrange the execution of the test cases based on their associated resources. For example, if two tests perform a safe access mode they can be grouped together. However, if two test cases perform a non-safe operation on the same resource, they are candidate to be placed in the same or separate groups depending on the access mode. The result of this grouping is a set of test cases put together with all scaffolding required for the execution. The objective of these groups is to avoid instantiation of a resource with more features than required and hence underutilized.
3. **Scheduling:** Although grouping achieves some optimization on the resource usage, the whole test process may be further optimized by ordering and splitting the TGroups taking into account the available infrastructure. For instance, TGroups may be distributed in parallel to achieve better use of the test infrastructure and reduce

the execution time. Not all schedules are aimed to minimize both execution time and the resource usage (one possible objective may be to maximize the usage of several instances, minimizing the idle time or another possible objective may be minimizing the execution time using more resources).

4 Working example

To illustrate the RETORCH approach, we present an example of its application on a real-world open-source application called FullTeaching [19]. FullTeaching is an educational platform that provides teachers with many features for organizing the teaching material, scheduling courses, and structuring classes; it provides also means for interacting with students, e.g., calendars, dashboards, forums. FullTeaching is a complex application involving several resources, including the OpenVidu Server [20], the Kurento Media Server [21], and the MySQL DBMS. In particular, for online teaching FullTeaching includes features enabling real-time video conferencing that are supported by OpenVidu via W3C Web-RTC [22] open source api. For the purpose of E2E testing such functionality, testers should take into account the underlying infrastructure and the re-source usage by OpenVidu.

For the several test cases that concern the OpenVidu Engine, deploying one instance for every TJob may be too expensive in terms of resource usage due to heavy resources for storage and graphical processing evolved by video streaming. For this reason, we divide the server in three types of resources with different usage rates exposed as follows:

1. **Minimal OpenVidu Server**: This type of server is a little mock that just provides a random number as session id, whenever any client requires it. Precisely, this resource has a No-Access mode meaning that the requests from the test do not access the real resource, it is a simple mock. It is used in those TJobs that only require this type of interaction with the OpenVidu Server,
2. **Medium Open Vidu Server:** The medium server is a light implementation of the service, with only basic functionalities and without any storage to record the session. This service will be employed in those test that require to check any functionality that needs a classroom or a simple chat between users.
3. **Heavy OpenVidu Server:** The heavy server provides all the functionalities of the OpenVidu Server, beside several video lessons recorded. This will be used in those test cases that require these video streaming recording functions or require all the functionality of the engine for his execution.

Using these three types of resources we proceed to classify all the test cases available depending on their resource usage requirements. For this we use the three phases of RETORCH, beginning with the resource identification, next the grouping and finally the scheduling.

Resource identification: In this stage we detect which type of OpenVidu engine is required for each test case. After collecting this information, we can proceed with the grouping phase.

Test cases assigned to a Light OpenVidu are the cheapest in term of elasticity cost: they can be available for testing on the fly and can be shared between multiple tests. The lifecycle of this resource is composed of the three usual phases, being the only without any cost.

Test cases assigned to a Medium OpenVidu Server require deploying a simple container that consumes a small amount of resources in terms of elasticity cost and allows sharing between multiple tests (although with some performance penalty). In this case, the set-up/dispose lifecycle phases do have a cost, so us the grouping performed later will try to share this set-up between several tests. Test on this resource has a Read-Write access mode.

Last, test cases assigned to Heavy OpenVidu Server should be executed in a sequential way due to the high elasticity cost that does not allow for deploying more than an instance. This resource has a read-write access mode because they employ and create the video into the resource.

Grouping: As described before, the next stage is to group those test cases that are compatible and can be executed with the same resource set up into TGroups. The deployment of the proposed RETORCH application for E2E testing of FullTeaching considering the OpenVidu resource is depicted in Fig. 3: we classify the Test Cases in heavy (Fig. 3 black color) medium (Fig. 3 red color) and light (Fig.3 blue color).

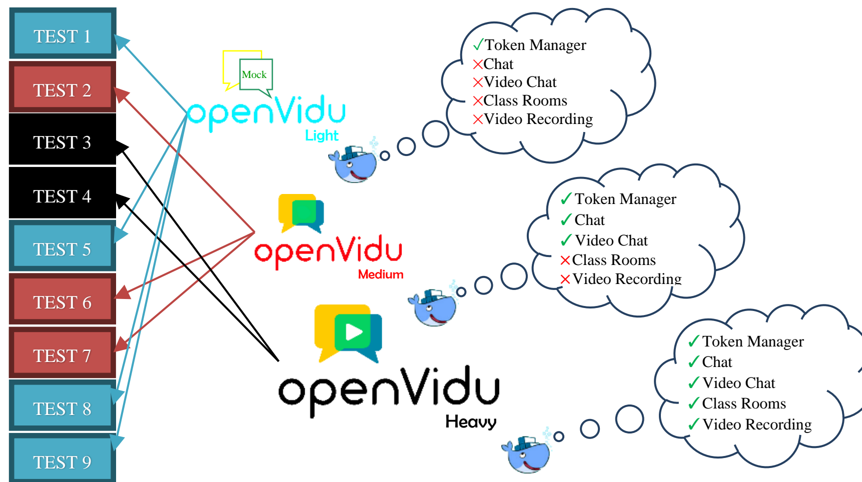

Fig. 3. Resource dependencies and grouping process

Let suppose that we have nine test cases and determine three TGroups as indicated below:

