Cloud Detours: A Non-intrusive Approach for Automatic Software Adaptation to the Cloud

Paulo Maia, Michel Vasconcelos, Nabor C. Mendonça

Cloud Detours: A Non-intrusive Approach for Automatic Software Adaptation to the Cloud

Michel Vasconcelos[1], Nabor C. Mendonça[1], and Paulo Henrique M. Maia[2]

[1] Programa de Pós-Graduação em Informática Aplicada (PPGIA)
Universidade de Fortaleza (UNIFOR), Fortaleza, CE, Brazil
michel.vasconcelos@gmail.com, nabor@unifor.br
[2] Centro de Ciências e Tecnologia (CCT)
Universidade Estadual do Ceará (UECE), Fortaleza, CE, Brazil
pauloh.maia@uece.br

Abstract. A major challenge facing cloud migration is the need to change a legacy (on-premise) application's source code so that it can better benefit from the inherit cloud computing characteristics, such as resource elasticity and high scalability. When performed manually, those changes are error-prone and may require a great effort from application developers. This paper presents a novel approach to support organizations in automatically adapting their existing software applications to the cloud. The approach is based on the loosely-coupled implementation of non-intrusive code transformations, called *cloud detours*, which enable the automatic replacement of local services used by an application with similar or functionally-related services available in the cloud. To illustrate the approach, the paper reports on how an initial set of cloud detours, implemented using aspect-oriented programming and a generic cloud library, was used to seamlessly adapt an existing file-based Java application to save application data in a cloud-based storage service.

1 Introduction

Despite the several advantages commonly associated with the cloud computing paradigm, e.g, greater control over operational costs, the illusion of infinite resources, high scalability, and self-service on demand [3], in practice many organizations have found it difficult to use cloud-based solutions, particularly when faced with the need to migrate existing legacy applications to public cloud providers [14]. As opposed to cloud *adoption*, which means that an organization will use cloud resources and technologies to develop new cloud-native applications, the term cloud *migration* implies that the organization already has existing software that must somehow be adapted to better suit (or to better benefit from) the target cloud platform [11].

Cloud migration decisions are inherently complex since they are influenced by multiple, possibly conflicting factors, such as cost, performance, security and legal concerns [4]. In addition, applications developers must carefully consider possible technical restrictions that may hinder (or even prevent) the migration process, such as when the legacy application relies on implementation technologies

that violate environmental constraints imposed by the target cloud platform [7]. Another challenge consists of modifying or adapting the legacy application such that it can take advantage of available cloud services and resources, for instance, by replacing an on-premise relational database by a cloud-based NoSQL storage service, a form of adaptation commonly referred to as *cloudification* [2, 13].

Although some recent work on automatic cloud conformance-checking [7] and the systematic classification of cloud migration types [10] and patterns [2, 13] have started to partially address those challenges, there is still a lack of (semi) automated tools to support the cloud migration process [11]. This limitation implies in a more complex and error-prone migration effort, since the necessary source code changes have to be performed manually by the developer, requiring both a deep understanding of the software's internal structure as well as a detailed knowledge of the target cloud's libraries and APIs. Even in the cases in which the necessary software adaptations can be fully or partially automated, such as in the work described in [12], those are usually performed intrusively, by directly changing the legacy application's source code. As a consequence, the adaptation code becomes tightly coupled to the specific cloud resources and libraries used when performing the changes, making it harder for the developer to reuse the adaption code across different applications as well as to evolve the adapted application to use different cloud services and providers.

This paper presents a novel approach to support the automatic adaptation of legacy (on-premise) applications to the cloud. The proposed approach is based on the modular specification, implementation and reuse of non-intrusive code transformations, called *cloud detours*, which enable existing legacy applications to use existing cloud resources and services seamlessly, without the need to change their original source code directly. The approach is implemented by an event-based framework that decouples the adaptation mechanism that is non-intrusively injected into the application source code, from the cloud-specific libraries and APIs used to invoke the target cloud services. In this way, the chosen adaptation mechanism and cloud libraries can evolve independently, giving the developer more freedom to reuse the adaption code in other applications sharing the same development or execution environment as well as to experiment with different cloud technologies. This approach can be particularly useful during the early stages of the migration process, when comparing the services offered by different cloud providers may play a key role in helping individual and organizations in making informed cloud adoption decisions [4].

