Component-based Scalability for Cloud Applications

Steffen Kächele, Franz J. Hauck
Institute of Distributed Systems, University of Ulm, Germany
{steffen.kaechele,franz.hauck}@uni-ulm.de

Abstract

Cloud computing enables access to an almost unlimited amount of resources combined with usage-based accounting. However, due to their design a lot of applications are not able to exploit the elasticity provided by the cloud. In this paper, we introduce several mechanisms that allow exploitation of the component structure of applications in order to scale them in a cloud computing cluster. We present our OSGi-inspired component framework COSCA that automatically manages elastic deployment of component-based applications. It isolates components of different applications and hides distribution using a virtualized and distributed OSGi-like framework. We present the results of several experiments which show that scalability of component-based applications benefits from such a platform. Moreover, we show how lightweight and agile component-based scale-out is. Our approach eases the usage of cloud resources and scalability for component-based applications.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures—Domain-specific architectures; D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks

Keywords PaaS, cloud computing, scalability, elasticity, component, OSGi

1. Introduction

Cloud computing increases the flexibility of companies as they can move away from on-premise hosting to large scale hosting providers. Tenants can use a huge amount of resources with no need for a large initial investment. Another big advantage of cloud computing is elasticity. One problem, however, is to deploy and scale applications so that they can benefit from the sheer unlimited resources. In early cloud computing systems (i.e. IaaS), the resource allocation is manual. Tenants have to explicitly add resources, whereas the resource allocation in these systems is usually very coarse-grained (i.e. completely virtual machines). Thus, tenants have to split applications into pieces, deploy them onto virtual machine images, develop automatic resource allocation and do many administrative tasks such as application reconfiguration for seamless scaling, which is a complex task. In the past, a lot of new programming paradigms were developed towards scalability support for applications [1, 2]. New languages or even new programming models (e.g. Map Reduce) were designed. However, they fundamentally change the way applications are built and do not address the huge amount of existing code. They need a complete redesign of existing applications and finally do not match all applications.

In this paper, we argue that the component structure of well-designed software can be exploited to provide scalability in cloud computing systems. We propose a PaaS system on which component-based applications could be uploaded, are analyzed with respect to the component structure and used for distribution. It fully automates resource provisioning and deployment. We show how to monitor interaction between software modules, thus being able to decide when the ideal time is for scaling in or out. Our framework manages the distribution between cluster nodes and uses service interaction on node edges. As a consequence, resource allocation is implicitly provided. As scalability does not depend on large deployment units such as virtual machines, it is quite fine-grained and thereby very agile. Thus, it becomes possible to adapt applications in real-time to changes in load, infrastructure and objective function. Our prototype uses Java and a component model equivalent to OSGi [13].

Section 2 of this paper summarizes and discusses issues with respect to scalability. In Section 3, we introduce the OSGi programming model. Section 4 sketches the conceptual background of our component-based approach that is implemented and evaluated in Section 5. Before we conclude, we present related work in Section 6.

2. Scalability using a component structure

A major feature of cloud computing is *elasticity*. Applications that are compatible with the cloud model can make use of nearly unlimited resources. Elasticity is typically achieved

CloudDP '13, April 14, 2013, Prague, Czech Republic.

by scaling applications code. In general, scalability can be vertical or horizontal. *Vertical scalability* is addressed by increasing the power of nodes whereas *horizontal scalability* uses more nodes for the same job. The latter is mainly used in cloud computing.

One way to provide horizontal scalability is to replicate application code onto multiple nodes so as to distribute workload to available instances. Both the unit of replication and of distribution typically depend on the deployment unit. A IaaS deployment, e.g., can provide scalability by replicating virtual machines. Distributing and replicating a large amount of data, however, creates a lot of overhead and the way the platform can respond to workload peaks is rough.

The approach presented in this paper proposes a more fine-grained and thus agile structure by distributing *components*. Components are self-contained pieces of software. They are generally considered to be larger units of composition than objects [12]. In well-designed software, these components have high cohesion internally and low coupling to the outside. They encapsulate a dedicated functionality accessible via a special interface and are thus ideal parts for distribution. Distributing application on component level has many advantages. Using components, a platform can replicate exactly those parts of the application that form a bottleneck by introducing a minimum of overhead. On the other hand, when components use a lot of static resources (e.g. memory), the platform can keep the amount of instances low and improve its utilization by redirecting requests from many consumers.