TGroup 1 (Light OpenVidu): Test case 1, 5, 8 and 9
TGroup 2 (Medium OpenVidu): Test case 2, 6 and 7
TGroup 3 (Heavy OpenVidu): Test case 3 and 4

In the most basic grouping, an instance is used by each TGroup to deploy the TJobs of the three types created in the scheduling phase (each type in one isolate instance). The TGroups include the test cases with their different scaffolding, avoiding the underutilization of a streaming server with more resources than needed.

Scheduling: Once the grouping is done we proceed to schedule the different TGroups to optimize the resource usage, execution time and execution cost. In this case, in Fig.4 we give four different examples of schedules, for each type of TGroups (assigned to a Light, Medium and Heavy resource):

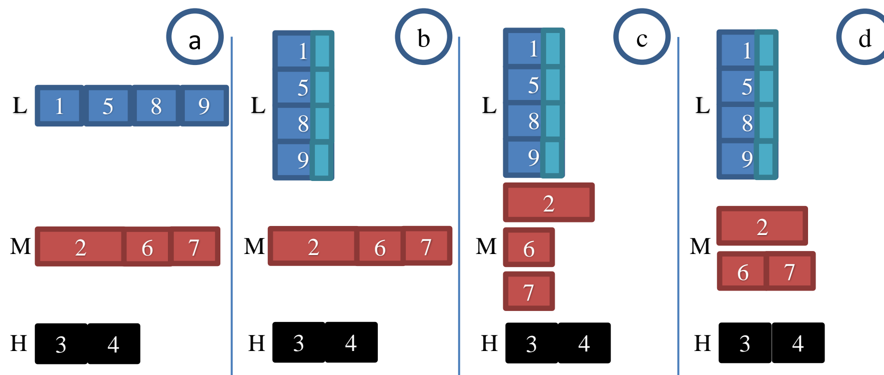

Fig. 4. Different TJob Scheduling examples depending on the objective

1. Fig. 4.a) represents a sequential execution of Light, Medium and Heavy OpenVidu servers. This scheduling provides a baseline, giving the worst execution time, but using the minimal number of instances required to keep the TGroups isolated. All TGroups are deployed in separate instances sharing the same set-up between them.
2. Fig. 4.b) depicts a similar configuration to the previous one concerning the Medium and Heavy OpenVidu server and instead the parallel execution in one instance of the Light one. The difference with the previous resides in lower execution time of the Light instances in addition to a small reduction of the time employed to execute all tests regarding the Scheduling represented in Fig. 4.a). Each Light OpenVidu TJob employs more time in the individual execution due to the overload caused by the concurrent access, but in this case, it is not relevant because the critical execution time corresponds with the Medium TJobs that mark the final execution of the Test phase.
3. In Fig.4.c) Medium TJobs are deployed in a parallel way into three instances because they cannot share the same instance. The performance of this scheduling is limited by the execution time of the heavy TJobs, and requires the high amount of resources represented in the figure. This type of scheduling improves the execution time of the test cases whereas it reduces the idle time of the instances which deploy light and heavy TJobs.

4. Last, Fig. 4.d) depicts a parallelization of the Medium TJobs in two separated instances. With this scheduling, the number of instances required by the TJobs are the lowest, providing a similar execution time due to the time employed by the execution of the Heavy TJobs. This is suitable when the execution time is the most valuable resource to improve and it depends on whether the elasticity cost of performing one extra instance of this type can be assumed.

As shown in this working example, there are several features and constraints that need to be considered for optimizing test scheduling based on resource usage. The critical step is the proper identification of which resources are needed by the test cases and their dependencies.

5 Conclusions and future work

This paper introduces a resource-aware E2E test orchestration proposal through: the identification of resources required to run an E2E test case, the grouping of the test cases based on the minimization of the resources to be deployed and on the parallel scheduling of the tests in several machines in order to optimize the resource usage and execution time. The framework proposed, RETORCH, is intended to be used with the ElasTest platform for Cloud test automation, to optimize the execution of the E2E test cases in terms of resource usage. In the paper, we provide a practical example to illustrate the different type of E2E test cases and potential scheduling that can be put in place with them. RETORCH is still under development and we will make it available when more mature. We expect that the several concepts we defined can improve the efficiency of E2E testing by achieving significant savings of resources usage.

There are several open questions that we can summarize in two main lines for future work. The first one is concerned with the resource identification process: we would like to automate the process allowing the detection of the dependencies between resources and test cases. This would require a comprehensive evaluation of these dependencies through tools or resource specification files. Last, the second line of work is focused on the development of one new orchestration method based on Grouping and Scheduling process to optimize in automatic way the resources employed in the E2E tests.