The remainder of the paper is organized as follows. Section 2 compares our approach with related work. Section 3 presents the main concepts behind the proposed approach, while Section 4 describes its supporting event-based framework. Section 5 illustrates the feasibility of the approach by reporting on the successful use of our framework to seamlessly adapt an existing file-based Java application to save application data in a cloud-based storage service. Finally, Section 6 provides some conclusions and directions for future work.

2 Related work

Jamshidi *et al.* have recently presented a systematic literature review where they discuss and compare several cloud migration strategies and techniques [10]. To this end, the authors introduce the *Cloud Reference Migration Model* (Cloud-RMM), which provides a conceptual basis to classify existing cloud migration approaches according to three main migration tasks, namely *planning*, *execution*, and *evaluation*. In addition, the authors consider complimentary approaches that address managerial issues, such as governance, effort estimation and risk analysis, as crosscutting concerns of the model. Our adaptation approach fits within the execution task of Cloud-RMM and addresses one of the main challenges identified by that study, which is to offer automated support for the cloud migration process [10].

Early works on cloud migration have focused on automatically detecting potential incompatibilities between the legacy application and the target cloud environment [6], on model-based transformation of legacy applications in to cloud services [14], and on providing high-level process support for cloud migration [4]. Another related work in this direction is an architecture-centric migration framework which includes pre-migration tasks and decisions, such as the development of a migration plan [1]. Our work on cloud detours can be seen as complementary to those works, as it provides a flexible, non-intrusive way to implement the necessary adaptations in the source code of the applications being migrated.

In another related research line, Andrikopoulos *et al.* have identified four migration types, namely, *replace*, *partial migration*, *migrate the whole execution stack*, and *cloudify*, according to the different application layers and adaptation levels required to make the migration possible [2]. In a similar fashion, Mendonça has identified two main migration strategies, namely *cloud hosting* and *cloudification*, with the former representing the case in which some (possibly modified) application components are hosted in the cloud and the latter the case in which those components are replaced by functionally-related cloud-based services [13]. In the context of those works, our adaptation approach follows the *cloudification* strategy, based on the *replace* migration type, since it non-intrusively transforms the original application source code to replace some of its original components with equivalente services in the cloud.

Finally, Kwon and Tilevich have propose the concept of *cloud refactorings* [12], which are code transformations used to automatically integrate on-premise applications to cloud-enabled services. The proposed code transformations are implemented by means of an IDE plugin and a recommendation tool based on static analysis and runtime monitoring of the application being migrated. Differently from our work, cloud refactorings follow a fully intrusive adaptation approach, since the refactoring tool changes the original application source code directly. As we have discussed previously, this approach makes it harder for the developer to experiment with different adaption mechanisms and cloud technologies, as those are hardcoded in the implementation of the cloud refactoring plugin.

3 Cloud Detours

Our cloud adaptation approach is based on the assumption that the change sets required to implement the desired software adaptions should be grouped into reusable assets called *cloud detours*. The idea is that developers could reuse these assets across different applications and execution environments, thus reducing the overall time and effort involved in the migration of existing (on-premise) applications to the cloud.

More specifically, a *cloud detour* is a non-intrusive reusable artifact containing source code change sets necessary to adapt applications to be hosted in a cloud environment or to use available cloud services. Developers may use cloud detours to automatically adapt on-premise applications, thus avoiding the risks and drawbacks involved when modifying the source code directly. Besides accelerating the migration process itself, cloud detours also can be useful to gradually adapt different parts of the application before fully migrating the whole application to the cloud.

