Abstract Behavioral Specification: unifying modeling and programming

Bezirgiannis, N.

Chapter 5

A Distributed Implementation of HABS

The IT industry, always looking for cutting operational costs, has been increasingly relying on virtualized resources offered by the "Cloud". Besides being more economically attractive, the Cloud can allow certain software to benefit in security and execution performance. For these reasons, software applications are steadily being migrated to run on virtualized hardware, essentially turning cloud computing into a hot topic among the software community.

Recent research has led to numerous methodologies, tools, and technologies being proposed to help the migration and execution of software in the cloud, ranging from (static) configuration management tools to (live) orchestration middleware, and from simple resource monitoring services to the dynamic (elastic) provisioning of resources. Unfortunately, the (so-called) DevOps engineers are now burdened with developing and maintaining an extra logic for such cloud tools, besides the usual application logic. These cloud tools may be best described as semi-automatic and it is often the case that an engineer has to manually intervene to apply the desired configuration and deployment of a cloud application.

These cloud applications are migrated unaltered: monolithic boxes of code which are transferred from a non-cloud setting to the new cloud environment by the DevOps engineers. Such separation of the application from its execution is traditionally believed to be an advantage, long before Cloud came to existence. However, one would expect that with the introduction of the virtualized (dynamic) hardware of the Cloud, and since software logic is inherently dynamic, an application could "become aware" (as in Artificial Intelligence) and make use of its own computing power for managing its cloud resources and deployment in an optimal way, and without requiring constant administering of an engineer.

In this section, we aim to address the challenges of engineering cloud applications by introducing a "cloud-aware" programming language that provides certain high-level abstractions for unifying the application logic together with its deployment logic in a single integrated environment, while in the same-time, hiding any lower-level hardware and cloud-provider considerations. The language is intended for DevOps engineers and (potentially) computational scientists who are responsible for both the development and execution of software residing in the Cloud but would rather focus more on the application's logic than continuously manage its deployment. Applications written in the proposed language are christened "cloud-aware" in the sense that they can *actively* monitor and control their own deployment.

The proposed language is based on the *Abstract Behavioral Specification language* (ABS) that we introduced in chapter 2. We extend ABS with DC's that serve as a suitable abstraction over Cloud Virtual Machines and which allow the application to distribute itself among multiple (provider-agnostic) computing systems. The ABS developer writes code that can dynamically create, monitor and shutdown such Deployment Components (Virtual Machines) and most importantly bring up new objects inside them. To this end, an ABS cloud-application forms a cloud-aware *distributed-object* system, which consists of a number of inter-VM objects that communicate asynchronously, while recording any failures that may happen in the cloud.

An implementation of this extension must be efficient and safe so that it can be put in production code. For this, the Haskell backend of ABS (HABS) is chosen for translating ABS code to Haskell intermediate code, which is again typechecked and transformed to an executable by an external Haskell compiler. The choice of Haskell was made mainly for two reasons: the HABS backend seems to be currently the fastest in terms of speed and memory use (see Section 3.6.3, attributed perhaps to the close match of the two languages in terms of language features: Haskell is also a high-level, statically-typed, purely functional language. Secondly, compared to the distributed implementation sketched in Java, the ABS-Haskell runtime utilizes the support of Haskell's lightweight threads and first-class continuations to efficiently implement multicore-enabled cooperative scheduling. Java does not have built-in language support for ABS' algebraic datatypes; its system OS threads (heavyweight) coupled with no support for continuations make it a less ideal candidate to implement cooperative scheduling in a straightforward manner. On the distributed side, we decided against layering our solution on top of Java RMI (Remote Method Invocation) framework due to a lack of built-in support for asynchronous remote method calls and superfluous features to our needs, such as code-transfer and fully-distributed garbage collection.

We augment the HABS backend with support for *Cloud-Haskell*, a framework for type-safe, fault-tolerant distributed programming in the Haskell ecosystem. The implementation, although in its infancy, is already being tested in a real cloud environment, exhibiting promising results which are also presented. We further extend the (parallel) HABS runtime with support for Deployment Components that provide a suitable abstraction of the Virtual Machines provided by the Cloud and

which allow the application to distribute itself among multiple machines. The ABS programmer can dynamically create, monitor and shutdown such Deployment Components (Virtual Machines) and most importantly assign new objects to them. As such, an ABS cloud-application consists of several inter-VM communicating objects, effectively forming a distributed-object system which can control its own deployment and still benefit from the (local) parallelism.

