Dynamic Program Code Distribution in Infrastructure-as-a-Service Clouds

Rostyslav Zabolotnyi, Philipp Leitner, Schahram Dustdar
Distributed Systems Group
Vienna University of Technology
Argentinierstrasse 8/184-1, 1040 Vienna, Austria
rstzab@infosys.tuwien.ac.at, leitner@infosys.tuwien.ac.at, sd@infosys.tuwien.ac.at

***Abstract*—Elastically scaling cloud computing applications are becoming more and more prevalent in today's IT landscapes. One problem of building such applications in an Infrastructure-as-a-Service cloud is the runtime distribution of program code, configuration files and other resources. While it is possible to include all required program code in the used IaaS base images, this severely restricts the achievable dynamicity at runtime. In this paper, we present a framework for dynamic program code distribution. Our approach handles code distribution entirely transparently on middleware layer. We base our solution on an existing middleware, CloudScale. The paper discusses the design and implementation of our code distribution approach on top of CloudScale, and numerically evaluates the practicability and performance of the approach based on an illustrative case study.**

I. Introduction

In the last years, the advancement of cloud computing [16] has transformed the entire IT industry, and has given new opportunities and abilities to developers and users. Moreover, cloud computing simplifies the implementation of innovative ideas for small companies or individuals, and lowers production and maintenance costs for industrial applications [1]. Because of cloud computing, elasticity of resources becomes a feature of applications instead of a description of data centers. The notion of elasticity for cloud applications morphs into elasticity of cost, resource and quality [10], giving additional dimensions and abilities for developers to optimize application and service provisioning.

Cloud computing is a promising technological choice for new application development projects, but if an application already exists, migration costs have to be considered before the advantages of the cloud can be leveraged. Cloud migration is known to be a challenging problem [12]. Often, migration reveals architectural problems and requires refactoring or redesigning of applications. However, even when the application architecture already fits the cloud computing paradigm, some amount of additional work is required in order to let the application fully benefit from the cloud.

The most basic cloud service model is the Infrastructure-as-a-Service (IaaS) approach [16]. On this level, cloud service providers offer virtual machines with requested configuration and operation system (usually in a form of hard drive image) to satisfy application computation requirements [5]. This layer is preferable for cloud migration as it requires less migration effort (and is better standardized) than Platform-as-a-Service [16] (PaaS). When an IaaS application has to scale up (i.e., use more virtual machines than before), one problem is how the availability of the current version of the application code, configuration files and other resources can be ensured on the new host. In the following, we will use the term "program code" as shorthand for the application code and all dependent files. The trivial approach is to include this program code in the virtual machine base image, but this approach is reasonable only in situations when it is entirely static and will not be modified during application lifetime. However, real-life applications are typically not quite as static. Instead, the program code often evolves over time, and multiple different versions of an application have to be executable in parallel. In such scenarios, hardcoding the program code and other files into the virtual machine images becomes complicated or even impossible. An alternative way to achieve program code distribution is to include facilities for dynamic code search and distribution on middleware level.

This paper introduces a framework for seamless runtime program code distribution. The framework is based on our earlier CloudScale [15] research prototype, but due to high independence level, can be used separately in other systems as well. CloudScale is a middleware, which simplifies the development of Java application in an IaaS cloud. In our framework, program code distribution is entirely handled by the underlying CloudScale middleware. We evaluate different configuration and implementation options, and present numerical performance results based on an illustrative case study.

The remainder of this paper is structured as follows. Section II describes illustrative case study that demonstrates the code distribution problem based on a real-world application. Section III describes research related to our work. Afterwards, Section IV presents the background of the approach we present in this article, most importantly, the CloudScale framework. Section V presents the actual contribution of this paper, which is consequently evaluated in Section VI. Finally, the paper is concluded in the Section VII.

II. Illustrative Case Study

In the rest of the paper, we will use a Web 2.0 sentiment analysis [17] application for illustrative purposes. This case study application has originally been introduced in [14], and

978-1-4673-6435-5/13/$31.00 © 2013 IEEE

PESOS 2013, San Francisco, CA, USA

discusses a cloud-based Software-as-a-Service (SaaS) system. The SaaS application allows clients to register with their service, and have the online popularity of their brands and products monitored. To this end, a data collector application will analyze content posted to various social media (e.g., Twitter, Facebook, status.net). This data is then used by the SaaS application to produce detailed reports based on real-time sentiment data.