Due to its non-intrusive design, a cloud detour needs to be aligned with the architecture and technologies of the local and target environments. In our work, we focus on cloud detours that are designed to adapt multi-layered software applications, by replacing services at a certain application layer with equivalent services in the cloud.

In a multi-layered application, each layer provides a set of services for the upper layers. This increases modularity and reduces the coupling between application components. Cloud detours benefit from this design by overriding local application services with services provided by the target cloud environment at execution time.

Cloud detours also can be defined in terms of elements that are external to the application. For instance, in a web application that is hosted by an application server that provides transaction control and persistence services, a cloud detour can be used to replace those services by similar ones in the cloud. In our work, we call such basic services and elements as *operating services*.

Figure 1 depicts the cloud detour architectural model. The dashed arrows highlight the different architectural levels at which a cloud detour can be used to adapt an application. Note that, depending on the chosen level, a cloud detour can be used to replace services that are either internal or external to the application being adapted. Choosing a proper adaptation level is important since each level restricts the suit of adaptation technologies as well as the context information available for implementing detours.

At the *application level*, a detour's adaptation logic can be implemented in terms of the components, patterns and technologies being used by the application itself. When defined at that level, a detour can access source code elements like units, classes, methods and parameters, and can be implemented using source or binary code instrumentation mechanisms, such as aspect oriented programming and meta-programming. As an example, we can cite a detour to adapt a Java application that initially accesses data through a certain (internal) local

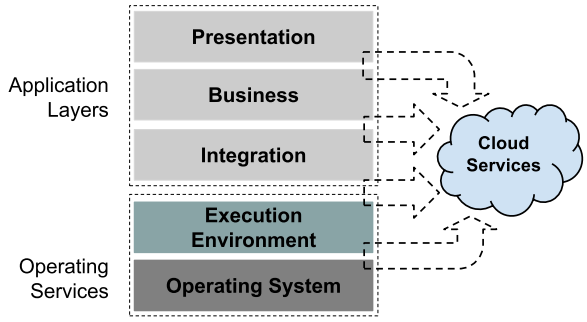


Fig. 1. Application architectural model and detour opportunities.

data service and, after adaption, replaces that service with a similar cloud-based storage service.

At the *operating level*, a detour's adaptation logic can be implemented in terms of elements belonging to the underlying execution environment or operating system (e.g., virtual machine or sandboxes APIs, low-level libraries). The context information available are system calls, libraries and environment variables. Implementing cloud detours at that level requires non-code based interception mechanisms like library overloading and function interposition.

Cloud detours are usually much easier to implement at the application level than at the operating level. This is because application level detours can be defined in terms of syntactic elements that are clearly visible in the application source code, while operating level detours require external system knowledge of the underlying execution platform that are not easily accessible to most application developers. On the other hand, application level detours tend to be less reusable as they rely on structural and contextual information that may be too specific to a given application.

Considering the same adaptation scenario described above, adapting a new Java application to use the same cloud-based storage service would require a new detour, as the implementation of the original detour would be too tightly coupled to the source code elements of the original Java application.

4 Cloud Detours Framework

The *Cloud Detours Framework* (CDF) provides a library of detours as well as the needed backbone to deploy them as part of the execution flow of existing legacy applications. This section describes the CDF design and its current implementation as a *proof-of-concept* for our cloud adaptation approach.

4.1 Domain Model

Figure 2 depicts the DCF's domain model.

