JadexCloud - An Infrastructure for Enterprise Cloud Applications

Lars Braubach, Alexander Pokahr, and Kai Jander

Distributed Systems and Information Systems
Computer Science Department, University of Hamburg
{pokahr | braubach}@informatik.uni-hamburg.de

Abstract. Cloud computing allows business users to scale up or down resource needs according to current demands (utility computing). Infrastructures for cloud computing can be distinguished with respect to the layer they operate on. In case of platform as a service (PAAS) frameworks are provided in order to simplify the construction of cloud applications relying on common base abstractions and tool sets. The focus of current PAAS frameworks is quite narrow and directed towards support for web applications and also visual tools for non-programmers. We envision that cloud computing can also substantially push forward typical enterprise applications if these applications are made to exploit cloud capabilities. In this paper we present an infrastructure for developing, deploying and managing distributed applications with the vision of distribution transparency in mind. The infrastructure is meant to be on PAAS layer but addresses developers instead of end users. It is founded on the active component paradigm for distributed applications, which allows applications being composed of agent-like autonomous entities interacting via services. The model facilitates building scalable and robust enterprise applications due to the high modularity and independently acting modules that can be replaced or restarted if unexpected errors occur. In addition to the infrastructure, a real world scenario of a distributed workflow management systems will be explained.

1 Introduction

Cloud computing [8] is seen as a new approach to IT infrastructure management that facilitates a pay-as-you-go usage model. By making computational resources available on a demand-driven basis instead of statically devoting physical systems to certain applications, the approach minimizes wasted resources. Taking this idea further, it seems reasonable that existing computers in a company network should contribute their spare resources in a company private *enterprise cloud*.

Applications in the cloud can be built on the IAAS (infrastructure as a service) or the PAAS layer. With IAAS, access to the cloud is granted by virtual machines that allow a fine-grained control of the software stack to be used, including low-level aspects like the operating system. On the one hand the level of access does not restrict the application types deployable on the IAAS layer but on the other hand it does not contribute to any of the hard problems of how

to develop a complex distributed application. Using PAAS, the cloud operator establishes a new software layer with a dedicated middleware API (application programming interface) and in this way abstracts away lower-level details. This facilitates development of applications on top of the given platform, but on the other hand it firmly restrics the types of applications to those supported by the platform. Today, PAAS platforms are mostly targeted towards typical data-driven web applications with an additional focus on support for non-programmer interfaces.

Summarizing the IAAS and PAAS characteristics, an important gap can be identified for the systematical support for a wide range of enterprise applications in the cloud. This gap is only partially filled by existing enterprise solutions like application servers as these have not been conceived with cloud properties in mind and do not allow transparently exploiting additional resources of the cloud. To achieve the vision of a versatile private enterprise cloud two fundamental challenges remain:

- How to turn a highly dynamic environment consisting of volatile nodes into a robust, manageable cloud infrastructure.
- How to design and implement enterprise applications such that they are able to exploit the cloud characteristics.

These two challenges can be broken down into a number of more concrete requirements. With regard to the first challenge, the cloud middleware should require minimal installation effort and zero administration effort for the single nodes, otherwise it would not be feasible to include the many different types of computers usually found in a company network. Furthermore, the operation of the cloud infrastructure should not affect normal operation of the nodes, e.g. it should not restrict the way, an employee uses her computer. Therefore, the infrastructure has to deal with dynamically appearing and disappearing nodes, as employees turn on and switch off their computers. To support typical enterprise applications, the cloud environment needs to support administration tasks also for applications distributed in the cloud and therefore facilitate a transparent management of distributed applications as a whole. Finally, the deployment of applications should be efficient in terms of resource utilization, which requires monitoring the available resources and reconfiguring the deployment structure based on current application characteristics and infrastructure shape.