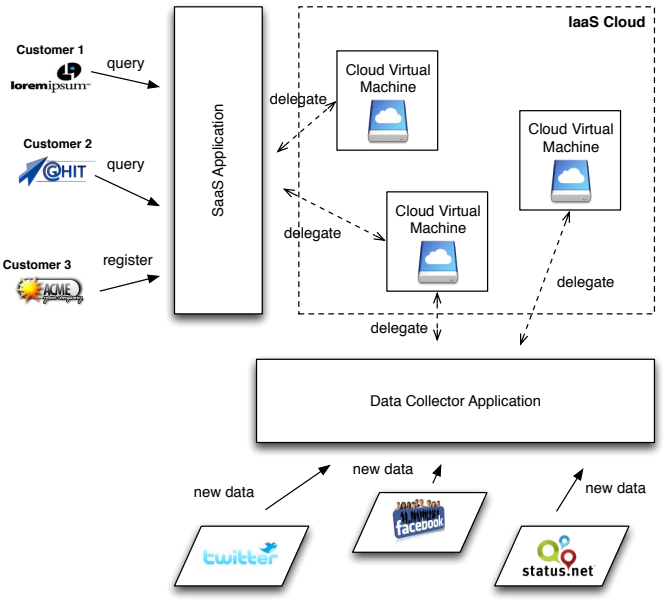

Figure 1. Sentiment Analysis Scenario Overview (Based on [14])

A high-level overview over this setting is depicted in Figure 1. Various clients access the SaaS application with requests for reports. In order to achieve elasticity and scalability, the application delegates the execution of processing-intense tasks, such as report generation, to a number of different virtual machines, which are rented on demand from an IaaS cloud, e.g., Amazon EC2.[1] Similarly, the data collector part of the case study uses IaaS virtual machines to retrieve, normalize and store data items retrieved from social networks. Normalization includes tasks such as stop word removal and stemming.

III. Related Work

The problem of code distribution over the network appeared almost at the same time as network communication became possible. The simplest solution that requires minimal development effort is to update code manually prior to execution. This solution is good enough for systems that update rarely, or in situations when the network speed is insufficient for code transmission or version verification at runtime. However, with further development of networking and network-aware applications, automated code updating has become common. Nowadays, applications often check for updates periodically or at startup, and download updated code versions when necessary. This approach is suitable and becomes a standard for common user-oriented applications, but in other cases more sophisticated methods are to be used.

One scientific area that inherently faces the problem of code distribution is grid computing. Usually, tasks developed for grid execution are computation-intensive and long running. Therefore, it is applicable to distribute code to the appropriate grid nodes prior to execution either manually or automatically. However, some approaches to distribute program code and additional data on-a-fly were proposed in [20]. Still, code distribution in grid computing is different from the approach we present, as they solve the problem of initial long-running code distribution or "hot patching" [6], but we provide the framework for the encapsulated task execution, that allows running different code versions on the same machine. This is important for development, testing and multiple user usage scenarios (multi-tenancy [4]).

The approach we present is more similar to the idea of mobile agents in agent-based computing. With this paradigm, applications are able to migrate from one computer to another autonomously and continue their execution on the destination computer [11]. Code distribution is a vital concept for such applications and a lot of research has been conducted to achieve different goals and improve code migration [3], [18], [2], [7], [8]. However, in contrast to our approach, mobile agents are active and choose themselves to migrate between computers at any time during their execution [9]. In the framework we present, the application is distributed transparently and is not aware that the code is being distributed. From this point of view, the presented approach is more similar to the idea of remote code evaluation [19], when a task is transmitted to the server to execute. Also, our approach exhibits some features of the code on demand approach [2], when missing code and related files can be fetched from the remote location on demand.

Finally, it should be noted that the solution we present here falls into the larger class of weak code mobility [7], as both code and data is transmitted, but not application state.

IV. Background and Contribution Context

The work we discuss in this paper has been carried out within the larger CloudScale research project, which has been initially presented in [15], [13], [14]. In the following, we will introduce CloudScale to the extent necessary for the purposes of this paper, and further detail the challenges that are being addressed in this contribution.

A. The CloudScale Framework

The general concept of CloudScale is to use aspect-oriented programming (AOP) techniques to dynamically modify the bytecode of Java-based applications, and transparently move designated parts of the application (which we refer to as cloud objects) to virtual resources in the cloud (referred to as cloud hosts). This process is entirely invisible to the application developer, and happens fully automated at application run time. In the end, applications built on top of CloudScale look like regular (local) Java applications, but are actually executed

[1] http://aws.amazon.com/ec2/

in a heavily distributed fashion. A simplified architectural overview of the CloudScale framework is depicted in Figure 2. Essentially, client applications access the cloud via a local cloud manager component. The cloud manager moves the execution of part of the application to various cloud hosts in the IaaS cloud. To this end, the cloud manager monitors the performance of each host [14] and schedules requests so that the total performance of the application is optimal, according to policies defined by the application developer [13].