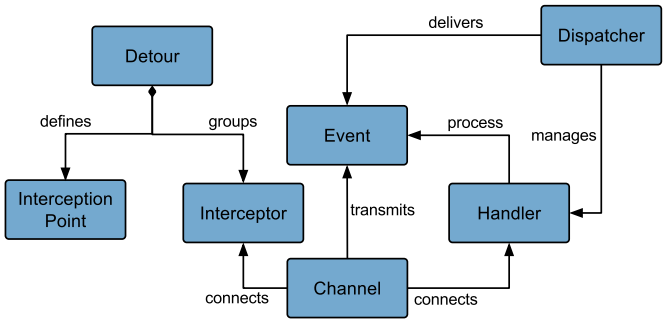


Fig. 2. Cloud Detours domain model

Detour, as described in the previous section, is the key entity that comprises the necessary change sets to adapt an application to interact with a cloud service. It can be extended according the category of services it replaces (e.g., I/O detour, database detour, messaging detour). Each detour defines a set of hot spots called *interception points* that describe the execution points where interceptors are expected to be injected into the application flow. *Interceptors* are responsible for catching the application execution flow and forwarding configured actions as events. Together, detours, interception points and interceptors comprise the code instrumentation mechanism provided by the CDF.

Event is the main entity responsible for interchanging data among the framework's front-end and back-end components. It can be extended according to the action types it encapsulates (e.g, I/O operation, database access). *Channels* interconnect interceptors and adapters and provide a safe path to send and receive events. *Adapters* connect the framework's back-end components to the target cloud and map the actions triggered in the application to available cloud service operations. Finally, *Dispatchers* manage the life cycle of adapters and distribute incoming events according to the tuple `<event, channel, adapter>`.

4.2 Architecture Overview

When adapting the system to interact with the cloud, the developer must consider different abstraction levels of services and features, such as modifying application source code, exception handling, resource allocation and bindings to external services. Although performing all those adaptations in a single element is possible, this would make it a into a high complex and poor maintainable software artifact. The CDF design addresses this issue by separating the concerns regarding application interception from those related with cloud interaction. To this end, the CDF relies on an event-based layered architecture to decouple those concerns and to allow different levels of abstraction to be softly handled.

Architectural Layers The CDF architecture is organized in multiple layers to promote low coupling and high cohesion amongst its components, as well as to

allow components of different levels of abstraction to evolve independently. The framework layers are described below:

Interception The Interception layer interfaces with the on-premise application or the local environment to capture actions defined as interception points so that the application's original execution flow is detoured. Detours are "first-class citizens" of this layer and are implemented through instrumentation techniques like aspect-oriented programming and function interposition. Regarding the abstraction level, detours can be classified as (i) *abstract*, when they are generic and not bound to any particular application; or (ii) *concrete*, when they are tailored to specific applications or configured to catch particular actions. Besides instrumentation, detours also handle other concerns, such as binding code and exception handling, which are implemented through plain classes, units or other resources available in the local environment;

Transmission This layer works as the central hub that connects the Interception and Adaptation layers. It provides event distribution, component mediation, location transparency, and mux/demux services. It also is responsible for physical and logical decoupling among the other layers such that they could be implemented and evolved using patterns and technologies that best suit their respective needs. Events and channels are key elements of this layer;

Adaptation This layer provides access to cloud resources and services. Dispatchers and adapters, its main elements, are responsible for handling incoming events, mapping actions, and integrating with the target cloud. Adapters are components that execute incoming events as actions. They also collaborate with channels to provide a service abstraction to the Interception layer. Actually, this layer provides a service descriptor by supplying location (channels), data typing (events) and operation (adapters) to possible clients. A dispatcher accounts for components coordination in the Adaptation layer. It manages adapter life cycle, identifies incoming events and associates channels with respective adapters.

Build and Assembly The elements of this layer cooperate with elements of the other layers to accelerate the configuration, building and deployment of the CDF components in the (local) application environment. The main elements of this layer are build tools, shell scripts and configuration files.

Event-Reactor Pattern The Event-Reactor pattern is used in the CDF as a decoupled way to process events triggered by interceptors while preserving the latency and synchronism required by the adapted application [15]. This pattern is implemented through components of the Transmission and Adaptation layers.

4.3 Implementation Details

The CDF design enables the development of its internal components using programming languages and technologies that best suit their purpose. In its current

version, interception components, detours and helper classes are implemented in AspectJ and Java. Transformation and building mechanisms are realized through Groovy and Gradle[3]. Cloud Detours back-end elements, adapters, dispatchers and execution infrastructure are implemented in Python. In the following we detail some CDF implementation decisions. The CDF source code, including its documentation, is publicly available at `https://github.com/michelav/cloud-detours`.