3. OSGi

OSGi [13] is a common industry standard to modularize applications. It allows installation, updating, and uninstallation of software components at runtime without the need to restart the entire platform. Thus, it perfectly matches long running server applications that should remain reconfigurable.

The OSGi platform consists of multiple layers that provide different abstractions of interaction between components. The *module layer* is the lowest logical layer of OSGi. It defines the basis of the module concept. In OSGi, application code is parted into modules called *bundles*. They contain a manifest file describing bundle properties such as bundle name and version. Initially, bundles are completely isolated from each other. However, code sharing on the basis of Java packages is possible between bundles. For this purpose, the bundle manifest can specify which code is to be imported from other bundles and which of its own code is accessible for other bundles. OSGi framework implementations realize the isolation of bundles by assigning each bundle a separate Java class loader. It connects class loaders according to the imports in the respective bundle manifest.

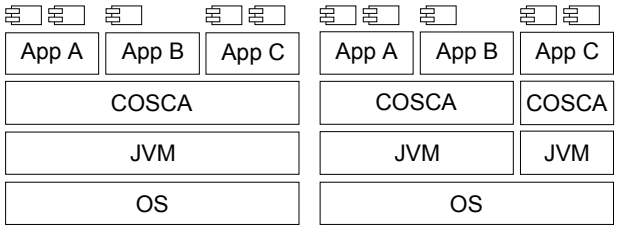
The *service layer* realizes a services-oriented and thus loosely-coupled system. The major entity of this layer is the *service registry*. It allows bundles to register services under a certain interface. Other bundles can look up registered services in the registry and use them to implement their required functionality. Bundles are allowed to register and unregister services at any time. For this purpose, OSGi provides an event concept to notify service users about changes.

The *distributed OSGi specification* extends the OSGi model. It allows services running in another container to be accessed. The container can even run on another node. Remote services can be accessed as a usual service via the service registry. However, this does not cover scalability and cases with one consumer and multiple providers. When using a service proposed in the distributed OSGi specification all registered services appear as individual entities in the registry and thus it remains the task of the application to manage scalability and choose the references for each request. Yet, it provides no support for automatic scale-out.

4. COSCA

COSCA is a Java-based PaaS platform that uses an OSGi-inspired programming model [8]. It is designed to bring OSGi functionality to large scale systems.

COSCA aims at hosting multiple OSGi-based applications on shared third-party hardware. For deployment and management, it introduces the concept of applications that are composed of multiple bundles (i.e. components). COSCA applications can be directly deployed onto the platform. Being aware of this relationship, bundles that belong to an application are treated as a single entity. Yet, deploying multiple applications on a single node requires separation. For that purpose, the COSCA platform provides isolated OSGi-like environments for each application. We call these environments *virtualized frameworks*. Virtualized frameworks are very lightweight and aware of the mapping of bundles to applications (i.e. they only wire bundles of the same application). As multiple frameworks can run in the same JVM, creating a virtualized framework generates a very low footprint. COSCA can thus host multiple applications on a single host without using an additional IaaS layer that causes additional overhead. To separate applications, the platform uses different mechanisms on both module and service layers. On the module layer, it uses mechanisms of classloaders and security managers to separate applications. For isolation, the framework ensures that classloaders of different applications can only access their own specific code base and are not connected during wiring. Thus, bundles can only import packages that they export in an appropriate framework. Furthermore, we use security-manager features to prohibit access to Java functionality that may endanger isolation, and to limit access to dedicated resources (e.g. only parts of the file system). On the service layer, each application has its own service registry. Thus, when an application looks up a service, it will only see those it has registered before. Similarly, framework, bundle, and service events are only



(a) One instance (b) Multiple instances

Figure 1. Deployment and Architecture

published to the application of the corresponding tenant. An optional runtime layer [8] allows for additional separation, such as separating threads, file system, and network.