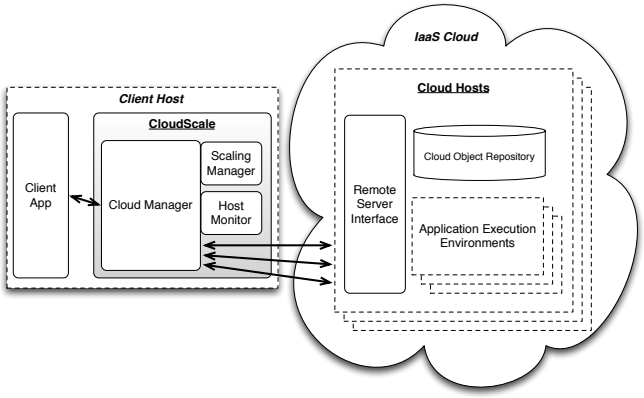

Figure 2. Simplified CloudScale Architecture (Adapted from [15])

In order to implement this CloudScale model, a plethora of challenges need to be addressed, including virtual machine management, application parallelization, scheduling of requests to virtual machines, performance as well as cost monitoring, and application code distribution. The latter challenge is the main focus of this paper. More thoughts on the other challenges can be found in our earlier CloudScale publications [15], [13], [14].

B. Program Code Distribution Challenges

Dynamically distributing program code in a real-life IaaS cloud requires a number of aspects to be addressed. (1) Firstly, the framework needs to detect when the code that is to be executed is not available at all, and request the code from a code server (the machine that has correct version of the application binaries). For instance, if the SaaS application described in Section II wants to delegate the generation of a report to a cloud host, the required program code needs to be available at this virtual machine. (2) If this is not the case, cloud host has to find the trusted code storage where appropriate code can be retrieved from. In our case, usually the application itself will act as a code server and deliver the required program code on demand, including the correct versions of all missing dependencies that are required to execute this code. Alternatively, it is possible to install a specific dedicated code server in the cloud, which then takes over this task from the client application to reduce its load. Evidently, it is possible to hard-code all required program code directly into the virtual machine images used by CloudScale, but this drastically reduces the flexibility of the system and makes maintenance of the system cumbersome and time-consuming. (3) Thirdly, some means of communication need to be established which allow program code to be transferred at runtime from the trusted code storage to the cloud virtual machines. This communication can be handled either in a point-to-point fashion (e.g., via Web services technology, such as SOAP) or via a messaging middleware. (4) Fourthly, for practical performance reasons, the middleware needs to optimize the communication protocol between application and cloud hosts. For instance, it is typically not feasible to initiate the dynamic code exchange routines separately for each missing class of application code. Instead, the middleware needs to smartly decide which additional code and non-code resources (e.g., images, configuration files) will also be required in addition to the already detected missing code. These dependencies should be distributed over the cloud at the same time to minimize the overhead of the code distribution. (5) Fifthly, after dynamically loading the program code, the middleware needs to decide how long this code and its dependencies can now be considered as valid. To this end, it is required that the CloudScale middleware is able to detect when a different version of the program code is needed and use it.

Table I
SUMMARY OF CODE DISTRIBUTION CHALLENGES

#	Challenge Name	Challenge Synopsis
1	Missing Code Detection	Cloud hosts need to be able to dynamically detect if program code needs to be loaded on demand
2	Trusted Code Storage	Cloud hosts need to be able to locate the code storage service (typically the client application or a dedicated code server in the cloud)
3	Communication Middleware	Cloud hosts need to have access to a suitable communication middleware that allows them to dynamically load code
4	Communication Protocol	Cloud hosts and client application need to use an efficient protocol for minimizing the communication overhead incurred by dynamic code loading
5	Code Versioning	Cloud hosts need to be aware that program code can change, and that loaded program code is not valid indefinitely

These challenges are summarized in Table I. In the remaining paper, we will discuss our approach to solve these challenges within the CloudScale project.

V. DYNAMIC PROGRAM CODE DISTRIBUTION

Whenever a cloud host in the CloudScale middleware has to execute a new task, the system has to ensure that all necessary resources are available and schedule the execution of the task. In this section we will describe our approach and show how we try to achieve efficient and seamless code distribution, solving the challenges described in Section IV-B.

A. System Overview

When the client application approaches a code segment that can be delegated to the cloud, it schedules the execution on the cloud hosts that are available at the moment. If there are no hosts available, additional cloud machines can be started. In more details this process is described in [15]. On the cloud host, the scheduled code starts executing, while a special class loader on platform level maintains and fetches all required program code and other relevant resources, such as configuration files. The architecture of our solution is visualized in Figure 3, where you can see application started from the client host distributing work to the set of cloud machines that retrieve necessary code from the cloud code cache or directly from the client. Due to this architecture, code that is being executed does not have to care about code availability and version, as the underlying infrastructure handles these problems seamlessly and transparently.