Abstract and Concrete Detours Under the development perspective, detours are coarse-grained components that comprise aspects, helper classes (interceptors), general scripts and configuration files. Detours weave interceptors, implemented as plain Java classes, into the application through aspects. Each detour declares its own building and usage rules, therefore making them highly cohesive. This reduces the effort needed to extend the framework.

One important flexibility point of Cloud Detour resides on the relation of abstract and concrete detours. Abstract detours determine life cycle, interface usage and general behavior for all of its concrete detours. However, they cannot be instrumented alone since they lack interception points. Concrete detours cooperate with abstract ones by providing application-specific interception points and complementary behavior. Consequently, one must provide a suitable concrete detour in order to address a different on-premise application.

When instrumenting an application, the detours behave simultaneously as factories [8], by instantiating interceptors, and as a dependency injection mechanism [5], by injecting interceptors in the application seamlessly.

Figure 3 shows the source code for an excerpt of the IO detour provided by the framework. The abstract pointcut `outputStreamAP` defines the hotspot developers should configure in concrete detours. For instance, if someone needs to deviate all output actions incoming from `foo.bar` package, she should create a concrete detour and set it up as `pointcut outputStreamAP():within(foo.bar.*)` .

Cloud Detours Back-End Events, channels and adapters are packaged together with the framework back-end. Events are implemented as plain Java classes and Python dictionaries.

Channels are developed as plain classes that provide send and receive primitives with the purpose of interconnect detours and adapters. Detours channels delegate low-level transmission procedures to *ZMQ* communication library. ZMQ provides asynchronous communication, concurrency control and several communication patterns (point-to-point, multipoint, pub/sub, broker, etc.) to its clients [9]. Each channel contains a *ZMQ Socket* that physically connects the application to the framework back-end.

Cloud Detours back-end may be deployed in a single **remote** area, accessible through local network and serving multiple applications at the same time, or in **local** one, executing in the operating system that hosts the application. The

[3] `http://gradle.org`

```
public abstract aspect AbstractIODetour {

  pointcut fileOutput(File f):
    call(public FileOutputStream.new(File)) && args(f);

  abstract pointcut outputStreamAP();

  FileOutputStream around(File f):
    outputStreamAP() && fileOutput(f) {
      DetFileOutputStream dfos = new DetFileOutputStream(f);
      dfos.configureChannel(channel);
      // Injecting new service
      return dfos;
  }
}
```

Fig. 3. Abstract IO Detour (code excerpt)

system architect must evaluate variables like quantity of application to be enable to cloud, solution complexity, event payload size and network latency in order to define the best deployment method to be used.

4.4 Adaptation Process

Cloud Detours adaptation process comprises stages that demand local system and target cloud evaluation, selection and extension of detours and reconstruction of previous application as a new binary artifact. Figure 4 draws the process that is described as the following steps:

(1) Evaluation Adaptation scope is defined during this activity. Development team collects architectural information of the system, its logical (packages, components and classes) and physical organization (tiers, protocols, etc). Considering this information and the target cloud, developers select cloud services to be used and identify possible restrictions. The team defines which events will be intercepted and maps them to the available services. In case there is not a detour capable of intercepting an event, the developers may implement a new one. Finally, the development team decides the interception points and the application level of the detours.

(2) Extension In this step, *Cloud Detours* is extended to instrument the on-premise application. Abstract detours are extended according to the category of events and interception points defined in previous activity. Detours building and assembly scripts are configured to execute in the local environment. At the end of this step, concrete detours are created and ready to be applied.