Acknowledgments

This work was supported in part by the Spanish Ministry of Economy and Competitiveness under TestEAMoS (TIN2016-76956-C3-1-R) project and ERDF funds, and by the European Project ElasTest in the Horizon 2020 research and innovation program (GA No. 731535).

References

1. Meyer, M.: Continuous Integration and Its Tools. Software, IEEE. 31, 14–16 (2014). https://doi.org/10.1109/MS.2014.58.

2. Yoo, S., Harman, M.: Regression Testing Minimisation, Selection and Prioritisation : A Survey. 60 (2007).
3. Fitzgerald, B., Stol, K.-J.: Continuous software engineering: A roadmap and agenda. Journal of Systems and Software. 123, 176–189 (2017). https://doi.org/10.1016/j.jss.2015.06.063.
4. Bertolino, A., de Angelis, G., Gallego, M., García, B., Gortázar, F., Lonetti, F., Marchetti, E.: A Systematic Review on Cloud Testing. ACM Computing surveys. (2019).
5. Bertolino, A., Calabró, A., De Angelis, G., Gallego, M., García, B., Gortázar, F.: When the Testing Gets Tough, the Tough Get ElasTest. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. pp. 17–20. ACM, New York, NY, USA (2018). https://doi.org/10.1145/3183440.3183497.
6. Harman, M.: Making the Case for MORTO: Multi Objective Regression Test Optimization. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops. pp. 111–114. IEEE, Berlin, Germany (2011). https://doi.org/10.1109/ICSTW.2011.60.
7. Bertolino, A.: Software Testing Research: Achievements, Challenges, Dreams. In: 2007 Future of Software Engineering. pp. 85–103. IEEE Computer Society, Washington, DC, USA (2007). https://doi.org/10.1109/FOSE.2007.25.
8. Rothermel, G., Harrold, M.J., von Ronne, J., Hong, C.: Empirical studies of test-suite reduction. Software Testing, Verification and Reliability. 12, 219–249 (2002). https://doi.org/10.1002/stvr.256.
9. Wong, W.E., Horgan, J.R., London, S., Mathur, A.: Effect of Test Set Minimization on Fault Detection Effectiveness. In: Proceedings - International Conference on Software Engineering. pp. 41–41 (1995). https://doi.org/10.1002/(SICI)1097-024X(19980410)28:4<347::AID-SPE145>3.0.CO;2-L.
10. Engström, E., Skoglund, M., Runeson, P.: Empirical Evaluations of Regression Test Selection Techniques: A Systematic Review. In: ESEM'08: Proceedings of the 2008 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 22–31 (2008). https://doi.org/10.1145/1414004.1414011.
11. Bell, J., Kaiser, G., Melski, E., Dattatreya, M.: Efficient Dependency Detection for Safe Java Test Acceleration. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 770–781. ACM, New York, NY, USA (2015). https://doi.org/10.1145/2786805.2786823.
12. Gyori, A., Shi, A., Hariri, F., Marinov, D.: Reliable Testing: Detecting State-polluting Tests to Prevent Test Dependency. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. pp. 223–233. ACM, New York, NY, USA (2015). https://doi.org/10.1145/2771783.2771793.
13. Gambi, A., Bell, J., Zeller, A.: Practical Test Dependency Detection. In: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST). pp. 1–11 (2018). https://doi.org/10.1109/ICST.2018.00011.
14. Gambi, A., Gorla, A., Zeller, A.: O!Snap: Cost-Efficient Testing in the Cloud. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST). pp. 454–459 (2017). https://doi.org/10.1109/ICST.2017.51.
15. Chakraborty, S.S., Shah, V.: Towards an approach and framework for test-execution plan derivation. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). pp. 488–491 (2011). https://doi.org/10.1109/ASE.2011.6100106.
16. Yu, L., Su, Y., Wang, Q.: Scheduling Test Execution of WBEM Applications. In: Proceedings - Asia-Pacific Software Engineering Conference, APSEC. pp. 323–330 (2009). https://doi.org/10.1109/APSEC.2009.27.

17. García, B., Lonetti, F., Gallego, M., Miranda, B., Jiménez, E., Angelis, G.D., Santos, C., Marchetti, E.: A Proposal to Orchestrate Test Cases. In: 2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC). pp. 38–46 (2018). https://doi.org/10.1109/QUATIC.2018.00016.
18. Esfahani, H., Fietz, J., Ke, Q., Kolomiets, A., Lan, E., Mavrinac, E., Schulte, W., Sanches, N., Kandula, S.: CloudBuild: Microsoft's Distributed and Caching Build Service. In: Proceedings of the 38th International Conference on Software Engineering Companion. pp. 11–20. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2889160.2889222.
19. Pérez, P.F.: A web application to make teaching online easy. Contribute to pabloFuente/full-teaching development by creating an account on GitHub. (2019).
20. OpenVidu, https://openvidu.io/.
21. Kurento, https://www.kurento.org/.
22. WebRTC Home | WebRTC, https://webrtc.org/.