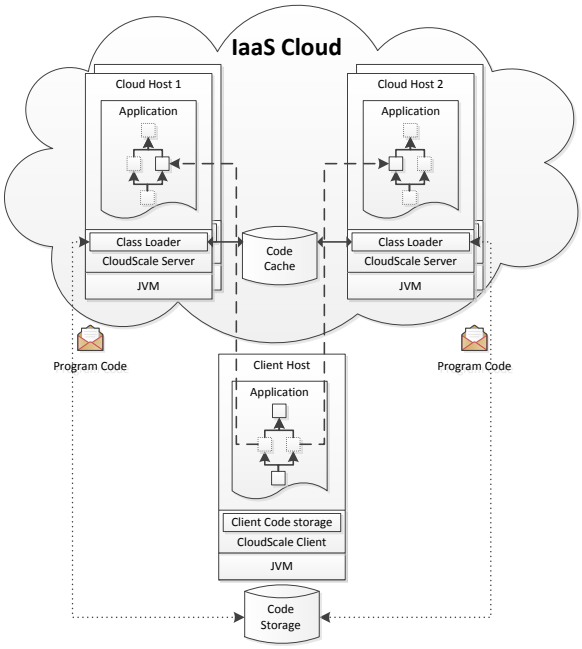

Figure 3. Overview of program code distribution model

1) Missing Code Detection: In most programming languages (including Java, as used by CloudScale), detection of missing resources (both code and non-code files) can be handled by the developer through an Application Programming Interface (API). However, in order to avoid misbehaviors, solve stated challenges and be able to control code availability and load sequence, we implemented, basing on available API, a special module in our middleware to intercept all requests for program code at cloud hosts. Concretely, we intercept the class loading mechanism of the programming language to check against a set of already resolved classes. If the required code has already been loaded, it can be provided again without any additional work required from the class loader. If the required code has not been loaded before during this execution, the class loader checks a code cache for it, as shown in Figure 4. The details of this mechanism will be explained later. If the code was not found in the cache, the class loader requests the code from the client (or from a trusted code storage), and waits for the response (see Figure 3).

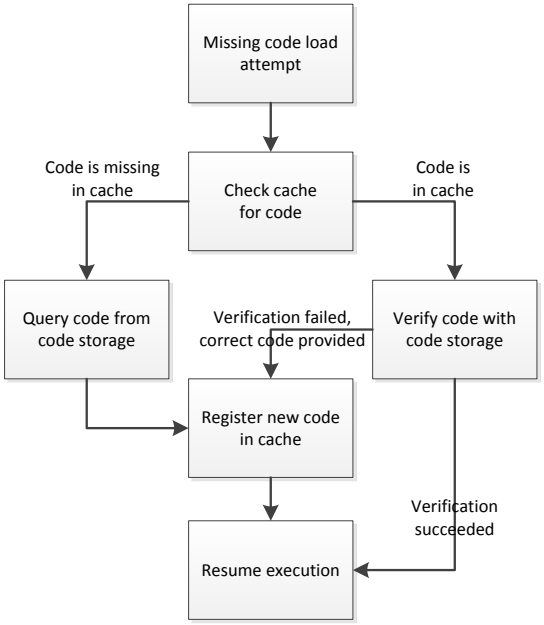

Figure 4. Code loading strategy

2) Communication Middleware: Our code loading system does not have any specific requirements for a particular communication channel, and usually can be used over the same communication facilities as used by the rest of the CloudScale middleware. Resource loading works based on simple blocking calls and may require the ability to initiate communication with the trusted code storage facility. The only communication channel properties that are important for this use case are reliability and a reasonable data transfer speed. Channel speed is very important as communication delay is directly influencing the application performance on the cloud. Evidently, any network communication is slow as compared to local code retrieval, and the slower the communication, the lower the performance benefits that can be reached by distributing the application over the cloud in the first place. Reliability is also vital. Transfer errors, which cloud host can detect with the help of check sums, can dramatically impact communication speed due to code retransmission. In case of communication failure, the application has to shutdown gracefully, as there is no code to continue executing the task.

In our current version of CloudScale, we require a JMS-

compatible message queue (e.g., Apache ActiveMQ[2]) to provide the communication channel used for all client-host communication, including dynamic code loading.

3) Trusted Code Storage Location: While the creation of dedicated code server may improve reliability and performance of code distribution framework, for some cases this solution is not preferable. Sometimes it is required to be able to fetch actual code directly from the CloudScale application. For example, during software development or testing, it makes more sense to use initial application startup machine for code distribution instead of dedicated server that has to be updated prior to every run. In such situations, code delivery service has to be provided from the client application. Moreover, the client application is typically the most reliable source of the code, as the client application codebase contains exactly the code that the developer expected to run. Therefore, by default, the client application always runs the code distribution service, even in situations when a dedicated code server is expected to be used. This simplifies framework configuration and allows using the client application to update or verify code on the code server, or as a fallback option in case the code server is offline or overloaded.