Fig. 1 summarizes the architecture. One option is (a) to run COSCA as a single entity on a computing node. It provides a central Java-based platform to deploy multiple applications in this case. When an application requires higher isolation (e.g. to prioritize application execution on OS level), it is (b) also possible to separate an application by spawning another dedicated JVM on this node. In this case, we have multiple instances of the platform running on one node. The COSCA frameworks with multiple nodes form a distributed PaaS system. Applications may have bundles that are deployed on multiple nodes. This allows us to address load balancing and scalability.

4.1 Component-based scalability

The OSGi service platform concept provides an elegant way of deploying component-based applications. However, current frameworks lack functionality to exploit scalability in the cloud. This includes platform support for distribution, automatic load balancing, and a mechanism that decides when to scale. It also requires automatic deployment support when the platform scales out.

For that purpose, we introduce a number of services for scalability support. In the following, we describe services that can be used to distribute applications among multiple nodes and to transparently access them using remote services. We describe strategies on how to handle services that exist on the local as well as on distributed nodes.

Distributed Service Registry Our distributed service registry is the central unit that manages and eases distributed deployment of applications. It supports adding the capacity of an additional node to an application by seamlessly extending a virtual framework among multiple nodes. It is aware of redundant service instances in case of replication. When applications get deployed on different nodes (e.g. during scale-out), our distributed service registry connects the virtual frameworks by synchronizing the services that are available for an OSGi application. Whenever a service gets registered, the registry automatically distributes the service by exporting it via a distribution provider (i.e. remote service specification) and publishes it within an application-specific distributed registry. Thus, the platform can handle applications that exceed the resources of one physical node, e.g., when the node does not have enough memory. Another example is an application that has high workload in some components. Replicating these components to an additional node is one solution in this case.

Platform support for elasticity In order to provide platform support for elasticity, we introduce the concept of *virtual services* that addresses two different issues. The first issue affects scaling support. To distribute applications among different nodes, we use the capabilities of the remote admin service that allows access of services on a remote host via a URL using an OSGi-compliant RPC. These services, however, appear as dedicated units in the service registry. In this case, the application has to cope with scalability which is complicated, error-prone and is not applicable to legacy applications. The second issue concerns the statically designed references to services. When a service consumer looks up a service in an OSGi framework, it will get a direct reference to the service implementation. As a consequence, the platform is no longer able to balance requests to another instance or to transparently migrate components within the cluster.

For supporting scalability and migration, virtual services decouple service consumers from service providers. Virtual services are automatically created by our distributed service registry and are a facade to real services. Our platform thus decouples the logical service reference from the service instance. Decoupling provides flexible mapping between the service reference and the actual service. Initially, in a non-replicated application, virtual services point to local services. When scaling, they can add further service instances to the pool of services (see Fig. 2). For scaling support, they support strategies such as a random strategy, a weighted random strategy (incorporating the workload of nodes), or even a strategy with stickiness to address state.

Monitoring and automatic deployment When deploying component-based applications, one challenge is to detect and locate overload situations. Whenever a node gets overloaded it becomes noticeable due to typical operating system parameters such as high CPU usage or high memory consumption. Whereas this is enough in virtual-machine–based deployments (e.g. IaaS), detection in a component-based application gets more complicated as there is no direct mapping of component utilization to operating system parameters. For a fine-grained automatic scale-out it is essential to locate bottleneck components. For that purpose, our platform includes a *monitoring service* observing service invocations that represent the interaction between components. To estimate utilization of a component, we can measure either the time an invocation takes or the resources such as CPU time a thread in a service call consumes. The platform thus knows which component consumes how much resources. When components consume a high percentage of local resources, it may be replicated. When resources of a

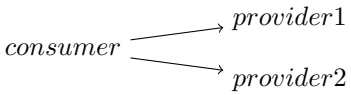


Figure 2. Virtual Services

Figure 3. Plot Application

node are exhausted, components are migrated. When monitoring resources, nested calls (i.e. when a service calls another service) are not easy to handle. Our monitoring service, therefore, uses a stack-based algorithm that stores CPU time consumed by nested invocations in the current stack frame. Thus the exact resource usage of the current component can be deduced by subtracting resources consumed outside, and overload situations can be detected.

To enable automatic scale-out and scale-in, we developed an internal REST-based API that allows frameworks to control the life-cycle of both applications and dedicated bundle instances of other frameworks (e.g., install, start, stop, update, and uninstall). An additional web interface allows users to manage and monitor applications.

