Introducing Scalileo

A Java Based Scaling Framework

Tilmann Rabl, Christian Dellwo, Harald Kosch
Chair of Distributed Information Systems
University of Passau, Germany
{rabl,dellwo,kosch}@fim.uni-passau.de

ABSTRACT

Scalability is a major concern of internet based applications. Access peaks that overload the application are a financial risk. Therefore, systems are built to scale. They are usually configured to be able to process peaks at any give moment. This can be very inefficient. Yet, there are various ways to improve efficiency. One reasonable approach is to scale applications according to their current workload. This requires the possibility to scale a system up and down. In this paper we present an scaling framework for Java applications. It allows not only autonomic scaling, but also migration of distributed applications. We will then show how energy efficiency can be increased by scaling applications. To present an example we have used our framework to autonomically scale a web server cluster.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms

Scaling Framework, Energy Efficiency, Distributed Applications, Autonomic Scaling

1. INTRODUCTION

In order to increase the efficiency of a distributed system local as well as global optimizations should be taken into consideration. Local optimizations try to enhance the efficiency of a single system, while global optimizations capture the distributed system as a whole. Examples for local optimizations for energy efficiency are dynamic voltage scaling and switching off devices such as hard drives. Global optimizations are based on global decisions such as switching off complete nodes or prioritization of services. Naturally, the combination of both options brings best results. However, recent research showed that global optimizations result in a higher benefit than local ones [18]. Today distributed systems are usually designed to scale out rather than scale up.

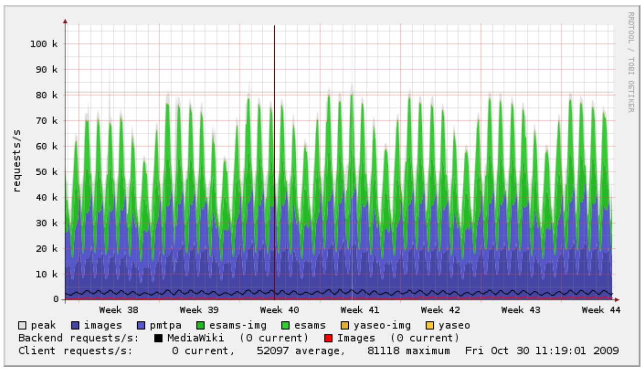

Figure 1: Requests per second at the Wikimedia clusters in October 2009 in Europe (green) and the USA (blue) (image source: `http://en.wikipedia.org/wiki/Most_viewed_article`).

Therefore, the most common hardware architecture is *shared nothing*. The number of processing nodes in such a system is an important efficiency factor [3]. Too few nodes will not be able to process the given workload; an over scaled system will however contain many nodes that are idle or only used scarcely. It is common practice to scale a system, so that it has enough resources to process peak workloads easily. Even if the peaks occur very frequently, most of the time the system is underloaded. An example of this behavior can be seen in figure 1: it shows the number of requests per second at the Wikimedia clusters, host of the Wikipedia website. The workload is variable and the average load is only about 65% of the maximum load. So scaling the system according to the active workload could reduce the number of nodes on average by 35%.

In highly dynamic environments, such as web applications, the distributed system should be able to adapt its resources automatically. In autonomic computing this form of self tuning is usually described as a control cycle, for example in the MAPE model introduced by IBM [1] or the OPR model introduced by Weikum et al. [19]. The *online feedback control loop* described by MAPE contains four phases: monitoring, analysis, planning and execution. In the monitoring phase the systems status is recorded. Periodically this status is analyzed. Based on this analysis a plan for optimizations is created, if necessary. The plan then is executed. After the execution, the system is monitored again and the loop starts over. The MAPE loop is shown in figure 2.

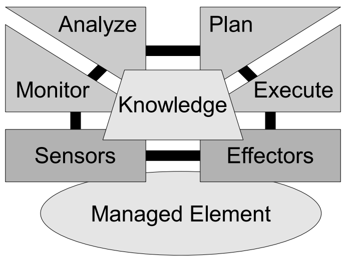

Figure 2: The MAPE online feedback controll loop.

In this paper we introduce a scaling framework that uses an online feedback control loop in order to dynamically scale distributed applications. It is highly configurable and easily extendable. We will show how dynamic scaling combined with automatic switching-on and -off of nodes increases energy efficiency.

Our contributions are the following. We introduce Scalileo, a universal scaling framework for distributed Java applications. The framework supports self-scaling and requires only the implementation of certain interfaces to be integrated into an application. Depending on the application, no change of the code base is needed. Scalileo uses a bootstrap procedure to acquire new nodes. Therefore, no installation or configuration is required on previously unused nodes. The framework is suitable for heterogeneous environments of arbitrary scales. Because of the minimal installation overhead, it is also cost-effective for small scale installations. We show how our scaling framework can be used to increase energy efficiency by 30% for a cluster of web servers. The framework can be integrated into any distributed application, e.g. resource management, database systems, web services, etc.

The rest of the paper is organized as follows. In the next section we provide an overview of the architecture of Scalileo. In section 3 we show how an efficient web server cluster can be constructed with our framework. We evaluate its efficiency in section 4. Related work is presented in section 5. We conclude in section 6 with our results and future work.

2. SCALILEO'S ARCHITECTURE

Most distributed applications feature several worker nodes that carry out computationally expensive tasks, as well as a central component which is responsible for organizing and controlling the entire system. In these systems not all of the nodes are equal, as the central component has different tasks then the worker nodes. Therefore, the Scalileo framework implements a multi-tier architecture. It uses a central component, called master node, which is responsible for organizing a set of workers running on different physical nodes. The workers are controlled by the master and are intended to carry out its commands. Further on they perform benchmarks to monitor the status of their corresponding node. An overview of the architecture of Scalileo and its relation to the distributed application can be seen in figure 3.