The code delivery service within the client application has to be able to provide the code to the cloud machines without interrupting the main application thread. To achieve this, the service is started in dedicated thread. When the code distribution service receives a request, it checks for availability of the requested code and decides what to send. The code provided by the trusted source is then stored in the cache, mapped to the appropriate task to enable multi-tenancy and execution is resumed.

4) Code Versioning: To solve the challenges related to code version control and updated code propagation, we implemented a code verification system as part of the CloudScale class loading mechanisms. In case the code is available in cache, the class loader still has to ensure that the code has the same version as the client expects. Therefore, the class loader carries out code verification based on the last modified date and size of the code files, as depicted in Figure 4. Evidently, some other alternatives to implement code verification are feasible as well (e.g., using hash-codes, explicit versioning via version numbers, or partial transfer), but we deemed the selected heuristic approach to be the fastest, while still being reliable enough for practical applications. This point of view is supported by the fact that similar approaches are used in other state-of-the-art solutions, e.g., RSync,[3] Apache Ant,[4] GNU Make[5] and others.

As code is stored in the cache, not only the required program code itself, but rather all files that were provided previous time for the same code request are verified. For each file in this set, client either confirms that this is the expected code or provides the file that should be used (see Figure 4). After this, the class loader delivers the correct code for the execution and, if necessary, updates the cached version.

[2]http://activemq.apache.org/
[3]http://rsync.samba.org/
[4]http://ant.apache.org/
[5]http://www.gnu.org/software/make/

B. Code Caching

In CloudScale, cloud hosts execute each separate request in a sandbox. To this end, the class retrieval infrastructure on each cloud host resolves all resources for each request separately. This allows the execution of different requests using different code bases, and restricts any possible influence of one request on others. However, evidently this approach introduces some redundant code transmission, because if the same program code should be used more than once, it still will be transmitted separately for each request. To avoid this redundancy, we introduce a smart code caching mechanism.

For the first code request, when the required code is not yet cached, it has to be downloaded from the trusted code storage, while each of the following requests only uses the code available from the cache (if code verification is successful). When changes are detected during verification, the outdated code is either replaced or used in parallel to the updated version, depending on the cache usage and configuration policy. When there are no changes, the cached code can be used without transmission through the communication channel.

Table II
CACHE DEPLOYMENT SELECTION TRADEOFF

	Host Private Cache	**Cloud Cache**
Cloud-Based Code Storage	+ code access speed - low cache hit rate	- no speed up + good cache hit rate
External Code Storage	+ code access speed - low cache hit rate	+ code access speed + good cache hit rate

The main task of the caching mechanism is to provide faster code fetching in the situations when the same code is requested multiple times. Therefore, code from the cache has to be accessible faster than from the trusted code storage (e.g., the client application). The fastest possible location of the cache is the hard drive or even memory of the cloud host. This will give ideal access speed, but will reduce the cache hit rate, as each cloud host will have to maintain its own cache. In case of some distributed applications, this approach may give no benefits at all, as it is shown in Table II. Another possible approach is to create a dedicated cache server or share one cache between multiple cloud hosts. This is a good solution if the code is initially transmitted through an unreliable or slow channel, but if the application is already using a dedicated code service, a shared cache in the cloud hardly makes any sense, as access speed will be almost identical as to the code server.

From the situation described above, it is clear that we face a tradeoff as discussed in the Table II. Depending on the environment configuration and situation, different approaches will be more efficient and, hence, preferable. Therefore, to achieve the best performance, it makes sense to allow the application to decide on the preferred caching strategy.

C. Batch Loading

When the class loading infrastructure receives the request for new classes or resources to be loaded, there is not much information available to make some assumptions about the data that should be loaded. The only thing that is available is the name of the resource that should be retrieved. Therefore, cloud host has to send request to the storage facility with only required resource name specified (as described above, the situation is slightly different when there is code already available in the cache, we will omit this case now for the sake of simplicity).

When the code retrieval request arrives to the storage facility, appropriate service has to find the required piece of code and decide what to send along with it. Of course, the simplest scenario would be to send only the requested resource, but this would increase the cost of dynamic code load and slow down application, especially at the startup. Another extreme would be to send all application code at the first request: this would decrease amount of messages, but might introduce even longer delay for the very first request, when the entire set of libraries and code base is transmitted. Considering the fact that usually not all code would be required on each cloud host, this option may introduce even more overhead than the first one.

One possible option to solve this tradeoff would be to allow end-user application to configure amount of code that should be transferred for each request. However, this approach would be rather cumbersome for the developers and against the primary design goals of CloudScale (making it easy to build cloud applications). Another choice would be to use heuristics, which would propose satisfying solution for common usage scenarios.