For addressing the second challenge, the infrastructure has to provide an intuitive programming model for distributed systems, which facilitates building applications such that they can be transparently partitioned and deployed in the cloud infrastructure. The computing model should also support distribution transparency in the sense of hiding complex distribution and concurrency issues. In summary, this paper aims at developing a distributed computing infrastructure for private enterprise clouds that meets the following requirements:

1. Require minimal installation and administration effort for the infrastructure
2. Support independently operated nodes and dynamic environments
3. Provide an intuitive programming model for distributed applications
4. Allow transparent application administration
5. Perform dynamic reconfiguration

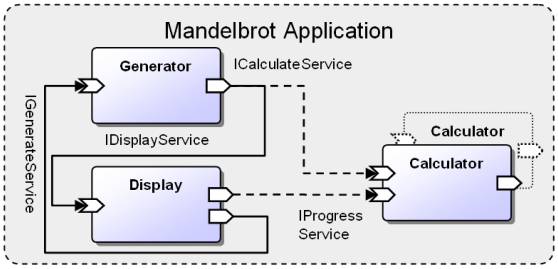

Fig. 1. Architecture of the Mandelbrot example application

Although all of the above mentioned challenges are vital for a full-fledged private enterprise cloud, the first three are sufficient for an initial solution showing the basic functioning of the cloud. Hence, in this paper mainly the first two questions are tackled and corresponding solutions are presented. An answer to question three with regard to an intuitive programming model for distributed systems has already been given as part of our earlier research work [7] and is only shortly recapped here. The underlying idea consists in using active components, which are software agents with features of components and services. Active components can be seen as extension of SCA (service component architecture) [6], which is a promising new paradigm for enterprise system development superseding traditional approaches like Java EE and has been pushed forward by influential industry players like IBM and Oracle.[1] The last two questions have been addressed only partially so far and are largely subject of future work.

The remainder of this paper is structured as follows. In Section 2 calculating Mandelbrot images will be introduced as a running example. Thereafter, in Section 3 the novel architecture of an agent-inspired cloud middleware will be presented. In Section 4 an extensive real world scenario from the area of distributed workflow management will be described. Section 5 reviews related work and in Section 6 a conclusion and an outlook to planned future work is given.

2 Running Example

Our approach is illustrated by a running example throughout this paper, that provides a complete application scenario but is simple enough to be easily understood. It is called *Mandelbrot* and allows users to render fractal images. To speed up the rendering process, the system should be able to distribute the computation across different hosts in a network.

The application is developed based on the active components paradigm introduced in [7] and the implementation is available as part of the open source Jadex active components framework.[2] Each active component represents an independent entity, that serves as unit of concurrency. The decomposition into components thus facilitates a later partitioning and deployment of applications in a distributed infrastructure. The interdependencies between the active components are made explicit by defining appropriate required as well as provided

[1] `http://www.osoa.org`

[2] `http://jadex-agents.informatik.uni-hamburg.de/`

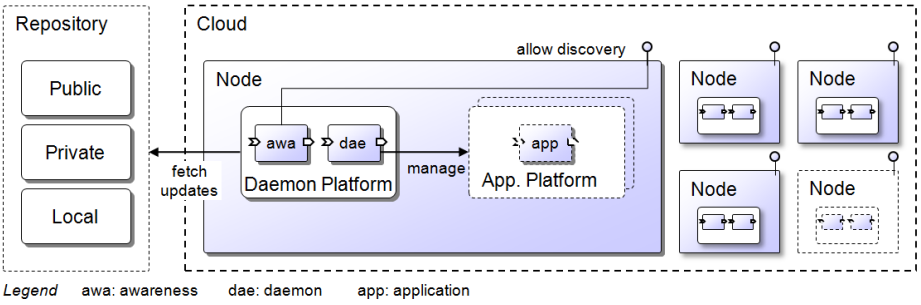

Fig. 2. Daemon layer structure

service interfaces, representing functionality that is publicly offered and used by a component. Compositions of an application are designed by specifying concrete bindings for connecting provided and required services at runtime.

Figure 1 shows the Mandelbrot application architecture. The *display* component provides interaction capabilities for a user of the system. It is responsible for presenting rendered images to the user and allowing the user to issue new rendering requests (e.g. by zooming into the picture or by manually entering area values and selecting a fractal type). The *generator* component handles user requests and decomposes them into smaller rendering tasks. It acts as a coordinator responsible for task distribution and result collection. The *calculator* component accepts rendering tasks and returns results of completed tasks to the generator. It implements different fractal algorithms and is thus able to provide the color values for pixels of the image to be rendered. These components are connected by respective required and provided service interfaces. E.g. the display component uses the *IGenerateService* to issue rendering requests to the generator component. The explicit specification of required and provided interfaces allows the application to be dynamically configured and adapted to the available resources in the infrastructure.