2.1 Workers

A worker in Scalileo is a process, running on a specific physical node where a process of the distributed application runs or may run in the future. One main task of the worker is to spawn or shut down processes of the distributed application in order to increase or decrease their capability. Therefore, the distributed application can supply a set of files and a set of commands to be executed on this machine. These will then be transferred by the Scalileo system. By doing so, the application can transfer an executable program as well as initial data needed by the distributed process that should be started. Thus, the remaining duty of the distributed process is to integrate itself in the distributed application, e.g. by registering itself at a superior instance. If the node is not needed any more the worker can shut down the spawned process by executing a command that is provided by the application. If specified, it then transfers the data back from that node.

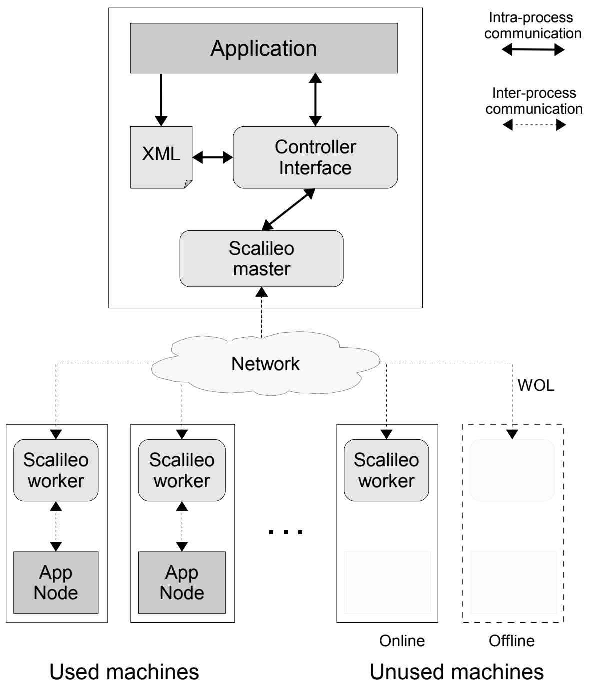

Figure 3: Overview of the Scalileo architecture.

The second task of a worker is to run benchmarks on its system. This can be measurements of system wide performance parameters like load, CPU power or free space but also special performance parameters provided by the distributed process that currently runs on this machine. These data are acquired in specified intervals and are sent to the master node for evaluation.

2.2 Master

A worker deals with the tasks that are processed on a single node. In contrast, the duties of the master node are the organization and management of the worker nodes and keeping track of the global state regarding the overall performance of the distributed application. The main tasks of the master are:

- Initializing workers:
  First, the master starts a worker on every node that may be used to run processes of the distributed application. It will log on to each node, transfer the worker's executable file to that node and start the worker process.

- Collecting benchmark results:
  The master node collects the benchmark results collected by the workers on their respective nodes. In the course of this process the workers send these results via a network connection back to the master node. The master node has a special thread listening for incoming results.

- Keeping track of the overall performance state:
  As the central instance in Scalileo, the master node is the only one aware of the benchmark data of all workers in the system. Using this information, the master will reduce the values for each benchmark type to a single value. For example the free space of every single node in a distributed storage system is accumulated to a single value describing the free space still available on all active nodes in the system. It is also possible to aggregate the reduced values of different benchmarks into a single combined value.

- Maintain performance specifications:
  The master node constantly monitors the benchmark results and compares them to a set of preset constraints that define upper and lower bounds between which these values should lie. If a certain value exceeds such a boundary for a specified time, the system needs to be scaled up or down. If this is the case the master node informs the distributed application that scaling is necessary. The master then passes a list of nodes that can be added to the system or respectively removed from it. This list is ordered according to the suitability of the nodes: the most promising nodes are at the top of the list. If a storage system for example runs out of free space the nodes with the most available space are ranked first.

- Add and remove nodes from the system:
  When the distributed application receives a scaling request from the master and decides to comply with this, it picks a node from the given list. As the easiest option, the application can take the first element on the list. It is most promising to solve the problem in correspondence to the benchmark data. However, it can also independently choose a node for scaling or even decide not to scale at all. If a node is chosen for scaling, the application provides a command to the master node either to spawn a node or shut it down. Additionally, a set of files can be defined that is be transferred to the node or back from it. Then the master node first spawns a worker process on that node (if not already running) and secondly executes the given command with the help of the worker.

As the master node has to communicate with the distributed application in order to perform the scaling, an interface is necessary. In Scalileo this is achieved by implementing the Controller interface. It contains all necessary methods to cooperate with the master node and to provide the required data. We describe the different interface methods in detail below:

- *constraintViolated*: This method informs the controller that a constraint has been violated. Such an event might not necessarily lead to a scaling, as the violation could be temporary and therefore not long enough to initiate a scale action, but it gives the application the chance to react to certain changes of the system's performance state.

- *beforeScale*: This method is called by the master node if it determined that scaling is necessary and the violated constraint is passed to the application.

- *chooseNode*: Next the ordered list of nodes is passed on to the application by the *chooseNode* method. The application returns the node that should be used for scaling or null if no scaling is desired at this time.

- *getNodeSetup*: Subsequently, the master node will ask for the setup of the chosen node through the *getNodeSetup* method. The setup contains the command to be executed, the files to be transferred to the node as well as the target directory where the files should be copied to and where the command will be executed. The executed command has to terminate after it has started the desired process and return 0 if this was successful and a value greater than 0 if a problem occurred.