A Deployment Component (DC) is a minimal description of the computing resources available to an ABS application (section 4.3). We propose to extend this notion to allow any cloud resource that can properly be quantified (for example memory, disk, network, etc). On the other hand, and in contrast to the original specification, we restrict a DC to correspond solely to a Platform Virtual Machine (VM) — indeed, the terms DC and VM are used interchangeably by our extension. We call each deployed ABS application of a separate DC/VM, an *ABS node*. A running ABS program on the cloud will effectively form a distributed network of multiple inter-communicating ABS nodes.

5.1 Implementation

We extend the parallel HABS runtime with support for Deployment Components that provide a suitable abstraction of the Virtual Machines provided by the Cloud and which allow the application to distribute itself among multiple machines. The ABS programmer can dynamically create, monitor and shutdown such Deployment Components (Virtual Machines) and most importantly assign new objects to them. As such, an ABS cloud-application consists of a bunch of inter-VM communicating objects, effectively forming a distributed-object system which can control its own deployment and still benefit from the (local) parallelism.

At runtime, each COG is a Haskell lightweight thread (with SMP parallelism). The COG-thread holds a process-enabled *queue*, a process-disabled *table*, and a local *mailbox*. Upon an asynchronous method call, a new process is created and put in the end of the process-enabled queue; note that processes are not threads, they are coroutines (first-class continuations) and thus can be stored as data. The COG resumes the next process from the queue until it reaches an `await` (on a future or a condition), where the process is suspended and moved to the process-disabled table. Later, another process informs the COG (by writing to its mailbox) that the await-condition is met; the COG will move back the process to the enabled queue. This strategy avoids *busy-wait* polling the boolean await conditions of processes. For more information, we refer to the section 3.5 where the runtime execution of HABS is detailed.

Moving on to distributed part, the distributed communication of ABS processes is realized by Cloud-Haskell [Epstein et al., 2011], which is a Haskell library for type-safe, fault-tolerant distributed programming. The distribution model of Cloud-Haskell resembles that of the Erlang programming language, with the difference being

that Cloud-Haskell has extra support for type-safe and version-safe message passing, features that we also make very much use of in our Cloud Runtime extension. We reuse the network transports and serialization protocols defined in Cloud Haskell for the ABS transmitted data between Virtual Machines. Each COG-thread is accompanied by a separate Cloud-Haskell thread (also lightweight) that listens for messages in a *public* mailbox and *forwards* them to the local mailbox of its associate COG-thread. This approach was chosen to firstly, avoid needless network-serialization between local communication and secondly, treat our distributed extension as optional to our (previously SMP-only) Haskell backend. The resulting remote communication remains transparent to the user: new objects can be remotely created inside a different machine and asynchronous calls are made to remote objects (living inside a remote machine) without changing the syntax and semantics of the ABS language.

As we have seen previously in chapter 4, a minimal implementation of DC has functionality for (1) shutting down the corresponding virtual machine and (2) probing for its average *system load*, i.e. a metric for how busy the system stays in a period of time. We use the Unix-style convention of returning 3 average values of 1, 5 and 15 minutes. In the case of (1), a VM shutdown implies that its cloud resources are eventually freed. Each kind of cloud infrastructure service carries its own implementation of DC. The intention is that the user will not have to provide such class implementations since the implementation of deployment components will come bundled with class libraries for common cloud-backend technologies (Amazon, OpenStack, Azure, etc).

The newly-created object will "live" in the specified DC. The annotated `[DC: ..] new` behaves similar to the `new` keyword: it creates a new COG, initializes the object, and optionally calls its run method. The annotation `[DC:] new Class` returns a *remote object reference*, (also called a proxy object; o1 in the above example) whose methods can be called asynchronously and which can passed around in parameters as normal. Every remote object reference is a single "address" *uniquely identified* across the whole network of nodes. Calls to `[DC: ...] new` will also (besides `shutdown`, `load`) block until the VM is up and running. From a theoretical standpoint, a remotely-spawned object must point to the same code (attributes and methods) as in a local object.

5.1.1 Connection to Cloud infrastructure

To properly support resource-aware programming (letting an ABS model manage and control its resources), when creating a new DC, a cloud provider is contacted in the background (usually via an XML-RPC API) and asked to bring up a new VM with the given characteristics. After the machine has booted, the caller replicates itself (the current ABS application) by transmitting its machine code to the newly-created machine. In case the cloud provider offers heterogeneous platforms (different OS or CPU architecture), we instead transmit the ABS source code and compile it in-place with our compiler toolset (that resides beforehand in the VM's image). The

new machine runs the transmitted ABS application and sends an acknowledgment signal to its creator, which can now start computations to the new DC by spawning new objects in it. The image of the virtual machine that is chosen is pre-defined (via a specific tag name in the cloud infrastructure) and should come with the appropriate Haskell and HABS compiler toolset already installed. Uploading the image is the only manual step required by the cloud ABS user.