3 JadexCloud Architecture

Key concept of the proposed architecture is a layer model that helps separating responsibilities and managing complexity. It is composed of three layers. The *daemon layer* provides a minimal node infrastructure for basic management of cloud resources, e.g. automatically announcing available nodes participating in the network. On top, the *platform layer* supports application related management tasks including e.g. the deployment of application artifacts to different nodes as well as starting and stopping components. Finally, the *application layer* facilitates the application development by providing APIs and debugging tools.

3.1 Daemon Layer

The daemon layer (cf. Fig. 2) consists of different nodes on which a *daemon platform* is running in the background. The daemon platform represents the entry point for a node to the cloud. It has the purpose of facilitating the discovery of the underlying node, which can only be part of the cloud when announced by the

awareness agent (*awa*). Depending on the type of network the cloud should span, different announcement protocols have to be used. In case of a local area network, a simple TCP/IP multicast mechanism is sufficient in many cases, whereas more complex network setups also require more elaborated announcement methods.[3] Furthermore, the daemon agent (*dae*) provides a high-level service API for application management from the platform layer. This API allows for starting and stopping application platforms, on which application components (*App*) can be executed. The design is meant to enforce a strict separation between daemon and application execution in order to ensure long-lived manageability of the node even if an application is erroneous. During application execution the daemon can monitor the application platforms and terminate them whenever appropriate. The daemon platform also has access to repositories containing software bundles. Currently the repository is based on flat files, but the idea is to use a chain of Maven repositories to support versioning, etc. In this context it is distinguished between local, private and public repositories. A local repository is located directly on the node of the daemon, whereas a private repository is typically owned by an organization and shared by the member nodes of this organization. Public repositories have global scope and are thus available on Internet scale. The daemon uses these repositories for updating itself by regularly testing if new versions of its library are available.

In the following the role of the daemon layer is illustrated with respect to the running example of the Mandelbrot application. When a user wishes to deploy the Mandelbrot application on, e.g., a pool of workstation computers she has to make sure that all nodes can be discovered by the cloud infrastructure. Therefore, the minimal daemon platform has to be installed on each node. The platform registers itself e.g. as a unix daemon or windows service, such that it is started each time the host operating system is started. Therefore, the daemon has to be installed only once and needs no further attention afterwards. When assuming that the daemon is already present at each of the nodes, e.g. as part of a customized system distribution used for each pool workstation, then no administration tasks are required in the daemon layer for the Mandelbrot application.

3.2 Platform Layer

The purpose of the platform layer (cf. Figure 3) is to provide a management and execution infrastructure on which applications can be deployed and administered. It can be seen that the layer reuses many of the functionalities of the daemon layer, but introduces a different view. The entry point to the management functionality is an administration tool called *JCC* (Jadex Control Center) on an *administration platform*. To perform administration tasks, a user would typically start a local platform including a JCC. Yet, the choice of the administration platform is unrestricted as the JCC tool can be executed in principle on any platform. Based on local configuration options and user privileges, the

[3] In ongoing work Internet scale announcement and discovery is analyzed by utilizing existing peer-to-peer mechasims based on registries and superpeers.

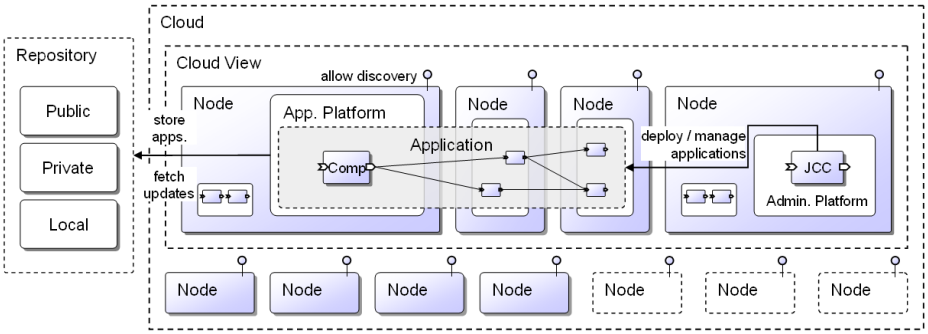

Fig. 3. Platform layer structure

JCC provides access to a subset of the existing nodes called the *cloud view*. The administrator can choose, which nodes to include in the deployment of an application, by assigning application components to the platforms running on the different nodes. To start the separate components, each platform will obtain the required component implementations from the local, private, or public repository. Alternatively, e.g. when deploying a new application, the JCC can upload component implementations to the remote platforms, which store them in their local repository. During the runtime of an application, the JCC tool can be used to connect to the platforms hosting the application components. Therefore, an administrator may at any time inspect the running components as well as alter the deployment configuration by starting and stopping components.