- *getNodeShutdown*: In the case of a scale-down event, a shutdown command is be retrieved through this method from the controller. The application provides the command for shutting the application down. It can optionally specify a path to a file or directory on the remote node, which will be zipped and transferred back to the master.

- *afterScale*: After the scaling process the application is informed about the result. With this information the application can determine if the scaling was successful or if an error occurred. The method returns an error code if an error occurred before the command could be started. If the command could be executed the return value of this command is returned instead.

Additionally, there exist further methods for informing the controller of certain events, like the receiving of benchmark results, handling errors such as hardware failure, etc. However, since many of these methods are not necessary for every application, Scalileo offers an AbstractController class, which already implements most of these methods with default behavior, thus reducing the programming effort. In this case a programmer only needs to implement the two obligatory methods for retrieving the node setup and shutdown commands.

2.3 Parameterized Components

For every task in Scalileo a certain type of component exists that either specifies how to do this task or holds the necessary data for it. To make Scalileo extensible and to

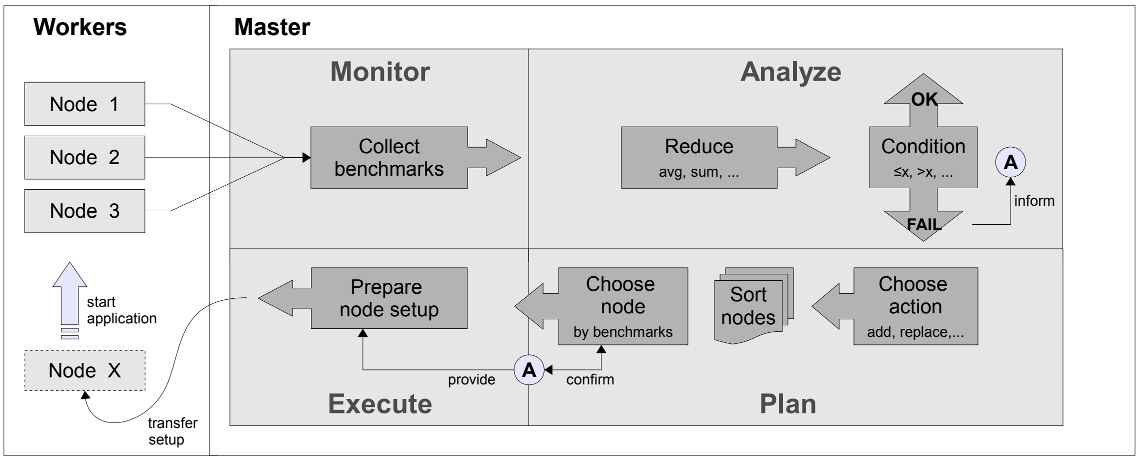

Figure 4: Scalileo feedback control loop.

achieve a high adaptability for most scenarios and applications, the components have a common structure so that the complete configuration of Scalileo can be defined within a XML file. For this Scalileo uses the Java Reflection API, an API which allows to construct objects at runtime by using metadata about the object's class. This metadata stores information such as the name of the contained methods, the name of the class, the name of parent classes, and/or what the compound statement is supposed to do. Using this information, an object can be created by reflecting upon the given class name and determine if that class exists, what kind of operations it supports, which interfaces it implements and what parent classes it has. This gives Scalileo the ability to specify its components in XML and to construct the corresponding Java object at runtime with this information. Therefore, it is not necessary to hard-code any predefined components. Instead, it is sufficient to specify only certain interfaces which define the methods that the implementation of a certain component must provide.

In the configuration file every Scalileo component is specified with a certain tag, depending on the type of this component. The tag requires two attributes: an ID for referencing this component and a fully qualified Java class name that specifies the Java class which will be implementing this component. Consider the following example that implements the login process to nodes via the Secure Shell protocol (SSH):

```
<login-method id="sshLogin"
    class="scalileo.login.SSHLogin" />
```

The chosen ID is ”sshLogin” and the implementing class will be scalileo.login.SSHLogin. To instantiate this class later via the Java Reflection API it has to be assured that these classes follow a common architecture. Thus all classes that implement a Scalileo component must be derived from a certain abstract class, depending on the component's type. Common to all these classes is that the class must have a public constructor that takes two arguments: an ID of type java.lang.String and a set of parameters for this component of type java.util.Map<String,Object>. Every parameter in this map is identified with a key and has a java.lang.Object as corresponding value. Additionally, a component must provide a method *hasValidParameters* which is called after constructing the object to determine if all necessary parameters are set. In this method the programmer must assure that the parameters set in the constructor are complete and valid. If they are not valid, false has to be returned and the component will not be instantiated.

```
public <constructor>(String id,
    Map<String,Object> parameters) {}
public abstract boolean hasValidParameters();
```

The parameters used in the constructor are specified in the XML file when a certain instance of this component is defined. Following the example above, an instance of a login component is defined in the XML file at the specification of every node. For the SSH login example two parameters for username and password must be passed to the login component. In the node definition in the XML file a login-method must be defined in the <login-with> method, so that Scalileo knows how to access the specified node. As login type the ID of the desired login-method must be chosen. Additionally the <login-with> tag can contain several <parameter> tags that define the parameters for this component. A parameter is identified with a key and must specify the type of the parameter which must be a fully qualified Java class name. This class must provide a public constructor that takes a String as argument, where the used String will be the content of the value attribute. Thus, every class that provides such a constructor can be used as a parameter object. The Java framework already provides many possibilities for this as for example all primitive types like String, Integer, Float, Boolean or complexer objects like java.util.Date. A node specification in the XML file could therefore look like this:

```
<node name="node-01"
    address="node1.example.com">
  <login-with type="sshLogin">
```

```
    <parameter key="username"
        type="java.lang.String" value="alice"
        />
    <parameter key="password"
        type="java.lang.String" value="secret"
        />
  </login-with>
  ...
</node>
```

When Scalileo parses the node definition in the XML configuration file it will first instantiate all defined parameter objects and save them in a parameter map object. Then the specification of the given login type ”sshLogin” is looked up and the corresponding class ”scalileo.login.SSHLogin” will be instantiated with the components ID and the parameter map. When Scalileo later needs to log on to that node the created login-component will be used. As all other components are defined following the same principle the procedure to construct them is analogue to the one described in the example above.