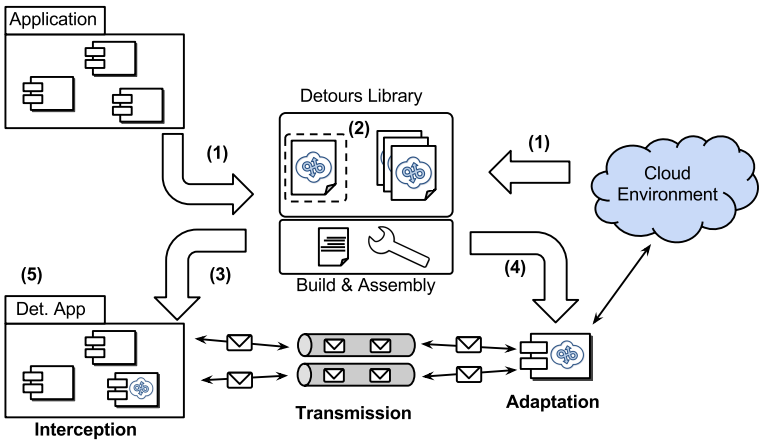


Fig. 4. CDF adaptation process

(3) Transformation This step provides a new application binary to be deployed. One must use the building and assembly mechanism, configured earlier, to generate a new deployment asset corresponding to the on-premise application weaved by selected concrete detours.

(4) Configuration *Cloud Detours* infrastructure is configured and loaded. At this moment, developers define the channels distribution and back-end deployment method as described in section 4.3. Channels must be created and assigned prior to deployment. At last, the adapters are configured (e.g.: endpoints definition, authentication and authorization issues, service addresses, etc) to work correctly with the target cloud.

(5) Deployment The instrumented application is installed in the environment. The developers follow the normal directives to install the application in local environment.

The next section illustrates this process and shows an example of use describing how CDF can be applied in a third-party application.

5 Example of use

In this example of use, we evaluate the *Cloud Detours* in a practical scenario of adaptation. During the process, we have also investigated the effects of the framework in the adapted application. For this, we have selected the IO detour available in the *Cloud Detours* framework and have measured its performance by comparing the time for executing IO operations before and after using the framework.

The remainder of this section details the tasks performed in the evaluation process and discuss the found results.

5.1 Tools selection

The following tools have been selected to this example of use: JSpider[4] as the local application (the one that will be migrated), and Google Storage[5] as the cloud storage service.

JSpider: JSpider is a Java configurable engine to download and store web sites. It is commonly used to find errors and to perform structural analysis in web sites. Its architecture is designed as a set of plugins organized in layers that allow the creation of new tasks. Finally, JSpider makes available a embedded command line tool. In this example, we address the application components that save the web site files in the local storage area.

Google Storage: It provides an object-based storage area via Internet that offers high availability, data replication and protection using OAuth[6], an open authorization standard, as access control. The service users can organize their objects in containers, called *buckets*. An object is an data agglomerate submitted to Google Storage.

In our experiment, each file downloaded by JSpider is submitted as an object to the Google Storage. Although JSpider organizes its information as files and directories, Google Storage does not provide a file system abstraction. Hence, the framework adapter has to map the concepts.

5.2 Evaluation method

To evaluate the performance of the solution, we have measured the necessary time for JSpider downloading two web sites that were locally hosted in order to eliminate the effects of the external network latency. The first one ($[S_1]$) is formed by several HTML pages containing links for internal and external pages, all of them summing up 55KB. The second one ($[S_2]$) is bigger (5MB) and has few HTML pages, but several text and binary files, varying from 500KB to 1000KB.

We can describe the scenarios evaluated in the experiment as:

- **Local application ($[C_1]$):** the selected application runs with no detours and stores all of its data in the local file system. This first scenario establishes a base line to be compared by the other tests;
- **Detoured application ($[C_2]$):** a new version of the application is generated after applying the *Cloud Detours* framework. However, we have used a special adapter that continues saving the application data in the local file system instead of sending them to the service in the cloud. This situation allows us to assess the cost of using *Cloud Detours*;

[4] available at `http://j-spider.sourceforge.net`
[5] available at `https://cloud.google.com/storage`
[6] available at `http://oauth.net`

- Adapted application ($[C_3]$): a new version of the application is integrated to an adapter that uses the cloud storage services. This scenario allows us to assess how the application behaves after being adapted to the cloud.