The operation of the JCC is further exemplified by illustrating the steps necessary to deploy the Mandelbrot application. The vision is that a user selects an application to deploy and is presented a cloud view of currently available nodes according to her profile. Based on an optional deployment description with requirements of the application components the system generates a deployment plan. By applying the plan the application is started creating components on the selected nodes. This vision as also covered in challenge 4 from the introduction is currently only partially realized. Once the JCC is started by the user, it will discover the available nodes based on the daemon layer awareness and filter them according to user preferences, e.g. a specific IP range. Using the daemon service, the user can start new application platforms on each node to host the Mandelbrot components. Afterwards the user can choose, which components to start on which nodes. For the Mandelbrot application the user would typically start calculator components on the remote nodes with the number of components on each node corresponding to the number of available cores.

3.3 Application Layer

The application layer is concerned with how a distributed application can be built based on the active components paradigm as well as providing tools for debugging and testing applications during development. The active component metaphor (cf. Fig. 4) comprises three aspects: a public interface describing provided and required services, an internal autonomous behavior, and service binding specifications. Provided services describe the externally available function-

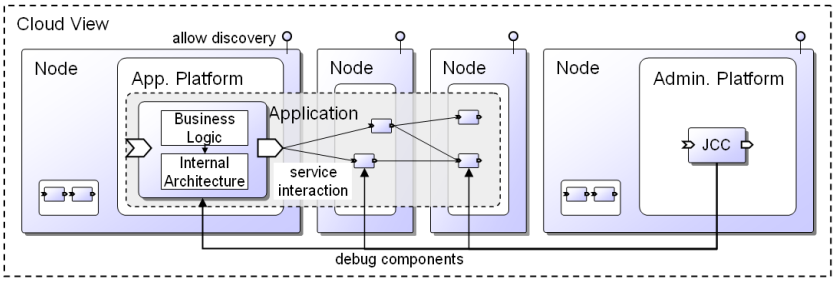

Fig. 4. Application layer structure

ality of the component, while required services explicitly state the dependencies of a component. The active components metaphor supports an agent-oriented view where components do not only passively carry out service requests, but are capable of autonomous behavior. The implementation of such behavior (i.e. *business logic*) is based on one of several supported *internal architectures* allowing to build e.g. BDI (belief desire intention) agents, simple task-based agents or BPMN (business process modeling notation) workflows. To establish a connection for *service interaction*, the active components runtime supports flexible binding mechanisms, which allow statically wiring components together, dynamically searching for available components and even creating new required components on demand. To enable the autonomous behavior of the component, each service request is decoupled from the caller and put into a request queue at the receiver side allowing a component to decide for itself when to react to a request. The active component runtime infrastructure is complemented by a suite of tools, included in the *JCC*, that support common development tasks such as debugging components, monitoring interactions, executing test cases, etc. These tools themselves are also realized using active components and are thus capable of operating transparently on local and remote components. Therefore a developer can debug remote components easily by starting the JCC on her local computer.

The Mandelbrot design from Section 2 has been implemented using the simple task-based agent kernel. Service interfaces and implementations are realized as plain Java files containing only application functionality, because the infrastructure is capable of adding the required glue for dynamic service binding automatically. The usage of the runtime tools is illustrated in Fig. 5a, showing the JCC while running the Mandelbrot application on three distributed nodes. The tree to the left shows the three platforms on the nodes (*alex_neu_896*, *workstation1*, *workstation2*) and the components on each platform. At the bottom, it can be seen that the main Mandelbrot application including the *display* and *generator* components is running on the local node, while each remote workstation runs two *calculator* components. At the top of the JCC there are tabs allowing to administer the remote platforms directly. To the right of the JCC, the debugger tool is activated for a remote calculator running on workstation1. The tool shows the current step of the agent as well as a history of previously executed steps and furthermore allows a stepwise component execution. In Fig. 5b