In the following, the different Scalileo components are described. Figure 4 depicts the internal control cycle of Scalileo, which reflects the MAPE model. It shows most of the components with their relationship and interaction.

2.4 Benchmarks

The components for measuring performance and other properties are called benchmarks. All benchmarks must extend the abstract class scalileo.benchmark.Benchmark which requires - in addition to requirements mentioned before - a method *run* that returns a double value as result of the benchmark.

```
public abstract double run() throws
    BenchmarkException;
```

The Scalileo Benchmark class provides functionality to repeat a benchmark in given time intervals. This can be achieved by specifying an Integer parameter called ”interval” which defines the interval length in milliseconds by which the benchmark should be repeated. In the XML configuration file an example benchmark definition could look like this:

```
<benchmarks>
  <benchmark id="ExampleBenchmark"
      class="example.package.ExampleBenchmark"
      />
  <benchmark id="OtherBenchmark"
      class="example.package.OtherBenchmark" />
</benchmarks>

<node ...>
  ...
  <use-benchmark type="ExampleBenchmark">
    <parameter key="interval"
        type="java.lang.Integer"
        value="10000"/>
  </use-benchmark>
  <use-benchmark type="OtherBenchmark">
  ...
```

```
</node>
```

When a worker process is spawned on a node by the master, the benchmarks belonging to this node are transmitted and the worker will ensure that the benchmarks are executed in the given interval. If no interval is specified, the benchmark will only be run once at the start of the worker process. This is used for benchmarks that measure static parameters like CPU frequency or other primarily hardware related parameters.

Scalileo comes with a set of predefined benchmark components like a PingBenchmark class measuring the round-trip time of a ping packet to a certain host or a FreeDiskSpaceBenchmark class measuring the available disk space on a certain file system. All predefined benchmark classes are located in the scalileo.benchmark package. It is also possible to use already available measurements, for example from monitoring systems such as Ganglia [14]. They only have to be wrapped by a implementation of the *Benchmark* interface.

2.5 Reduction

When the master node receives a benchmark result from a worker it will update all constraints affected by that benchmark. Before this can be done, the single benchmark results must be merged into a single value. To do this, Scalileo uses so called reduction components that will reduce a set of values in one aggregated value. Reduction components must be derived from the abstract class scalileo.reduction.Reduction which requires to implement one method for performing the reduction. This method takes a collection of Double values and will return the reduced value of this collection:

```
public abstract double
    reduce(Collection<Double> results);
```

Scalileo comes with a number of reduction components, covering the most common reduction functions. Among others these are components for reducing a given list of values to their maximum, minimum, sum, average or median value.

2.6 Conditions

The next components in the Scalileo framework are so called conditions, to which the reduced values are compared by the master node in order to determine if the system's performance is still in a desired state. Those conditional components must be derived from the abstract class scalileo.condition.Condition and therefore implement a *check* method that takes a reduced double value as its argument and return a boolean value which indicates if the condition is met or not.

```
public abstract boolean check(double value);
```

Three conditional components are predefined in Scalileo, one to test if a value is equal to another value (EqualCondition), if it is smaller than an upper bound (MaxCondition), or if it is greater than a lower bound (MinCondition). It is also possible to specify several conditions which are connected with a logical AND conjunction, so it is for example possible to specify an interval in which a value must be by using a MinCondition and a MaxCondition together.

2.7 Constraints

The three components mentioned earlier, i.e. benchmarks, reductions and conditions, are combined together into so called constraint components. A constraint component defines a boundary for the values of a certain benchmark. If the values are not within this boundary, the constraint is violated. Therefore, a constraint specifies one benchmark component whose values is reduced to a single value with a specified reduction component and defines one or several conditions which must be met by the reduced value. Additionally, the constraint does not just cover the last known value, but tracks the values over a specified period of time. Only when the constraint is violated over this period, a scale action is performed. This avoids that an expensive scale operation is executed due to a single short-term peak or a temporary slowdown of a node. By tracking the values over a longer time period, the collected values are averaged over this period. Only when this average exceeds the defined boundaries, a scale operation is suggested by the Scalileo framework. Like all components of Scalileo constraints are defined in the configuration XML file. An example for a constraint definition can be seen below. For now all parameters have to be preset.

```
<constraints>
  <constraint id="SomeID" historyDelay="20000">
    <use-benchmark type="ExampleBenchmark" />
    <reduce-by type="MaxReduction" />
    <check-condition type="MaxCondition">
      <parameter key="max" value="45"
            type="java.lang.Double" />
    </check-condition>
    <scale-action action="addNode">
      <choose-by benchmark="ExampleBenchmark"
            better="lower" weight="1" />
    </scale-action>
  </constraint>
  ...
</constraints>
```

Two types of scale actions (addNode and removeNode) can be declared in a constraint, causing the adding respectively the removal of a node. It is also possible to define both types in one constraint in order to achieve a replacement of a node. For each action the controller is given a list of nodes ordered by their suitability on being removed or added. This order is created by the master node by comparing the nodes with the help of so called choice methods. A choice method defines a benchmark component whose results are used in order to compare the nodes in an ascending or descending way. The order depends on whether the values of this benchmark should be treated better if they are higher or lower. Nodes can be compared by means of several choice methods, so that the ranking is not based on the values of a single benchmark, but on a set of benchmarks.