4.2 Optimizations

Our platform has further optimizations to handle component distribution.

Coupling of components Components can have different types of coupling. Whereas loosely coupled components are ideal to scale, tightly coupled components may incur high communication load and dependability issues when distributing (e.g. when importing packages via the OSGi module layer). Depending on the coupling it may or may not be better to distribute components. For that purpose, our platform has a mechanism that detects the coupling between components. It mainly uses facts from OSGi wiring and investigates inter-component package imports. When there is high coupling (e.g. importing an implementation of a class), we treat these components as one unit which can only be scaled together. Conversely, the platform marks components with low coupling as scalable.

Addressing state When scaling services, state may become a challenge as it directly affects the possibilities of distribution and scaling. Whereas requests to stateless services can be arbitrarily distributed to available instances, multiple requests to stateful service have to be routed to the same service. On our platform this issue reflects on the component composition. The platform supports the issue on both deployment (i.e. component) and service (i.e. request) level. On the component level, the platform allows single components to be prohibited from scaling. Indeed, this limits the scalability of applications. In cases where it decreases performance, developers can further optimize their applications. They can exploit the component structure to separate state from computational components, e.g. by outsourcing them into a state management component. A particular extract is to use a service that manages state in the backend. Depending on the state, a—possibly replicated—database service can store state. A distributed coordination service (e.g. ZooKeeper) and the configuration admin service proposed by OSGi are examples of that. On the service level, the platform can use a sticky policy and wire single service consumers to distinct service providers so that they access the same state.

5. Implementation and Evaluation

In our implementation, we integrated the proposed services into our cloud platform COSCA [8]. We use R-OSGi as distribution provider and Zookeeper [7] as backend for our distributed service registry. Management services such as the backend and internal services for topology and infrastructure information run in an additional administrative application on a dedicated virtualized framework. As these services should stay highly available, we allow replication of the services to multiple nodes (in conjunction with Zookeeper).

For our evaluation, we use two nodes equipped with an Intel Core2Duo CPU with 4 GB of main memory connected by a one gigabit Ethernet switch.

5.1 Experiment A

In our first experiment, we deploy an application that visualizes data by plotting a given dataset into a PNG image. Fig. 3 shows the component structure of this application. A servlet-based frontend forwards requests to an images service that plots the image using the open source Java library JFreeChart.[1]

Methodology We use three nodes to generate a synthetic workload. Each node simulates eight clients that simultaneously access the web service. As a result, we simulate 24 clients that continuously request plots.

Experiment A1 We first run the application on one node. In this setup, we achieved 61 requests per second (Table 1). When stressing the application, the platform detects high workload in the image service and replicates the corresponding bundle to an additional node. After scale-out, we measured 118 requests per second (193.8 %). Although we use a simple random strategy that equally distributes the requests to available nodes, the performance almost doubles.

5.2 Experiment B

In our second experiment, we deploy an ordinary OSGi application that was originally not developed for scalability. It calculates ideal routes between given waypoints on a map.

[1] http://www.jfree.org/jfreechart/

	one node	two nodes
request/s	61 ± 2.35	118 ± 4.01
relative	100%	193.8%

Table 1. Experiment A1

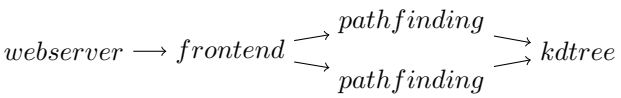


Figure 4. Navigation Application

Users can provide routes via a map on a website provided by the frontend bundle (see Fig. 4). The frontend uses a webserver to host the servlets. The path-finding service calculates the route using a contraction hierarchy algorithm [5]. It uses a KD-tree to map GPS coordinates to an internal data structure. This data structure is very efficient but consumes a lot of memory.

Methodology In our experiment, we use three nodes to generate a synthetic workload. Each node simulates 20 clients that simultaneously access the web service. As a result, we simulate 60 clients that continuously request routes.