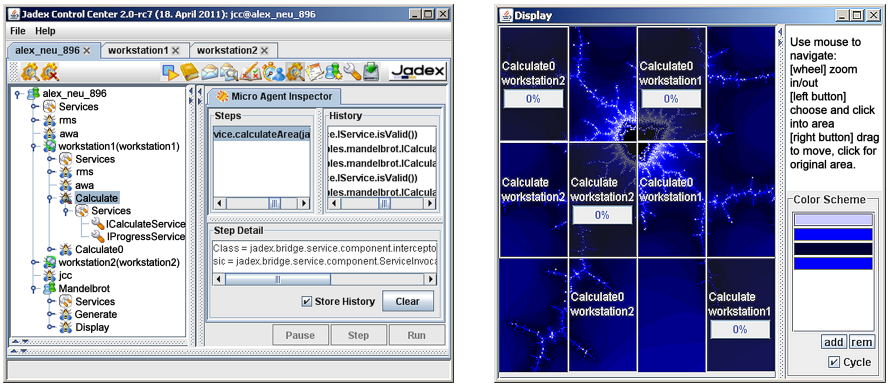

Fig. 5. Screenshot of the JCC (a) and the Mandelbrot user interface (b)

the user interface of the Mandelbrot application is shown, which illustrates the assignment of completed and ongoing tasks to the four calculator components.

Summary The previous sections have illustrated our vision and the current state of an infrastructure for a private cloud. The current implementation automatically discovers nodes in local networks and supports manual remote deployment of distributed applications based on the intuitive active components paradigm. In this respect the infrastructure represents a significant improvement for the development and management of complex distributed applications in e.g. company networks. Yet we regard it only as one step towards our ultimate private cloud infrastructure vision, which has to incorporate automatic monitoring and reconfiguration abilities as well as transparent remote application management.

4 Real World Scenario

Business Process Management (BPM) is an important topic for many organizations. *Workflow management systems* are widely deployed to automate and streamline business processes used within an organization. The purpose of such systems consists of managing workflow models, workflow execution, assignment and distributions of tasks (*work items*) and providing monitoring functionality, which allows management and process engineers to review workflow execution, improve the workflow models and intervene if there are problems [9]. Typical workflow management systems consist of centralized software which is deployed on a server and accessed using a web interface. This limits the flexibility regarding system configuration and makes it vulnerable to server breakdowns.

These shortcomings have been partially remedied by a distributed workflow management system implementation shown in Fig. 6, which is based on the cloud infrastructure presented in this paper. The system is partitioned into five components which can be distributed and replicated across multiple platforms and act together to provide the functionality of the system. The *access component* manages access to the system by external clients connecting to it, providing a unified interface to invoke system functions. User authentication and authorization is delegated to the *authentication component*. The access component uses dynamic

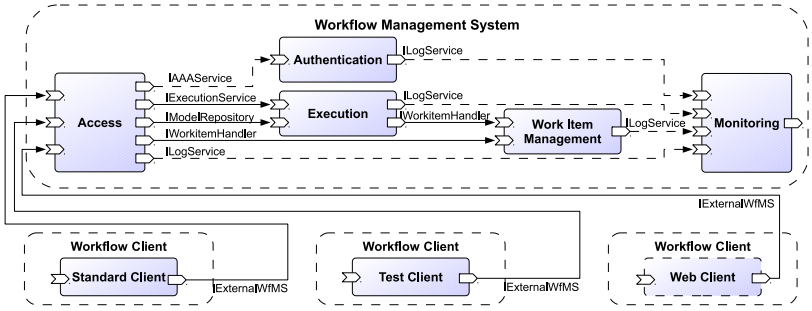

Fig. 6. Architecture of the workflow management system

service discovery to find available authentication services and therefore allows the authentication component to be replaced by an alternative at runtime.

Execution of workflows is handled by the *execution component*, including the workflow model repository and the execution service. The model repository uses the functionality of the platform layer to deploy workflow models which can then be started by the execution service. Executing workflows emit tasks to be performed by system users. Tasks are represented by *work items*, which are processed and distributed to the users by the *work item management component*.

Finally, the *monitoring component* receives events such as work item completion, user authentications and process execution from the other system services and stores them until accessed using the access component. Like the authentication component, this relationship is also dynamic and allows service substitution. Moreover, all available monitoring components are notified about events, allowing them to only store a partial set of the events enabling distributed storage.

Two *workflow clients* have been implemented which use the access component to interact with the system. The first is a standard workflow management client with a user interface enabling a workflow participant to process work items, monitor the system and perform administrative tasks, provided they possess the necessary privileges. The second client is an automated client which is used to test process models by repeated execution of test scenarios [4]. A third, web-based client is currently being developed. All of the clients use the service discovery of the infrastructure to find available workflow management systems and allow the user to choose any system available within the enterprise cloud.