The creation of Deployment Components is done under the hood by contacting the corresponding (cloud) platform provider to allocate a new machine, usually done through a REST API. The executable is compiled once and placed on each created machine which is automatically started as the first user process after kernel initialization of the VM has completed.

Simply put, a DC corresponds to a single Virtual Machine having certain resources which executes ABS code. We restrict the definition of DC to correspond only to a Platform Virtual Machine (VM)[1] residing inside the boundaries of a Cloud infrastructure.

By having a common DC interface, the different cloud backends can agree on a basic service, while still being able to provide additional functionality through subinterfaces and distinct DC-interfaced classes. Each DC-interfaced class implements the connection to a distinct cloud provider (e.g. Amazon, Openstack). A code skeleton of such a class follows, where the DC (VM) is parameterized by the number of CPU cores and main RAM memory:

```
module StandardLibrary.SomeProvider;

data CpuSpec = Micro | Small | Large;
data MemSpec = GB(Int) | MB(Int);

class SomeProvider(CpuSpec c,MemSpec m) implements DC {
    Unit shutdown() { /*omitted*/ }
    Triple<Rat,Rat,Rat> load() { /*omitted*/ }
}
```

The implementer can expose other properties to DCs, such as, network, number of IO operations, VM region location. A concrete implementation, which is omitted for brevity, usually involves some high-level ABS logic coupled with low-level code written in a foreign language (in our case Haskell). The average ABS user will not have to provide such connections to the cloud, since we (the implementers) intend to provide class implementations for most major cloud providers/technologies, in an accompanying ABS library. With this approach, we lift the low-level API of the cloud provider (usually XML-RPC) to a *typed* high-level API (e.g. CpuSpec and MemSpec datatypes).

[1] A platform virtual machine "emulates the whole physical computer machine, often providing multiple virtual machines on one physical platform" (from Wikipedia

Moving on, we create an object of the SomeProvider class by passing the number of cores and memory measured in GBs as class' formal parameters. The call to new SomeProvider contacts the specific cloud provider in the background to bring up a new VM instance from the pre-defined cloud image. The provider responds with a unique identifier (commonly the public IP address of the created VM) which is stored in the DC object. Finally, the machine is released by calling `shutdown()`, making the DC object point to `null`.

```
DC dc1 = new SomeProvider(Large, GB(8));
_ future_l1 = dc1 ! load(); // underscore infers the type
_ l1 = future_l1.get;
dc1 ! shutdown();
```

The creation of a DC object reference is usually fast, since it involves a single network communication between the current ABS node and the cloud provider. Still, the underlying VM requires considerably more time to boot up and be responsive, depending on factors such as provider's availability, congestion and hardware. To address this, we allow the creation of new DC objects to continue, but we require the program to potentially block when executing the first operation of the newly-created DC, as shown in the example:

```
DC mail_server = new Amazon(..);
DC web_server = new Azure(..);
DC db_server = new Rackspace(..);
mail_server !load(); // will block if DC is not up yet
```

Whereas the development of ABS code is by-definition provider-dependent — the user has to explicitly specify the class of the cloud provider —, the communication and interaction between the spawned remote objects is (in principle) *provider-agnostic*. To this extent, an ABS user could write an ABS cloud application that spans over multiple cloud providers and, most importantly, different cloud technologies.

The ABS user can create new Deployment Components (machines) just as creating objects (since DCs are modelled as objects). The DC class that is chosen dictates what kind of machine will be created; we currently provide library support for 3 DC classes talking to 3 different providers: OpenNebula, Microsoft Azure and Local (similar to Docker containers). The network communication is left to Cloud-Haskell and is provider-dependent: OpenNebula and Azure with TCP and Local with in-memory transport. We plan to extend our library with support for more (cloud) infrastructure providers.

Currently we are investigating the migration of ABS processes between DCs (machines); this can theoretically be achieved since ABS processes are merely data, and thus can be serialized and remotely transferred (migrated) from machine to machine.