Experiment B1 We first measured the speedup when this application is scaled-out due to high load (see Table 2). When running the application on a single node, we observe an overall throughput of 194 requests per second. After our monitoring service detects the high workload at the path-finding service, it scales it out on an additional node. As both the KD-tree and the frontend bundle cause relatively low workload, the platform will not replicate them. Especially not replicating the KD-tree saves a lot of memory. After automatic scaling, we measure a throughput of 312 routing requests per second (160.8 %). The difference (39 %) to linear scalability is twofold. On the one hand, each path-finding service has to communicate backwards to the KD-tree to resolve coordinates. This causes unnecessary overhead in a distributed environment. Modifying the path-finding service to start routing with already resolved coordinates may clearly improve performance. On the other hand, we also observe that while the first computing node is fully utilized the second is not. The reason is that we use a random strategy which uniformly distributes requests to available path finding services. As the first node has to manage webserver, frontend, KD-tree and one of the path-finding instances, it has more workload than the other node hosting just an instance of the path-finding service.

To address such a bottleneck, we may use a load-aware strategy and distribute requests with respect to the workload of nodes. Using such a strategy, the additional nodes would receive more workload in contrast to the node hosting the webserver, frontend bundle, and KD-tree. A deployment strategy could also improve utilization. For example, we could migrate the bundle with the routing service to another node so that the first node only hosts the webserver, frontend bundle, and KD-tree whereas the path finding service is hosted on the additional acquired nodes. Evaluation of these options is topic of further research.

	one node	two nodes
request/s	194 ± 2.68	312 ± 4.35
relative	100%	160.8%

Table 2. Experiment B1

	scale application	scale service
time(ms)	3212	64
relative	100%	2%

Table 3. Experiment B2

Experiment B2 In this experiment, we compare the time needed to scale-out individual components in contrast to a complete application. We measure how long it takes to start a new replica of the path-finding component compared to the time for starting a second routing application containing all components. The latter would be the default for scale-out in standard clouds. Table 3 shows the results. For our *path finding service*, we measure a startup time of 64 ms. In contrast, a complete application start up takes about 3200 ms, mainly caused by initializing the KD-tree. As a result, a component-based scale-out is about 50 times faster than replicating the complete application. Further benefit can be observed when comparing with a virtual machine replication in a IaaS setup.

6. Related work

In recent years, several new programming paradigms have been developed to support scalability and to map computation onto a cluster of nodes. One popular paradigm is Map Reduce [2] that Google introduced in 2008. It allows large jobs to be scale by splitting them into independent smaller ones to distribute them. Other examples are task processing systems from grid computing such as the Globus Toolkit [4] that uses web-service technologies to distribute grid jobs to a computational cluster. However, they only address computational jobs and do not match server applications. Message passing systems such as OpenMPI [3] are another mechanism to achieve scalability. In comparison to our work, they lack a mechanism for component composition and their programming model does not provide support for application deployment [10]. CORBA [18] introduces a component model [11] and proposes some parallel extensions [15], but the tight coupling between objects with respect to dependencies hinders flexibility and adaptability of applications. A^2-VM [17] and JESSICA [9] are JVMs to execute applications on multiple nodes. Instead of components, they use threads as a distribution unit combined with a distributed shared memory.

There are already some commercial and open source platforms available that fully manage deployment. ConPaaS [14] is a PaaS platform that allows deployment of general server applications that consist of multiple parts. In contrast to COSCA, it uses a coarser granularity for composition (processes). As a consequence, to scale an application in ConPaaS it has to be compiled to multiple processes that may communicate through IPC mechanisms such as sockets. In COSCA, we use software components for distribution. We exploit the fact that in software engineering well-designed software already consists of multiple components that are loosely coupled. These components may reside in one process as long as they are running on one node. During scaling, we implicitly add IPC mechanisms between components. Google App Engine[2] provides a Python, Java and Go environment. Yet, it limits the programming model to web servlets and developers have to use the AppEngine's proprietary APIs that restrict application portability so as to run them exclusively in the Google cloud. Currently, it does not provide high-level support for composing applications from fine-grained components. OpenShift Flex[3] and Cloud Foundry[4] are open source PaaS systems. Yet, none of them supports component-based deployment and scaling.