The following measurements have been performed in the experiment:

- Tm_i, average scenario execution time $[C_i]$ for all iterations (excluding failures);
- Cd_i, detour cost for the web site $[S_i]$ calculated by $Tm_2 - Tm_1$; and
- L_i, cloud latency for the web site $[S_i]$ calculated by $Tm_3 - Tm_2$.

5.3 Experiment Details

Following the process described in the previous section, we have inspected JSpider's code and identified that the *DiskWriter* plugin is responsible by performing the application's local disk writing operations. It is implemented in the `DiskWriterPlugin` class and is based on the classic file-related classes `File` and `FileOutputStream` of the `java.io` package to store the files. This way, the `DiskWriterPlugin.writeFile(File, InputStream)` method has been defined as the interception point to be configured in the concrete cloud detour.

After the inspection, we use the IO abstract detour, contained in the *Cloud Detours* library, as the interception base element and generate a new concrete detour defining its coverage area according to the desired package, class and method. In the implementation of the concrete detour, we have chosen to use a buffer so the data of each file is sent only after its last update. This strategy aims at reducing (i) the complexity involved in syncing the memory-stored file and its respective cloud object; and (ii) the cost of sending the file data.

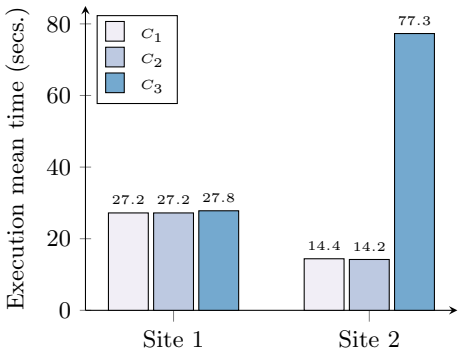
We have set up the cloud detour process to be executed locally and to be accessible via interprocess calls (ZMQ sockets). Thus, we not only reduced the solution complexity, but also eliminated the local network latency while the events were forwarded to the adapters.

To execute the tests, we have created a Python script that iterates for each test scenario and register its execution time in a CSV file.

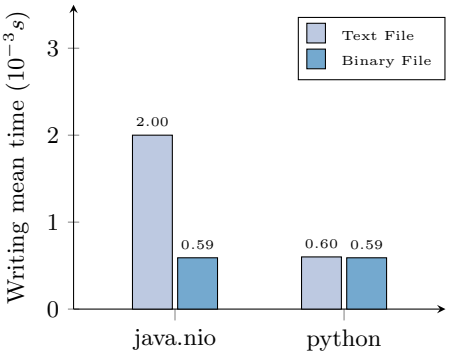
5.4 Results

Figure 5 (a) exhibits the experiment average execution time as the result of 30 iterations for each scenario. We can see that the average execution time of scenarios 1 and 2 were greater for web site 1 than web site 2, event though web site 1 is smaller. This happened due to the different configurations of each scenario. S_2 contains only a single HTML page to be processed, while S_1 has several ones. Therefore, this difference in time is caused by JSpider's own page processing mechanism.

The cloud latency for the second web site (L_1) was very small and the maximum value reached by one of the samples was 1,2 seconds. On the other hand, L_2



(a) Test cases execution time



(b) APIs writing time

Fig. 5. Experimental results

grows considerably due to the submission of S_2 bigger files. In both situations, the web sites integrity was preserved.

By assessing the detour costs (Cd_i), it was negligible for for C_1 and C_2 (both less than $1s$). We have also observed that in S_2 the average execution time of the detoured application is less than the one of the original application ($Cd_2 < 0$). By analysing the values for each sample, we have verified that the ones in the set (C_1, S_2) had their execution time greater than $(C2, S_2)$, which means that the detoured application run faster than the original one. Regarding the second web site, we had (C_2, S_1) faster than (C_1, S_1) in 46% of the samples.