5.1.2 Serialization

ABS data must be serialized to a standard format before they can be transmitted between DCs. The serialization of values of primitives and algebraic datatypes is automatically done by Haskell. We serialize object/future references to *proxy references* by serializing their Cloud-Haskell thread ID (network-unique) together with a COG-unique ID, and leaving out their actual attributes/future results. Each asynchronous method call is serialized to a *static closure*, i.e. a static code-pointer to the method (known at compile-time and platform-independent) and a serialized environment of its free variables (method arguments and local variables). No kind of code (source-, byte- or machine-code) corresponding to the method body is transferred. All serializations described above are type-safe and version-safe, in the sense that we include (in addition to the payload of an ABS datum) its serialized type signature and the library-versions of any types involved; thus, we avoid decoding bugs because of type and library-version mismatches.

Cloud Haskell code is employed for remote method activation and future resolution: the library provides us means to serialize a remote method call to its arguments plus a static (known at compile time) pointer to the method code. No actual code is ever transferred; the active objects are serialized to unique identifiers across the entire network and futures to unique identifiers to the caller object (simply a counter). The serialized data, together with their types, are then transferred through a network transport layer (TCP, CCI, ZeroMQ); we opted for TCP/IP, since it is well-established and easier to debug. The data are de-serialized on the other end: a de-serialized method call corresponds to a continuation which will be pushed to end of the process queue of the callee object, whereas a de-serialized future value will wake up all processes of the object awaiting on that particular future.

5.1.3 Garbage Collection

In a local-only setting, all ABS-based values, i.e. ADTs, futures, objects are automatically garbage-collected by the underlying Haskell GC. However, in our distributed setting some object/future references may have to be transmitted outside as proxy references, which results to the local ABS system garbage-collecting "too-early". An obvious solution would be to abolish automatic GC altogether, but that would hinder the development of software applications, especially those supposed to be long-running (as is the norm in cloud applications). On the other hand, introducing *distributed garbage collection* to ABS would allow both local and remote objects to be automatically GC'ed. The downside is that it is much more complex for the user to reason about the GC-incurred performance penalty which may be considerable. We chose a middle ground, where objects are by default GC-enabled and only become disabled when they are remotely communicated over (to another DC). The implementation has been straight-forward: a process appends the local object reference(s) that are transmitted remotely to a locally-held list of GC-disabled objects. This global list is held during the lifetime of the node, effectively surpassing

the Haskell's garbage collector underneath. Our design choice was based on best practice; we believe that a distributed cloud ABS application of many DCs would contain a combination of a lot of local ephemeral objects, and only few long-lived remote objects.

DCs, being special objects, are treated differently: when falling out of context, they are automatically GC'ed. That does not mean that the attached VM is shut down. The user that wants to shut down a DC but holds no reference to it any more, has to contact a remote object residing there to return a reference to the DC (with `thisDC`), or to shut it down on user's behalf. If the executing program holds (now and in the future) no reference to a DC and its objects, we consider its VM unreachable and fallen out of scope of the ABS application.

Futures are garbage-collected in a publish-subscribe pattern: the caller of an asynchronous method is a subscriber, while the callee is the publisher. When the callee has finished computing the future, it "pushes" the resulting value to its caller (the direct subscriber) and may now locally garbage-collect that value. A subscriber that "passes over" a remote future reference to other nodes becomes an intermediate broker with the responsibility to later also "push" that future value to all others *before* it is allowed to locally garbage-collect it. This forwarding strategy avoids unnecessary tracking and network communication between the initial node and all (directly and indirectly) subscribed nodes.

5.1.4 Failures in the Cloud

In cloud computing, and in any distributed system in general, failures are more frequent, mostly because of unreliable networks. Based on this fact, we further extend ABS with proper support for extensible, asynchronous exceptions. At the language level, exceptions are pure expressions modelled as single-constructor values of the ADT `Exception` , as detailed in section 3.2.1. To define new exceptions the user writes a declaration similar to an ADT declaration, e.g. `exception MyException(Int, List<String>);`. Our cloud extension predefines certain common "local" exceptions (e.g. NullPointerException, DivisionByZeroException) and cloud-related exceptions (e.g. NetworkErrorException, DCAllocationException, DecodingException).

Normally, if an exception reaches the outermost caller without being handled, its process will stop. We introduce a special built-in keyword named die that changes this behaviour and causes an object to be nullified and *all* of its processes to stop. With this in hand, a distributed application can easily model objects that can be remotely killed:

```
interface Killable { Unit kill(); }
class K implements Killable { Unit kill() { die; } }
Killable obj = dc1 spawns K();
obj ! kill();
```

Note that like Cloud-Haskell and unlike (distributed) Erlang, if a network error occurs between computing nodes, the connection will be dropped and not automatically re-connected. Unlike Cloud-Haskell, there is currently no primitive operation in HABS to allow the re-connection to a node after a network failure.