The order algorithm is implemented following the Java Comparator interface which in this case takes two nodes (n1 and n2). It returns a negative integer, zero, or a positive integer as the first node is more suitable, equally suitable or less suitable than the second node. To compare the suitability of two nodes a point system is used. The nodes will be compared for every choice method defined. At each of these comparisons each node is given a certain amount of points. The number of points is the proportion of the two nodes to each other. If the current choice-method states that higher values are better, the points for node n1 result from the benchmark value b_{n1} of n1 divided by the benchmark value b_{n2} of n2. The number of points assigned to n2 is the reciprocal value of this fraction. So if n1 has a higher benchmark value, it will get more points than n2 and thus be evaluated as more suitable in regards to this choice-method. The points for each choice method are summed up into a total number of points p_{n1} and p_{n2} for each node. The node which achieved a higher point number at the end will be ranked higher in the list. Additionally, every choice method can be weighted to increase the influence of a certain benchmark on the order of the nodes. The points of every round are multiplied with the weight w_c of the choice method before they are added to the total number of points. The calculation of points corresponds to the equations 1 and 2.

$$p_{n1} = \sum_{c \in C} \begin{cases} \frac{b_{n1}}{b_{n2}} \cdot w_c, & \text{if higher values are better} \\ \frac{b_{n2}}{b_{n1}} \cdot w_c, & \text{if lower values are better} \end{cases} \tag{1}$$

$$p_{n2} = \sum_{c \in C} \begin{cases} \frac{b_{n2}}{b_{n1}} \cdot w_c, & \text{if higher values are better} \\ \frac{b_{n1}}{b_{n2}} \cdot w_c, & \text{if lower values are better} \end{cases} \tag{2}$$

2.8 Login Methods

To enable Scalileo to access nodes running under arbitrary operation systems and environments, the login process was encapsulated in separate login components. When the master node must spawn a worker on a certain node it needs access to the node first. It further needs to have the ability to run commands on this node and transfer files to it. The login component's task is to provide this functionality to the master node. Therefore, it must be derived from the abstract Java class scalileo.login.Login. The derived class must implement one method for transferring data and running a program on the target node.

```
public abstract void runProgram(AppSetup
    setup);
```

The AppSetup object, which the *runProgram* method receives as argument, contains a command string that is executed, a path string to a target directory in which the command is executed and a file object that contains a link to the file that is transferred to the target directory before running the command. To transfer several files at once, the files must be zipped or compressed to a single file which can then be transferred via the method above. The command can then extract the file and subsequentially run the actual command.

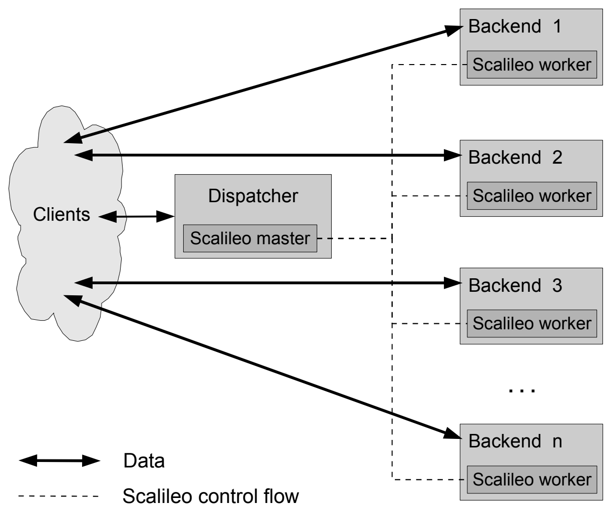

Figure 5: Clustered web server test application.

Scalileo comes with one login component implemented which is able to log in via Secure Shell (SSHv2) on the node. SSH is available for nearly every platform and is considered to be secure. Thus, Scalileo can cover a wide variety of node types out-of-the-box.

3. TEST APPLICATION

As a test case, we chose a simple distributed web server, consisting of a central dispatcher server, that distributes incoming requests to a set of up to 4 worker machines, which handle the actual request. The dispatcher uses round-robin scheduling, which is also used in DNS servers for load balancing. The redirection from the dispatcher to a worker machine is done by a HTTP 302 redirect (Found), so that the client will generate a new request to the corresponding worker. A overview of the test setup can be seen in figure 5.

The workload produced by the clients in this test was based on real-world system loads. We used workload traces of the web based E-learning management system Stud.IP at the University of Passau, which handles requests for a total of 15,000 users, consisting of different user groups like students and teachers. Like most web based systems, Stud.IP shows a significant variation over a day, a week and even a year. The load is high during working time on days in the lecture period, whereas at night time, weekends or semester break the load is comparatively low. A more detailed description of the system and the workload can be found in [16], a sample of the workload can bee seen in figure 6.