For example, if the requested class belongs to a library (e.g., a jar file), it makes sense to send the entire library instead, as the chances that other resources from that library will be requested are high. Similarly, if the class belongs to a package, it makes sense to consider sending the entire package. Also, if the class has some dependencies or belongs to hierarchy of the classes or interfaces, other classes are very likely to be needed as well.

All of these heuristics have their own benefits and problems and it is complicated to determine which of them should be used as the default behavior. To determine the influence of these factors on real-life applications, we included a number of different batch loading algorithms in our numerical evaluation and present evaluation results in the following section.

VI. Evaluation

To support our arguments and evaluate costs and benefits of our code distribution approach, we use an implementation of the sentiment analysis application described in Section II. This application has a number of features that allow us to better evaluate different code distribution strategies. Firstly, the application is easily parallelizable, therefore we can find appropriate task size to achieve effective load on different amount of hosts to evaluate the influence of the selected code distribution strategy on the application setup. Secondly, the code base of this application is of significant size (8 MB in 6 JARs), and also makes use of a number of large non-code resources (42MB in 14 files). This allows us to measure influence of the network communication and storage delay on application performance.

A. Evaluation Setup

We run our evaluation on a private IaaS cloud based on OpenStack.[6] Our private cloud consists of 8 physical machines (Dell blade servers with two Intel Xeon E5620 CPUs running at 2.4 GHz Quad Core and 32 GB of RAM each), which are connected via dedicated Gigabit Ethernet. The sentiment analysis application was hosted on a conventional laptop, which was connected to the private cloud via LAN. The application code was implemented in Java, and configured to work with the CloudScale framework [13], [15] and our private cloud. In our evaluation we used one medium-sized cloud instance (2 virtual CPUs, 3.75 GB of RAM) for the ActiveMQ communication server, and between 1 and 5 small instances (1 virtual CPU and 1.7 MB of RAM each) as application execution hosts. All hosts were running Ubuntu Linux 12.04[7] and Java 1.7.

The first step of our evaluation was to create a base line for the evaluation of code distribution. To do this, we ran the sentiment analysis application with all required code and files embedded into cloud machines in 3 setups: using 1, 3 and 5 small cloud instances as computational resources. This approach does not require any code distribution, therefore allows us to quantify the overhead of different code distribution strategies. After that, we repeated the same tests using the 3 different code distribution strategies that were described above:

- *Complete Code Distribution Strategy*, when all application code is provided on the first request at the startup.
- *Class-based Code Distribution Strategy*, when only the requested class or resource is provided, and the cloud host has to ask for each resource separately.
- *Smart Batching Code Distribution Strategy*, when code is delivered in highly-related batches, therefore optimizing amount of necessary requests and minimizing unnecessary code transmission.

On the cloud hosts side, we evaluate the performance of 2 different code caching strategies (private to each host and shared within the cloud), as described in Section V. The private caching strategy is using each machine's private hard disk as storage, while the shared caching is implemented on top of a Riak[8] key-value database hosted on a separate cloud instance.

Each experiment was executed multiple times, and mean values are used in the following. Still, we faced some execution anomalies that were caused by different application execution speed and environmental conditions, partially due to Twitter request rate limits. [9]

[6] http://www.openstack.org/
[7] http://releases.ubuntu.com/precise/
[8] http://basho.com/riak/
[9] https://dev.twitter.com/docs/rate-limiting

B. Evaluation Results

Average baseline experiment execution time was about 1.5 minute, while usage of code distribion algorithm extended this time to 22.5 minutes. To focus on performance costs of each class loading strategy, we decided to compare the execution duration overhead of each strategy in each environment configuration. In Figure 5 one can see the mean execution overhead (i.e., the actual execution time minus the baseline established before, by executing the application without code distribution) of different strategies in different setups (1, 3 and 5 cloud hosts) with private cache used on each host.

With this evaluation run we wanted to determine how our Smart Batching code distribution strategy behaves compared to two other extremes: Complete code distribution and Class-based code distribution.

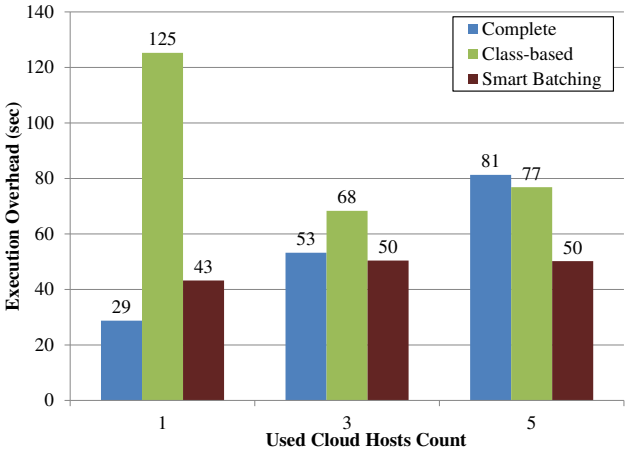

Figure 5. Sentiment analysis application execution time overhead caused by different class distribution strategies with local cache.