5.2 Extension: Service Discovery

Service discovery, the dynamic acquisition of a computing resource suitable to fulfill a specific task or group of tasks (i.e. a service), can help to decouple parts of a large distributed system. As such, service discovery is of interest to the Envisage case studies since certain large, distributed system architectures can be modelled naturally in this way. This section first briefly explains the basics of service discovery and lays out the design criteria for integrating service discovery into the ABS language.

In its most basic form, we see a service as a computing entity suitable to fulfill one or more specific tasks. Since in ABS tasks are modelled via method calls, it makes sense to model services as ABS interfaces and implement them using ABS objects. Note that in conventional object-oriented languages, objects and interfaces might not be sufficient, but the ABS concepts of asynchronous calls, distribution via deployment components, and safe parallel execution make ABS objects powerful enough to become services.

We augment the feature of "Deployment Components" (DC) with the ability of discovering available services offered by a DC. We adopt the notion of a service being represented by an ABS interface.

The acquire , expose, and unexpose methods are added to the DC interface. Thus, the DC interface becomes:

```
interface IDC {
    Unit shutdown();
    Triple<Rat,Rat,Rat> load();
    A acquire<A>(A);
    Unit expose<A>(A);
    Unit unexpose<A>(A);
}
```

The newly-introduced methods are parametrically-polymorphic; the programmer will instantiate their types when using them, as the following example:

```
{
DC dc1 = new NebulaDC(...);
MailService mail_server ;
Fut<MailService> f = dc ! acquire( mail_server );
mail_server = f.get();
mail_server ! send_mail (...);
}
```

The acquire method takes as input a "phantom object". The object is called phantom since the object's contents or reference are not actually send; the object is there only to give hints to the ABS compiler of what is the Interface we want the acquired object to comply with. The phantom object can also be introduced with a (nullary) declaration, as in the second line above: MailService mail_server ;

The call to acquire makes a request to the DC, asking for a reference to an (possibly remote) object that complies to the IDC interface/service. Upon processing the request, the DC searches through its *directory facility* for object subscriptions that support such an interface. If there is no search match, the DC will raise the ServiceNotFoundException and record it in the future as a fault. If the match succeeds, a reference to a complying object is returned. The returned object reference from the call to acquire can then be assigned back to the phantom object or any other object. This returned object reference is typed exclusively by the mentioned Interface and the user cannot normally know which is the actual class name (class implementation) behind it, unless this can be guessed through a method implementation.

An object can be subscribe to any DC's *directory facility* through the expose method. For example:

```
WebService ws = this;
dc ! expose(ws);
```

Accordingly, an object can be unsubscribed for some of its services/interfaces with the unexpose method:

```
AdminService a = admin_object;
MyInterface m = this;
dc ! unexpose(m);
dc ! unexpose(admin_object);
```

Following the approach of phantom objects, the arguments to expose and unexpose are type-checked with respect to the available interfaces that the object (class) implements. If the programmer omits such a phantom definition, the compiler will compute the object's principal interface (the object passed to acquire, expose, unexpose) through type-checking. If the ABS compiler cannot compute such a principal interface, it emits a type-checking error.

This peculiar design choice (of phantom objects) was made so as to not introduce any backwards-incompatibilities (adding interfaces as first-class citizens) and further more built-in keyword-statements. A further advantage is that the implementations of acquire and (un)expose methods can vary between DCs and thus be specific to the underlying service discovery technology of the cloud provider.

Two-times (un)exposing will not yield a runtime exception and will be silently suppressed. Each DC keeps track of its own subscribed objects and automatically

unsubscribes them in case they fall out of context, i.e. they have normally or exceptionally terminated.

5.3 Experiments and Results

We tested two instances of a real-world load-balancer: one with a static deployment of workers, and an adaptive (dynamic) load-balancer with worker VMs created on-demand based on how "well" the workers can keep up with incoming requests. Clients submitted job requests (of approximately of equal size) to the balancer at a steady rate; workers were distinct Cloud VMs that continuously computed the results for their incoming job requests.