In our test we set up the clients so that the workload during the test simulates the first day of the lecture period of the Stud.IP system. We differentiate between two types of requests: dynamic and static websites. The response that was returned to a dynamic client request from the web servers was a web page containing an image, which was resized for every request on the fly by the web servers. The computational effort was therefore mainly dependent on this image resizing. For static requests we chose to return a simple HTML document. The workload was chosen in a way, so

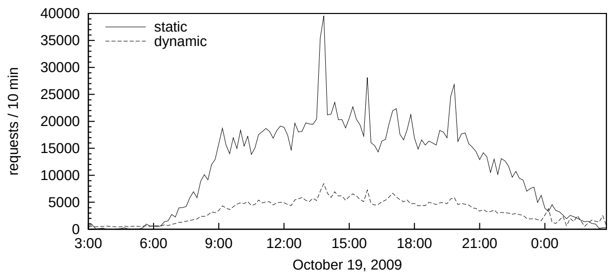

Figure 6: HTTP workload trace of Stud.IP at University of Passau for the first day of the winter term 2009.

that the system could easily handle the requests in peak times when using all four available web servers. Accordingly, in times with reduced load, it is possible to reduce the number of worker servers while being able to handle all requests. Aggregated in 10 minute steps, the original workload has a peek of about 80 requests per second and is in average 25 request per second. To reduce testing time and increase the workload, we sped the replay of the trace up by a factor of 48 and used only every 20th request, resulting in a peak load of 192 requests per second and an average load of 60 requests per second. The ratio between static and dynamic accesses varies between 0.2 and 66.4 and the median is 0.3. On average, every fourth requested webpage is dynamic. As the average load is only a third of the peak load, it is likely that reducing the number of servers in times of lower load can lead to a drastic reduction in energy consumption. It has to be pointed out, that the first day of the lecture period has the highest workload in the whole year (cf. [16]). Accordingly, for the complete year even higher energy savings will be possible.

4. TEST RESULTS

Our tests were conducted on 8 workstation PCs with Intel 4 GHz Dual Core Processors, 3GB RAM and 100 Mbit/s Fast Ethernet. The OS is Ubuntu Linux 8.04.2, Kernel 2.6.24-23 and the used Java version is 1.6.0_16. The workstations have a power consumption of 91 Watts when idle, 200 Watts during boot phase and 2 Watts when they are switched off.

We used the Scalileo framework to implement the distributed web server. The Scalileo master was running on the same machine as the dispatcher server and on every running worker machine a Scalileo worker node measured the processor load on the machine and reported back to the master as benchmark results. If the overall load on the system was beyond a threshold, Scalileo started a further web server worker on a new machine to increase the system's performance. To save energy the Scalileo system shut down unused machines and woke them up via Wake-on-LAN when they were needed again.

Since booting and halting of the machines consumes energy without any value for the distributed web server, it had to be assured that scaling did not take place on small and temporary load changes but only when necessary. The constraints for the application were iteratively determined and the final values were as follows. If the average CPU usage over all active nodes was higher than 45% in two third of all benchmark samples over 20 seconds in the simulation, a new node would be spawned. This accords to a period of 16 minutes in original speed. Respectively, at the lower bound the CPU usage had to be lower than 20% over 35 seconds, before a node was shut down. This corresponds to 28 minutes in original time. In a real world setup, both constraints could be set higher, but the high simulation speed enforced these parameters in order to give the system enough time to spawn a new node.

The main focus of the test lay on two issues: on the one hand the reduction of the necessary energy to operate the system and on the other hand the response time for a request. It is clear that both aspects have to be met, as energy saving must not happen at the cost of increased response times.

In figure 7 the power consumption of the cluster with and without the on/off policy is shown. It can be clearly seen, that shutting down unnecessary nodes effectively reduces power consumption. The total energy consumption for the test run was 175 Wh with scaling and 250 Wh without scaling, so the energy savings were 30%. In figure 8 the number of active servers compared to the workload is shown. Due to the sped up replay of the workload trace, the booting time of offline nodes resulted in the visible lag in scaling. Because of this lag the system had to boot nodes earlier than necessary under real world conditions. It also has to be pointed out that the fast replay has a negative effect on the energy efficiency, since booting time and booting power consumption had a much larger effect. In relation to the simulation speed a worker machine needed 50 minutes for booting. As mentioned before, the first day of the lecture period has the highest workload in the year. So even if the workload would be equally high every day, the savings per year were 1300 kWh.

5. RELATED WORK

Scaling Frameworks

There is little research on self-scaling applications or scaling frameworks. In the cloud computing world some frameworks for automatic application scaling are used. An example of a commercial system is Scalr[1], it uses the Amazon Elastic Compute Cloud (EC2)[2] [9]. Scalr uses different Amazon Machine Images (AMI), an Amazon proprietary virtual machine, to scale and replicate applications. These AMIs are configured to run certain applications such as web servers, load balancers or database servers. Additionally the AMIs have a monitoring suite, which is comparable to the benchmarking system of Scalileo. Unlike Scalileo, Scalr is limited to the EC2 environment and cannot be run on arbitrary clusters. Although virtualization is a viable technique for energy efficiency – an example is the Virtual Home Environment project [11] – Scalr does not aim for energy efficiency. Scalileo can also be used with virtualization. The virtualization layer can be made self-scaling, similar to Scalr, but independent of the base system. The distributed application is then started on a fixed number of virtual machines (VMs) and Scalileo allocates the VMs on real systems according to the current load.

Energy Efficiency Frameworks

Several frameworks for energy efficiency have been proposed. Petrucci et al. have presented a dynamic framework for power aware server clusters [15]. Besides an on/off-policy, they also support dynamic voltage scaling to reduce power consumption. Specialized hardware and software to measure performance and power consumption of individual machines are used. Based on the results a theoretically optimal cluster configuration is calculated using mixed integer linear programming. This involved approach makes it impossible to use self-scaling techniques as in Scalileo. Since management efficiency is an important factor besides energy efficiency [3], Scalileo has built-in performance benchmarks and makes simplified assumptions for energy consumption to reduce the setup complexity and management overhead.