Comparing private and shared caching strategies, we would like to focus on Figure 6, where we show the execution overhead of the Smart Batching code distribution strategy with private and shared batching used on cloud hosts.

C. Discussion

In Figure 5, we can see that if we are using only one cloud host, the Complete Code distribution strategy is indeed the fastest, while Class-based is the slowest. The reason of this is that network bandwidth in our evaluation setup is good enough that transferring all necessary code a few times induces a lower delay than establishing a large number of interactions for smaller code batches. In this evaluation setup, the Complete Code provider sent in total over 110 MB of data and showed the smallest execution time overhead, while the Class-based and Smart Batching sent only close to 50 MB.

Additionally, it can be seen that the Smart Batching strategy is comparable to the Complete strategy, but much faster than Class-based. The reason of this is that Smart Batching strategy transferred almost the same amount of code as Class-based strategy, but using significantly less requests. However, please

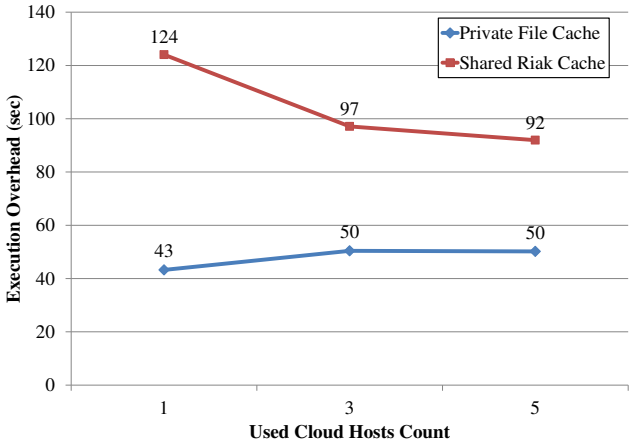

Figure 6. Comparison of execution overheads of private and shared cache with Smart Batching code distribion strategy.

note that in a different setup, when the connection between the client application and the cloud is slower, and the amount of transferred data is more important than the amount of requests, the Smart Batching strategy would be more efficient than any of competitors.

With the increasing amount of cloud hosts used, it can be seen that cost of Class-based strategy decreases, while amount of requests logically stays the same or even increases. This happens because these requests can be handled in parallel, reducing influence on overall application execution time. At the same time, the overhead of using the Complete strategy increases as it hits the bandwidth limit of the network communication between client and cloud hosts. It can be seen, that execution time in this case increases almost linearly with the number of hosts. This is caused by the fact that all hosts need code roughly at the same time (at the beginning of the execution), causing the client application to send the same code to each host at the same time, slowing down the overall execution of the application. For the Smart Batch strategy this cost is not as high and it keeps approximately the same performance, hence quickly becoming the best strategy if multiple cloud hosts are used.

Discussing Figure 6, it can be seen that when the hosts use a private cache, each of them is competing over the communication channel between client and cloud, while in the shared cache setup it is enough to download the required data only by 1 host, making it available to every host in the cloud at the same time. Hence, with the number of cloud hosts increasing, the private cache setup overhead increases, while the shared cache overhead decreases. However, it can be clearly seen from the figure, that overall overhead of the shared setup is still significantly higher. This is caused by the larger amount of communication and data transmission required for the shared cache setup. In the private cache setup, cloud hosts download data directly from the client, while in the shared cache case, one host has to download code from client,

put it into cache and then other hosts can access it. Hence, there are actually 3 transmissions instead of just one, which is significant in our case, when transmission speed from client to cloud is comparable to the communication speed inside the cloud.

Basing on our evaluation results, we can state that the Smart Batch code distribution strategy is usually the preferable way to achieve seamless dynamic code distribution with least overhead compared to other, simpler, options. Additionally, from this evaluation, we saw that caching strategy can influence application performance a lot; therefore it makes sense to allow users to configure this parameter according to their communication channel characteristics.

VII. Conclusions

With increased popularity of cloud computing, more and more developers and companies think about cloud-aware versions of their applications. With the cloud-oriented applications development, a set of problems arises that did not exist or were not that significant before. One of them is a problem of code distribution to the cloud hosts.

The seamless code distribution framework that was introduced in this paper, allows distributing code to the cloud on demand and seamlessly to the application. The introduced framework was evaluated on a real-life application and overhead of different code distribution and caching strategies where compared and analyzed. The evaluation showed that selected code distribution approach provides a list of benefits over alternatives and minimum overhead for the users, while requiring insignificant amount of time to configure and use.