The *static* load-balancer case is a fairly straight-forward cloud ABS application, consisting of 3 classes of LoadBalancer, Worker, and Client, exchanging asynchronous method calls of job requests/results. The LoadBalancer runs the main block and initially creates N number of Worker DCs (VMs) before starting to accept requests and forwarding them to workers in round-robin. We ran this static deployment against varying size (N=1..16) of worker VMs. The results of the runs are shown in Figure 5.1(a) stripped from the initial boot time of VMs. What we can draw from these results is that the completed jobs (per minute) nearly doubles when we double the number of worker VMs until we reach 5 workers. After that, we still increase the completed jobs but with a slower pace. This observation can be attributed to the fact that a point is reached where there is not a significant benefit from adding more worker VMs; the rate of job requests is always steady, thus worker VMs are "slacking".

We modified the static load-balancer to an *adaptive* version, that takes full advantage of the expressiveness of the cloud extension. The LoadBalancer creates now only one VM initially. We accommodate the LoadBalancer with a HeartBeater object which periodically retrieves the load from each worker in the VM "farm". The HeartBeater computes the average `load` of all VMs and if this average exceeds 80%, it creates a new DC (VM), adds it to the current farm, and remotely spawns a Worker in the new DC. We illustrate a particular run of this configuration in Figure 5.1(b) (NB: VM boot times are not subtracted from the result). Each asterisk $*$ in (b) is a point where the HeartBeater decides to create a new DC. This run stabilizes on 6 workers, which is a good approximation of maximum performance (according to Figure 5.1(a)), and possibly a good choice if we took into account any VM costs. As an extra, the HeartBeater could potentially `shutdown` machines if their load remained small (under a threshold) for a certain time.

The tests were conducted on the SURF cloud-provider with OpenNebula IaaS, on VMs with modern 8-cores, each with 8GB RAM and 20Gbps Ethernet. Interesting to mention is that each worker can benefit from ABS multicore (SMP) parallelism. A snippet of the HeartBeater follows with the full ABS code at our repository[2]:

[2]Upstream abs2haskell repository at `http://github.com/bezirg/abs2haskell`

```
class HeartBeater(List<Worker> farm, Balancer b) {
  Unit beat() {
    Rat avg = this.%*\textit{computeLoads}*)(farm);
    if (avg > 80/100) {
      DC dc = new NebulaDC(8,8192); // 8−core, 8GB RAM
      Worker w = dc spawns Worker();
      farm = Cons(w,farm);
      b ! updateFarm(farm); } } }
```

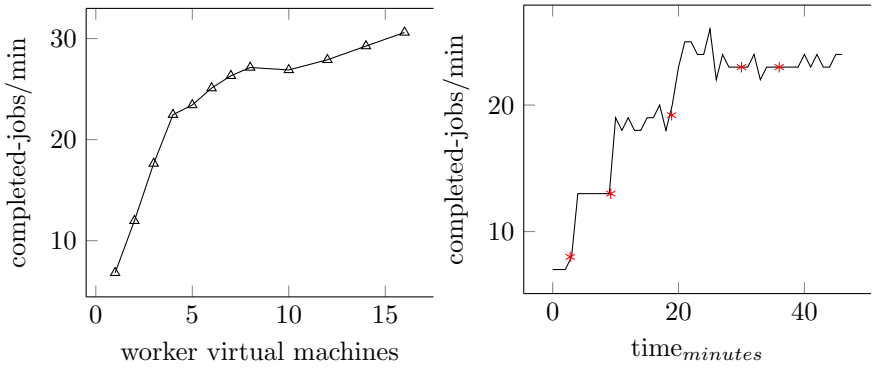

Figure 5.1: (a) Static deployment of VMs (b) Adaptive Deployment over time

For future work we are considering additions both at the language and runtime level. At the language level, it would be beneficial to include, besides the system load, other metrics such as memory, disk usage, object count, process count, exceptions raised (as partly done in the cloud simulation of Section 4.5). In this way, an ABS application would enhance its monitor and cloud-control logic. In a different direction, we plan to work on adding a basic *service discovery* mechanism to the standard library of ABS, as proposed by section 5.2, i.e. by extending the DC interface with two extra methods: an acquire(Interface obj) method that returns a reference to a remote object implementing the provided Interface ; an expose(Interface obj) that subscribes the passed object together with its current interface-view to the service registry of the DC.