To figure this out, we performed a comparison among the writing operations of the Java IO (`java.io` and `java.nio`) and Python APIs . For this, we used those APIs to write both a binary and a text file in 30 iterations. The writing average time of the `java.io` API was 1,48 seconds for the text file and 1,47 seconds for the binary file. Both `java.nio` and Python I/O API performed significantly better than `java.io` as shown by Figure 5 (b).

Therefore, we conclude that, due to the low performance of the `java.io` API, the adaptation of the application using *Cloud Detours* improved the execution time of JSpider during the experiment.

6 Conclusion and Future work

This work proposes a new approach for adapting legacy applications to the cloud, called *Cloud Detours*, that is based on non-intrusive software transformations. It uses the concept of *Cloud Detours* to intercept the normal application flow and replace the its provided services by cloud-based equivalent services. A framework that implements the proposed approach is also shown. Its architecture uses the Layers and Reactor patterns to make the adaption independent of interception technology and cloud environment.

Through an example of use in which the framework was used to adapt an application that uses local storage service to a cloud-based one, it was possible to observe that *Cloud Detours* did not affect the correct execution of the application. Furthermore, it even improved the application execution time, since the adaptation mechanism used a file writing library more efficient than the original one.

As future work we plan to extend the available detours provided by the framework including new event categories, such as database services. In addition, we plan to conduct new experiments to assess the necessary effort to set up and use the framework and to perform new performance measures.

References

1. Ahmad, A., Babar, M.A.: A framework for architecture-driven migration of legacy systems to cloud-enabled software. In: Proceedings of the WICSA 2014 Companion Volume. pp. 1–8. WICSA '14 Companion, ACM (2014)
2. Andrikopoulos, V., et al.: How to adapt applications for the cloud environment – challenges and solutions in migrating applications to the cloud. Computing 95(6), 493–535 (Jun 2013)
3. Armbrust, M., et al.: A view of cloud computing. CACM 53(4), 50–58 (2010)
4. Beserra, P., et al.: Cloudstep: A step-by-step decision process to support legacy application migration to the cloud. In: Proc. IEEE 6^{th} International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA). pp. 7–16 (2012)
5. Fowler, M.: Inversion of control containers and the dependency injection pattern (Jan 2004), `http://www.martinfowler.com/articles/injection.html`
6. Frey, S., Hasselbring, W.: The cloudmig approach: Model-based migration of software systems to cloud-optimized applications. Int. J. Advances in Software 4(3/4), 342–353 (2011)
7. Frey, S., et al.: Automatic conformance checking for migrating software systems to cloud infrastructures and platforms. J. Software: Evolution and Process 25(10), 1089–1115 (2013)

8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
9. Hintjens, P.: ZeroMQ: The Guide. Internet draft (2010), `http://zguide.zeromq.org/page:all`
10. Jamshidi, P., et al.: Cloud migration research: A systematic review. IEEE Trans. Cloud Comp. 1(2), 142–157 (Jul/Dec 2013)
11. Jamshidi, P., et al.: Cloud migration patterns: A multi-cloud service architecture perspective. In: Proc. 10^{th} International Workshop on Engineering Service-Oriented Applications (WESOA) (2014)
12. Kwon, Y.W., Tilevich, E.: Cloud refactoring: Automated transitioning to cloud-based services. Automated Software Engineering 21(3), 345–372 (2014)
13. Mendonça, N.C.: Architectural options for cloud migration. IEEE Computer 47(8), 62–66 (Aug 2014)
14. Mohagheghi, P., Sæther, T.: Software engineering challenges for migration to the service cloud paradigm: Ongoing work in the remics project. In: Proceedings of the 2011 IEEE World Congress on Services. pp. 507–514. SERVICES '11, IEEE Computer Society (2011)
15. Schmidt, D.C., et al.: Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects. John Wiley & Sons, Inc., New York, NY, USA, 2^{nd} edn. (2000)