In the future, we plan to improve our Smart Batch code distribution approach and implement some other performance tweaks. For example, in our current architecture we do not consider possibilities of predicting required code and sending multiple requests to the client with the same package or implementing some smart code prefetching algorithm that would improve code execution speed and allow background code loading. Additionally, historical information can be considered. For example, history of requests from similar parallel or previous executions can be used to predict future requests and ask all required information upfront.

Acknowledgements

The first author of this paper is financially supported by the Vienna PhD School of Informatics.[10] Furthermore, the research leading to these results has received funding from the Austrian Science Fund (FWF) under project references P23313-N23 (Audit4SOAs), as well as the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 257483 (Indenica).

[10]http://www.informatik.tuwien.ac.at/teaching/phdschool

References

[1] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[2] M. Baldi, S. Gai, and G. Picco. Exploiting code mobility in decentralized and flexible network management. In *Mobile Agents*, pages 13–26. Springer, 1997.

[3] J. Baumann, F. Hohl, K. Rothermel, and M. Straßer. Mole–concepts of a mobile agent system. *World Wide Web*, 1(3):123–137, 1998.

[4] C.P. Bezemer, A. Zaidman, B. Platzbeecker, T. Hurkmans, and A. t Hart. Enabling multi-tenancy: An industrial experience report. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–8, Washington, DC, USA, 2010. IEEE Computer Society.

[5] S. Bhardwaj, L. Jain, and S. Jain. Cloud computing: A study of infrastructure as a service (iaas). *International Journal of Engineering and Information Technology*, 2(1):60–63, 2010.

[6] S. Bratus, J. Oakley, A. Ramaswamy, S.W. Smith, and M.E. Locasto. Katana: Towards patching as a runtime part of the compiler-linker-loader toolchain. *International Journal of Secure Software Engineering (IJSSE)*, 1(3):1–17, 2010.

[7] G. Cabri, L. Leonardi, and F. Zambonelli. Weak and strong mobility in mobile agent applications. In *Proceedings of the 2nd International Conference and Exhibition on The Practical Application of Java (PA JAVA 2000), Manchester (UK)*, 2000.

[8] A. Carzaniga, G.P. Picco, and G. Vigna. Designing distributed applications with mobile code paradigms. In *Proceedings of the 19th International Conference on Software Engineering*, pages 22–32. ACM, 1997.

[9] V. CeronmaniSharmila and V. KomalaValli. Enhanced security through agent based non-repudiation protocol for mobile agents. *International Journal of Power Control Signal and Computation(IJPCSC)*, 3(1), 2012.

[10] S. Dustdar, Y. Guo, B. Satzger, and H.L. Truong. Principles of elastic processes. *Internet Computing, IEEE*, 15(5):66–71, 2011.

[11] M. Ghorbel, M. Mokhtari, and S. Renouard. A distributed approach for assistive service provision in pervasive environment. In *Proceedings of the 4th International Workshop on Wireless Mobile Applications and Services on WLAN Hotspots*, pages 91–100. ACM, 2006.

[12] A. Khajeh-Hosseini, D. Greenwood, and I. Sommerville. Cloud migration: A case study of migrating an enterprise it system to iaas. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 450–457. IEEE, 2010.

[13] P. Leitner, W. Hummer, B. Satzger, C. Inzinger, and S. Dustdar. Cost-Efficient and Application SLA-Aware Client Side Request Scheduling in an Infrastructure-as-a-Service Cloud. In *5th IEEE International Conference on Cloud Computing*, 2012.

[14] P. Leitner, C. Inzinger, W. Hummer, B. Satzger, and S. Dustdar. Application-Level Performance Monitoring of Cloud Services Based on the Complex Event Processing Paradigm. In *Proceedings of the 2012 IEEE International Conference on Service-Oriented Computing and Applications (SOCA'12)*, 2012. To appear.

[15] P. Leitner, B. Satzger, W. Hummer, C. Inzinger, and S. Dustdar. CloudScale - a Novel Middleware for Building Transparently Scaling Cloud Applications. In *ACM Symposium on Applied Computing (SAC)*, 2012.

[16] P. Mell and T. Grance. The nist definition of cloud computing (draft). *NIST Special Publication*, 800:145, 2011.

[17] B. Pang and L. Lee. Opinion mining and sentiment analysis. *Found. Trends Inf. Retr.*, 2(1-2):1–135, January 2008.

[18] K. Rothermel, F. Hohl, and N. Radouniklis. Mobile agent systems: What is missing? *Distributed Applications and Interoperable Systems (DAIS'97), Chapman & Hall*, pages 111–124, 1997.

[19] E. Sanchis. Mobility and remote-code execution. In *Mobile Wireless Middleware, Operating Systems, and Applications-Workshops*, pages 85–97. Springer, 2009.

[20] A. Van Hoff, J. Payne, and S. Shaio. Method for the distribution of code and data updates, July 6 1999. US Patent 5,919,247.