The cloud provider Force.com proposes APEX [1], a new programming language to support scalability. New programming paradigms, however, require a fundamental redesign of existing applications. CLOUDdep [16] introduces a service to support easy replication and recovery from failures in OSGi cloud environments. For replication, they use a micro boot mechanism. In contrast to our work, they focus on a service that makes OSGi service references dependable.

7. Conclusion

Scaling of applications is not an easy task. In this paper, we propose an approach to exploit a component-based structure to scale applications within a cloud computing cluster. We introduce services that allow transparent resource allocation as well as distribution of components on multiple computing nodes. They monitor distinct components of an application and decide when to scale out and in. Our prototype uses Java and an OSGi model. It fully automates elastic cloud resource provisioning and does not require any manual setup. As there are OSGi framework implementations for a wide range of devices, cloud applications remain portable and thus it is easy to move them from on-premise hosting to the cloud and vice versa.

As cloud environments are often built up from commodity hardware, they tend to fail. In future work, we will evaluate how we can extend our methodology to such undependable environments in order to provide fault tolerance [6].

[2] http://appengine.google.com

[3] http://openshift.redhat.com/app/flex

[4] http://www.cloudfoundry.com

References

[1] Apex: Salesforce on-demand programming language and framework. http://developer.force.com/.

[2] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Comm. ACM*, 51(1):107–113, 2008.

[3] E. Gabriel et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *LNCS*, pages 97–104. Springer, 2004.

[4] I. Foster. Globus toolkit ver. 4: Software for service-oriented systems. *J. of Comp. Sci. and Techn.*, 21:513–520, 2006.

[5] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: faster and simpler hierarchical routing in road networks. In *Proc. of 7th Int. Conf. on Exp. Alg.*, WEA'08, pages 319–333. Springer, 2008.

[6] F. J. Hauck, S. Kächele, J. Domaschka, and C. Spann. The COSCA PaaS platform: on the way to flexible and dependable cloud computing. In *Proc. of the 1st Europ. Workshop on Dep. Cloud Comp.*, EWDCC '12, pages 1:1–1:2, NY, USA. ACM.

[7] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proc. of the 2010 Ann. USENIX Techn. Conf.*, USENIXATC'10, pages 11–11, CA, 2010.

[8] S. Kächele, J. Domaschka, and F. J. Hauck. COSCA: an easy-to-use component-based PaaS cloud system for common applications. In *Proc. of the First Int. Workshop on Cloud Comp. Platforms*, CloudCP '11, pages 4:1–4:6, NY. ACM.

[9] M. J. M. Ma, C.-L. Wang, and F. C. M. Lau. Jessica: Java-enabled single-system-image computing architecture. *J. Parallel Distrib. Comput.*, 60(10):1194–1222, 2000.

[10] Malawski et al. Component-based approach for programming and running scientific applications on grids and clouds. *Int. J. of High Perf. Comp. Appl.*, 26(3):275–295, 2012.

[11] R. Marvie and P. Merle. Corba component model. TR, 2002.

[12] P. Mougin and C. Barriolade. Web service, business objects and component models. Techn. Report, Oct. 2001.

[13] OSGi Alliance. OSGi service platform core spec. 4.3, 2011.

[14] G. Pierre and C. Stratan. ConPaaS: A platform for hosting elastic cloud applications. *Internet Comp., IEEE*, 16(5):88–92, 2012.

[15] C. Pérez, T. Priol, and A. Ribes. A parallel CORBA component model for numerical code coupling. In *Grid Comp.*, volume 2536 of *LNCS*, pages 88–99. Springer, 2002.

[16] J. Rellermeyer and S. Bagchi. Dependability as a cloud service - a modular approach. In *IEEE/IFIP 42nd Int. Conf. on Dep. Sys. and Netw. Workshops (DSN-W)*, pages 1–6, 2012.

[17] J. Simão, J. a. Lemos, and L. Veiga. A2-VM: a cooperative Java VM with support for resource-awareness and cluster-wide thread scheduling. In *Proc. of 2011th Confederated Int. Conf. on on the Move to Meaningful Internet Sys.*, OTM'11, pages 302–320. Springer, 2011.

[18] S. Vinoski. Corba: integrating diverse applications within distributed heterogeneous environments. *IEEE Comm. Mag.*, 35(2):46–55, Feb. 1997.