A similar framework was presented by Rusu et al. [18]. It also uses dynamic voltage scaling and on/off policies, but relies – similar to Scalileo – on more simplified optimization schemes. However, it also requires extensive power consumption measurements and does not consider self-scaling. With the use of these measurements the framework is able to improve energy efficiency in heterogeneous clusters. The benefit of considering heterogeneity was also demonstrated in [10]. In Scalileo heterogeneity is only considered on a performance level. But using adapted benchmarks and conditions, previous hardware efficiency measurements could be utilized as well.

On an applicational level additional energy savings are possible. For example, Horovath et al. have shown that using prioritization for request queueing in webservers additional to dynamic voltage scaling can lead to substantial energy savings [12]. However, these kind of optimizations are outside of the scope of a scaling framework and have to be implemented in the scalable application.

At larger scales the GREEN-NET framework [7] and Muse architecture [5] show how energy efficiency of computing grids and hosting centers can be improved. This is done using on/off policies as well as energy aware service level agreements. At the scale of data centers, energy consumption of network devices such as switches can also be considered. These devices do usually not offer any low power states and constantly communicate even if systems are idle. Therefore, switching them off further reduces energy consumption [2]. These techniques are currently out of scope of the Scalileo framework, but will be considered for future work.

Similar approaches for energy efficiency have been presented in specialized applications. For example, Chen et al. present a energy aware cluster of connection servers for internet services [6]. It uses an on/off policy to adapt the number of connection servers to the number of TCP connections. Systems like this can also be implemented using the Scalileo framework.

6. CONCLUSION

[1]Scalr - `http://www.scalr.net`

[2]Amazon Elastic Compute Cloud - `http://aws.amazon.com/ec2/`

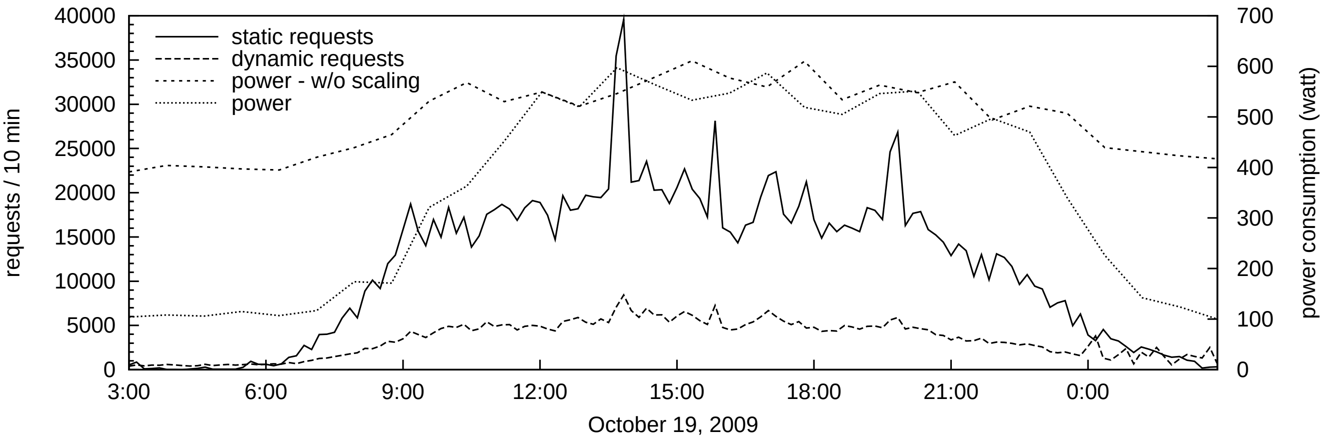

Figure 7: Energy consumption compared to workload.

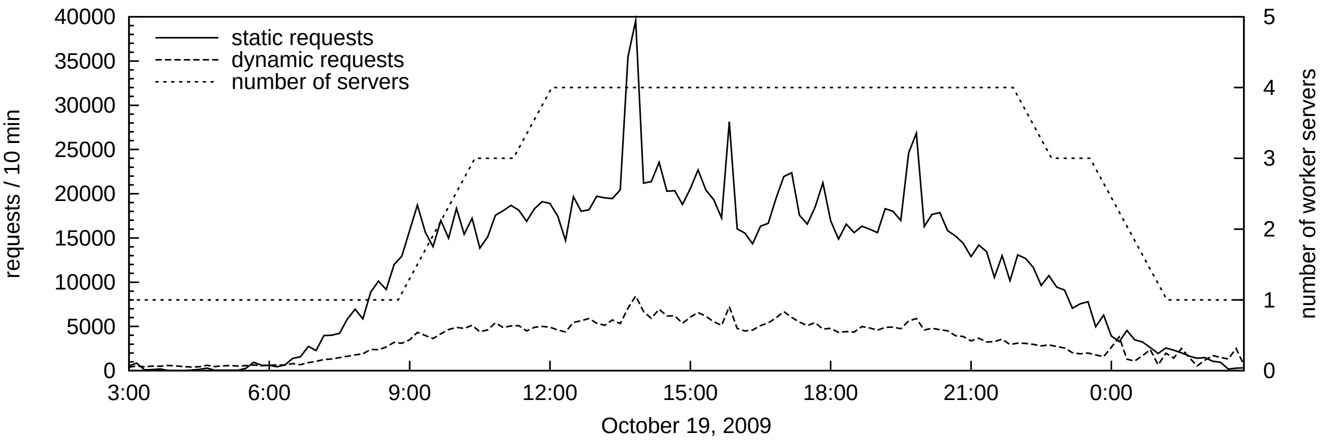

Figure 8: Number of servers used compared to workload.