At the system level, we are first interested in expanding our library support for other common cloud providers (such as Amazon EC2, OpenStack) and secondly providing user authentication for the cloud infrastructure. Besides the current open (peer-to-peer) topology of DCs we want to add support for other cloud topologies, such as provider-specific, slave-master, or supervision topologies – a

crude solution to topologies would be to introduce to the DC interface a method `List<DC> neighbours()` that lists all ABS nodes residing in the same private cloud network. In a different direction, we consider extending our virtualization technology support. With the introduction of micro-kernels (see the Xen hypervisor and unikernels), the cloud user no longer needs an OS underneath the application/service. By packaging the application into the kernel itself, the startup time of the VM is greatly improved, as well as its management and distribution. The Haskell Lightweight Virtual Machine (HaLVM) is a promising technology in this direction that allows the user to: "run Haskell programs without a host operating system". Likewise, *containers* (e.g. Docker), with its OS-level virtualization, would allow us to offer a more fine-grained control of deployment.

5.4 Case Study: Distributed Preferential Attachment

We ran the ABS-Haskell implementation of the PA algorithm by varying the graph size, on a distributed cloud environment kindly provided by the SURF foundation. The hardware consisted of identical virtual machines interconnected over a 10Gbps ethernet network; each Virtual Machine (VM) was a a single-core Intel Xeon E5-2698, 16GB RAM running Ubuntu 14.04 Server edition. The runtime execution results are shown in Fig.5.2; the execution time decreases while we add more VMs to the distributed system, which suggests that the distributed algorithm scales. However, even with 8 Virtual Machines the implementation cannot "beat" the execution time of one VM running PA sequentially; to achieve better performance, we may need to include more VMs. The reason for this can be attributed to the significant communication overhead, since each worker will send a network packet for every request call made.

On the other hand, the memory consumption (Table 5.1) is more promising: a larger distributed system requires less memory per VM. For example with the largest tested graph size, a distributed system of 8 VMs requires approx. 2.5 times less memory per VM than a local system. This allows the generation of much larger PA graphs than would otherwise fit in a single machine, since the graph utilizes and is "distributed" over multiple memory locations.

To improve the execution performance and time scaling, we further refined our approach to solving the PA problem by combining multiple request messages in a single TCP segment; this change increases the overall execution performance by having a smaller overhead of the TCP headers and thus less network communication between VMs, and better network bandwidth. In another (orthogonal) direction, we could utilize the many cores of each VM to have a parallel-distributed hybrid implementation in ABS-Haskell for faster PA graph generation, but this is left for future work.

This new improved version of the distributed PA algorithm is implemented as well in distributed HABS, [Bezirgiannis and Boer, 2016], with small high-level changes

	Total number of VMs			
Graph size	1	2	4	8
$n = 10^6, d = 3$	306	423	313	229
$n = 10^6, d = 10$	899	1058	644	411
$n = 10^7, d = 3$	1943	2859	1566	874
$n = 10^7, d = 10$	6380	9398	4939	2561

Table 5.1: Maximum memory residency (in MB) per VM.

	Total number of VMs				
Graph size	1	2	4	8	16
$n = 10^6, d = 3$	306	266	212	155	114
$n = 10^6, d = 10$	899	1028	547	354	221
$n = 10^7, d = 3$	1943	3242	1603	967	621
$n = 10^7, d = 10$	6380	9668	6702	3611	1905

Table 5.2: Maximum memory residency (in MB) per VM (refined approach).

to the model. Beside higher level of abstraction at the programming level thanks to our proposed improvement, the distributed runtime system provides more than $6x$ speedup performance compared to the same implementation without using the improvement, presented in [Azadbakht et al., 2017a]. The results of the refined approach are illustrated in Fig. 5.3. The distribution overhead increases the execution time for two machines, which is compensated by the parallelism achieved through adding more VMs. We managed this time to achieve positive speedup compared to a sequential algorithm implementation of one local machine, when using greater or equal to 8 virtual machines for distributing workload. As shown in the new memory results of Table 5.2, it is still the case that the memory consumption decreases by adding more VMs, which enables generating extra-large graphs which cannot fit in centralized-memory architectures.

5.5 Related Work

With the introduction of the Cloud, a plethora of cloud technologies and tools have appeared in the software community. We distinguish two categories of technologies related to our work: distributed-programming languages and cloud middleware.

5.5.1 Distributed programming languages

Erlang is one of the first distributed-oriented languages that next to the canonical message-passing communication, offers distinct features, such as hot-code loading and binary serialization of arbitrary closures — thus the capability to transfer them *over the wire*. This comes with a cost in safety since the serialized Erlang data are untyped and usually unversioned. The Akka framework brings distributed actors to the Scala language. Although Akka provides a rich library and toolkit, it currently lacks a cloud-aware API. At runtime The Java RMI (Remote Method Invocation) is a library bundled in the Java platform for communication between remote objects. The product pioneered in areas such as bytecode downloading and distributed-GC. The method invocation is strictly synchronous (the caller has to wait for the remote method to finish) and thread-unsafe. JADE[Bellifemine et al., 1999] is an active distributed multi-agent system also built in Java; agents are more expressive than actors at the expense of program complexity and, possibly, performance.