As shown, both the workflow management system and the available client software make extensive use of the functionality of the infrastructure. This enables them to distribute the workload across multiple nodes, dynamically replace parts of the system and access remote functionality. Since the infrastructure manages the technical details, the implementation can be relatively simple, requiring little effort to enable a complex application to be distributed.

5 Related Work

The main rationale of the assessment of related approaches, stemming from grid and cloud PAAS, consists in evaluating them with respect to the five challenges from Section 1. In Fig. 7 an overview of the analyzed approaches is given.

	Platforms	Min. Install and Administration	Ind. Nodes in Dynanmic Env.	Transparent Distributed Apps	Dynamic Reconfiguration	Programming Model
Grid Platforms	GlobusToolkit	no / configurations	yes / registries	no / distributed view	yes / dyn. service discovery	SOA grid/web service
	ProactiveGCM	no / configurations	no / static infrastruc. desc.	no / distributed view	partially / static wires/res.mgmnt	active comps Fractal
	GridGain	yes / normal installer	yes / dyn. node discovery	no / centralized application	yes/ dyn. task assignment	oo & functional Java/grid tasks
	Boinc	yes / client approach	yes / nodes contact server	no / centralized application	yes / dyn. task assignment	oo C++/grid tasks
Cloud PAAS	GoogleApp Engine	yes / centralized access	no / known structure	partially / centralized apps	yes / dyn. resource alloc.	oo web applications
	Run@Cloud CloudBees	yes / centralized access	no / known structure	partially / centralized apps	yes / dyn. resource alloc.	components/oo JEE/Spring
	Paremus	yes / daemon approach	yes / dyn. node discovery	yes / holistic view	partially / rather static deploymnt	components SCA/OSGI
	JadexCloud	yes / daemon approach	yes / dyn. node discovery	partially / man. global view	yes / dyn. services&comps	active comps SCA extension

Fig. 7. Related approaches from cloud and grid computing

The original intent of grid computing approaches is exploiting computing power of other nodes e.g. for high performance computing. This motivation has led to client/server approaches with load being distributed from the server to clients. They perform the assigned tasks and send back the results to the server.

The underlying task distribution model with a rather centralized application has inspired several grid approaches like BOINC[4] and GridGain[5]. Other approaches like the GlobusToolkit [3] and Proactive [1] have extended the computational model from client/server to generic distributed applications. The programming model of the approaches directly reflects their application shape assumptions. While GridGain and Boinc use traditional object oriented techniques with tasks as primary abstraction for work distribution, Globus envisions a service oriented world consisting of introspectable and transient grid services, which will be created and terminated on demand. Proactive proposes using a model of components and active objects resembling very much the author's active component approach [7]. Regarding installation and administration complexity GridGain and BOINC address an easy integration of nodes offering installer-based solutions, whereas Globus and Proactive assume that an infrastructure model is explicitly set up, describing e.g. where registries are located and components should be deployed. The dynamics of environments is addressed by very different means. Globus uses service registries, GridGain uses dynamic node discovery based on awareness and BOINC uses a centralized server infrastructure. Proactive is rather focused on the static infrastructure model. Dynamic reconfiguration is supported by all approaches to some extent. GridGain and BOINC allow dynamic distribution of tasks taking into account the current grid structure. Globus allows dynamic binding of services by registry lookups and Proactive components can also be rebound by stopping and restarting them.

Cloud PAAS centers on efficient execution of specific application types, currently dominated by web applications. Typically, cloud PAAS hides distribution

[4] `http://boinc.berkeley.edu/`
[5] `http://www.gridgain.com/`

and concurrency aspects from developers and enable them to deploy a standard application in the cloud. The application can be scaled by the cloud infrastructure according to the customer demands. The approach is appealing but bounded by the narrow focus of existing PAAS infrastructures. Furthermore, today's applications have to follow vendor specific APIs for characteristics that are subject of scaling, e.g. the storage system. This easily leads to vendor lock-ins and problems in case scaling does not work as expected.