This paper describes the architecture and design of the scaling framework Scalileo. The framework is highly configurable and allows dynamic scaling of Java applications. It is easy extensible and needs little to no changing of the target application. For evaluation purposes, we have implemented a simple web server cluster. By using an on/off policy, the system was able to reduce power consumption by 30% for a real world webserver workload. Since we used an annual peak workload and sped up simulation time, it is clear that much higher percentage of savings are possible.

For future work, we want to include more sophisticated benchmarks and constraints. Because of the periodicity of most web application workloads, it seems promising to introduce time series analysis. This will allow a better prediction of workload variations. Bertini et al. propose model measurement of quality of service [4], which could also improve efficiency. For now the boundaries of constraints are iteratively determined, a sensitivity analysis will further reduce the configuration overhead and improve the parameter settings. An other improvement will be the introduction of scaling hierarchies. This will make the framework better suited for systems with multiple hierarchical levels. In our example application we only considered global optimizations to reduce the energy consumption. However, our framework could easily be extended to support local optimizations like dynamic voltage scaling [21], this could also improve robustness when workloads are bursty [20]. Furthermore we will use the Scalileo framework to scale other applications such as database systems. For this we will use our allocation algorithms presented in [17]. In combination with a hierarchical model this could also be used to scale larger data intensive systems such as MapReduce clusters [8], as shown in [13].

7. ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers and especially Priya Mahadevan for their constructive comments, which helped to improved the quality of the paper.

8. REFERENCES

[1] An architectural blueprint for autonomic computing. Technical report, IBM Corporation, 2006.

[2] M. Allman, K. Christensen, B. Nordman, and V. Paxson. Enabling an energy-efficient future internet through selectively connected end systems. In *HotNets '07: Proceedings of the Sixth Workshop on Hot Topics in Networks*, 2007.

[3] E. Anderson and J. Tucek. Efficiency matters! In *HotStorage '09: Proceedings of the SOSP Workshop on Hot Topics in Storage and File Systems*, New York, NY, USA, 2009. ACM.

[4] L. Bertini, J. C. B. Leite, and D. Mossé. Statistical qos guarantee and energy-efficiency in web server clusters. In *ECRTS '07: Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 83–92, Washington, DC, USA, 2007. IEEE Computer Society.

[5] J. S. Chase, D. C. Anderson, P. N. Thakar, A. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. In *SOSP '01: Proceedings of the 18th ACM Symposium on Operating System Principles*, volume 35 of *ACM SIGOPS Operating Systems Review*, pages 103–116, New York, NY, USA, 2001. ACM.

[6] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *NSDI '08: 5th USENIX Symposium on Networked Systems Design & Implementation*, pages 337–350. USENIX Association, 2008.

[7] G. D. Costa, J.-P. Gelas, Y. Georgiou, L. Lefevre, A.-C. Orgerie, J.-M. Pierson, O. Richard, and K. Sharma. The green-net framework: Energy efficiency in large scale distributed systems. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[9] S. L. Garfinkel. An evaluation of amazon's grid computing services: Ec2, s3 and sqs. Technical Report TR-08-07, School for Engineering and Applied Sciences, Harvard University, Cambridge, MA, USA, 2007.

[10] T. Heath, B. Diniz, E. V. Carrera, W. M. Jr., and R. Bianchini. Energy conservation in heterogeneous server clusters. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 186–195, New York, NY, USA, 2005. ACM.

[11] H. Hlavacs, K. A. Hummel, R. Weidlich, A. Houyou, A. Berl, and H. de Mee. Distributed energy efficiency in future home environments. *Annals of Telecommunications*, 63(9-10):473–485, October 2008.

[12] T. Horvath, K. Skadron, and T. F. Abdelzaher. Enhancing energy efficiency in multi-tier web server clusters via prioritization. In *IPDPS '07: 21th International Parallel and Distributed Processing Symposium*, pages 1–6. IEEE Computer Society, 2007.

[13] J. Leverich and C. Kozyrakis. On the energy (in)efficiency of hadoop clusters. In *HotPower '09: Workshop on Power Aware Computing and Systems*, New York, NY, USA, 2009. ACM.

[14] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[15] V. Petrucci, O. Loques, and D. Mossé. A framework for dynamic adaptation of power-aware server clusters. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1034–1039, New York, NY, USA, 2009. ACM.

[16] T. Rabl, A. Lang, T. Hackl, B. Sick, and H. Kosch. Generating shifting workloads to benchmark adaptability in relational database systems. In R. O. Nambiar and M. Poess, editors, *TPCTC '09: First TPC Technology Conference on Performance Evaluation and Benchmarking*, volume 5895 of *Lecture Notes in Computer Science*, pages 116–131. Springer, 2009.

[17] T. Rabl, M. Pfeffer, and H. Kosch. Dynamic allocation in a self-scaling cluster database. *Concurrency and Computation: Practice and Experience*, 20(17):2025–2038, 2008.

[18] C. Rusu, A. Ferreira, C. Scordino, and A. Watson. Energy-efficient real-time heterogeneous server clusters. In *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 418–428, Washington, DC, USA, 2006. IEEE Computer Society.

[19] G. Weikum, A. Mönkeberg, C. Hasse, and P. Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *VLDB '02: Proceedings of the 28th International Conference on Very Large Data Bases*, pages 20–31. VLDB Endowment, 2002.

[20] A. Wierman, L. L. H. Andrew, and A. Tang. Power-aware speed scaling in processor sharing systems. In *NFOCOM '09: The 28th Conference on Computer Communications*, pages 2007–2015, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[21] L. Yuan and G. Qu. Analysis of energy reduction on dynamic voltage scaling-enabled systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(12):1827–1837, 2005.