5.5.2 Cloud middleware and management

Ubuntu JuJu is a tool primarily for scaling and orchestrating a system's deployment on the cloud. Juju also comes with a GUI for modelling and visualizing a cloud deployment and saving it to a "recipe" for later reuse. It is usually accompanied by a configuration-management tool (such as Puppet) for the provisioning of cloud machines. CoreOS is a container-based OS that provides service and configuration discovery. It can be thought as a low-level infrastructure, primarily targeted to system administrators, for managing system services across a cluster of cloud machines, The Aeolus research project has built various tools that can derive an optimized deployment from the constraint-based model of a desired deployment, and automatically deploy that derivation. Finally, general SaaS supported by cloud providers eases the migration of existing software to the cloud and its automatic scaling of deployment. Albeit dynamic, a SaaS deployment can only vary on the CPU consumption, whereas our proposal would allow a much more expressive deployment that can depend on arbitrary application logic.

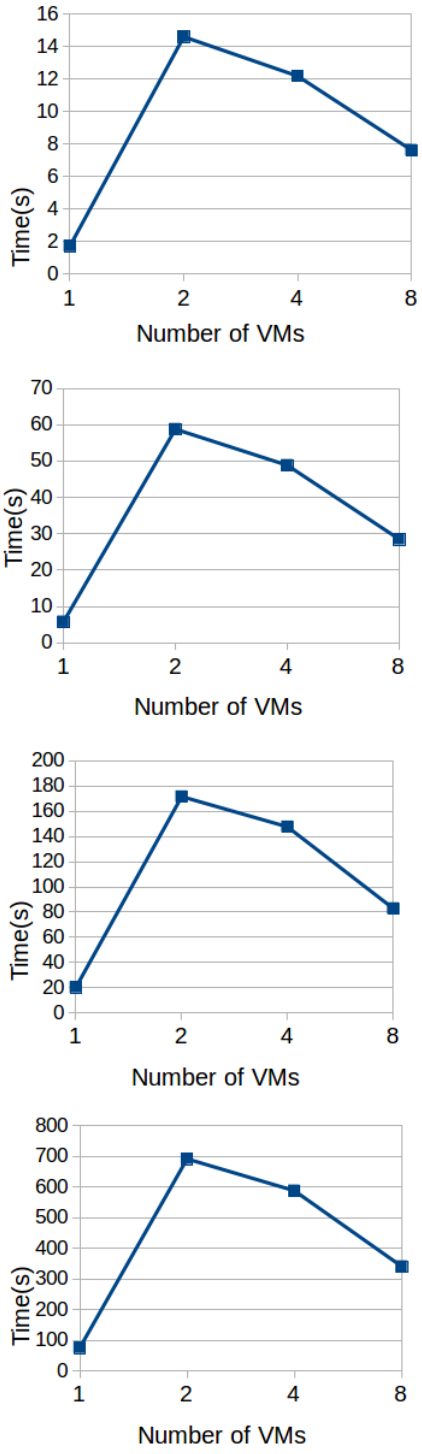

Figure 5.2: Performance results of the distributed PA in ABS-Haskell for graphs of $n = 10^6$ nodes with degree $d =$ (a) 3 , (b) 10 and $n = 10^7$ nodes with degree $d =$ (c) 3 , (d) 10.

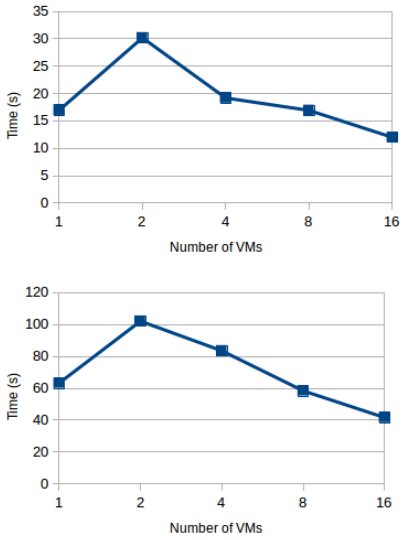

Figure 5.3: Performance results of the refined distributed PA in ABS-Haskell for graphs of $n = 10^7$ nodes with $m =$ (a) 3 , (b) 10.