Google App Engine[6] and Run@Cloud from CloudBees[7] are two typical platforms in the direction described above. The first facilitates development of standard web applications while the latter supports Java EE enterprise applications. In contrast Paremus [5] is the only platform similar to JadexCloud targeted at general distributed applications for private clouds. The underlying programming models are object orientation in case of the Google App Engine, component orientation in case of Run@Cloud, and component service orientation (SCA) in case of Paremus. JadexCloud further advances the programming model to active components, which rely on a structural model very similar to SCA. Installation and administration requirements for the first two platforms are simplified by centralized web access interfaces to the cloud infrastructure. Paremus and Jadex pursue the idea of a dynamic cloud with a varying number of nodes. For this reason both use a daemon approach, meaning that a minimal bootstrapping software has to be installed on all nodes of the cloud. The cloud structure is rather well known for a typical cloud PAAS like Google App Engine and CloudBees and made fully transparent to the cloud user. Instead, Paremus and JadexCloud are built to deal with dynamic cloud infrastructures allowing nodes to be discovered and dynamically included or excluded to/from the cloud. Thus, the application view is different as well. The first two keep the standard non-distributed application view and use internal logic for scaling, the latter two handle truly distributed applications, meaning that functionally different parts of applications run on different machines. Despite the application distribution both, Paremus and JadexCloud, aim at a high-level view on the distributed application being able to abstract from distribution aspects. All four platforms deal with dynamic reconfiguration of applications. In the Google App Engine and CloudBees, reconfiguration tasks are completely transparent. Paremus and JadexCloud have to deal with more complex situations due to the possibly fine-grained application deployment structure. Relocations of components are difficult to achieve in Paremus due to its reliance on SCA. In JadexCloud, relocations can be achieved, as active components support dynamic service bindings.

Summarizing, most grid toolkits are built for dynamic environments and allow for runtime grid adaptations but suffer from too simple programming models (except Proactive) based on object orientation or services. On the contrary, cloud PAAS infrastructures are typically built to make standard (web) applications scalable and are narrowly focused (except Paremus). They consider applications as being non-distributed and use established programming models like object

[6] `http://code.google.com/intl/de-DE/appengine/`
[7] `http://www.cloudbees.com/run.cb`

orientation and components. Approaches like Paremus and JadexCloud go beyond the typical cloud PAAS and facilitate building distributed systems running in a dynamic infrastructure.

6 Summary and Outlook

This paper has argued that enterprises can benefit from private clouds to perform heavy computational tasks allowing a smooth scaling according to current demands. Such a private cloud may not only consist of dedicated servers but may also include normal computers used for daily work. In order to run applications in such a dynamic cloud it is necessary to a) create a robust infrastructure from a varying set of nodes and b) use a programming model allowing applications being assembled of independent components that are dynamically coordinated.

As a solution the JadexCloud infrastructure has been presented, consisting of three layers. The daemon layer is responsible for managing basic cloud resources by e.g. announcing nodes participating in the network. The platform layer is used for application management tasks such as deployment and starting of application components. The application layer supports application development through APIs and tools. Using a workflow management application example it has been shown how a distributed cloud application can be built.

As future work mainly the remaining challenges for allowing transparent application administration and performing dynamic reconfigurations will be tackled. The first requires a conceptual abstraction for distributed applications and their components, possibly inspired by earlier work [2], while the latter has to cope with collecting and evaluating non-functional node data such as performance, utilization and uptime in order to reorganize a running application.

References

1. F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, and C. Pérez. Gcm: a grid extension to fractal for autonomous distributed components. *Annals of Telecommunications*, 64((1-2)):5, 2009.
2. L. Braubach, A. Pokahr, D. Bade, K.-H. Krempels, and W. Lamersdorf. Deployment of Distributed Multi-Agent Systems. In *Proc. of the 5th Int. Workshop on Engineering Societies in the Agents World (ESAW 2004)*. Springer, 2005.
3. I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. Technical report, Global Grid Forum, 2002.
4. K. Jander, L. Braubach, A. Pokahr, and W. Lamersdorf. Validation of agile workflows using simulation. In *Third international Workshop on LAnguages, methodologies and Development tools for multi-agent systemS (LADS010)*. CEUR, 2010.
5. Paremus Ltd. The paremus service fabric: A technical overview, 2009.
6. J. Marino and M. Rowley. *Understanding SCA*. Addison-Wesley, 2009.
7. A. Pokahr, L. Braubach, and K. Jander. Unifying agent and component concepts - jadex active components. In *Proceedings of the 8th German conference on Multi-Agent System TEchnologieS (MATES-2010)*. Springer, 2010.
8. B. Sosinsky. *Cloud Computing Bible*. Wiley, Indiana, USA, 2011.
9. Workflow Management Coalition (WfMC). *Workflow Reference Model*, 1995.