TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

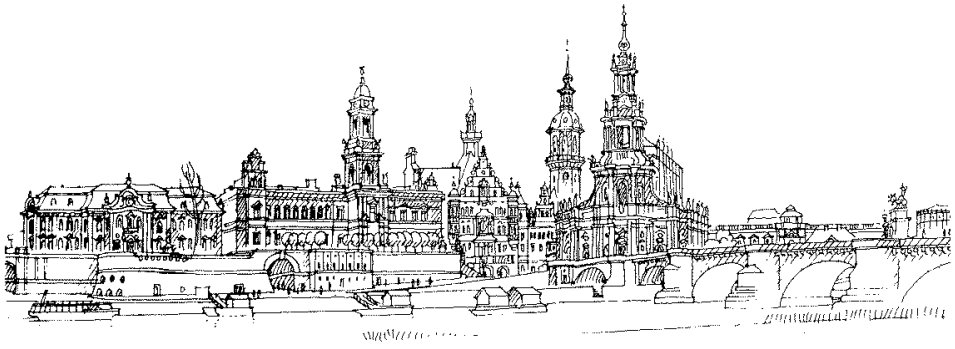
Technische Berichte
Technical Reports
ISSN 1430-211X

TUD-FI10-08-Dez. 2010

**S. Götz, C. Wilke, M. Schmidt,
S. Cech, J. Waltsgott, R. Fritzsche**

Institut für Software- und Multimediatechnik

THEATRE Resource Manager Interface Specification v. 1.0



Technische Universität Dresden
Fakultät Informatik
D–01062 Dresden
Germany
URL: http://www.inf.tu–dresden.de/

THEATRE Resource Manager Interface Specification v. 1.0

Sebastian Götz, Claas Wilke, Matthias Schmidt, Sebastian Cech, Johannes Waltsgott, Ronny Fritzsche

Technische Universität Dresden, Dresden, Germany

Abstract. Since more than 10 years energy efficiency is a well discussed research topic in computer science. Because the immediate consumers of energy are hardware resources, these resources have been subject to optimization in the first place. But hardware is in turn used by software, running on top of it. Software offers the interface to the end-user and thus allows to correlate the utility of end users with energy consumption by hardware resources. To bridge the gap between the users' needs and energy consumption on the hardware level we propose an energy-aware software architecture. In this paper we focus on a central element of this architecture: the resource manager.

1 The Key to Energy Efficiency: Bridging the Gap between Users and Energy

Our research towards an energy-aware software architecture revealed the need for an energy-aware software development process and *the energy auto tuning runtime environment (THEATRE)* [5]. Our work is part of the project CoolSoftware[1], which belongs to the CoolSilicon[2] cluster of excellence. The two main concepts in THEATRE are *energy* and *resource managers*. Figure 1 depicts the architecture of THEATRE.

The three layers of our architecture span from the user to the hardware. The users' needs are the utility which is gained in turn from energy consumption. But users interact with software, which runs on top of operating systems, virtual machines and so on. We see operating systems and virtual machines as virtual resources. Resources in general are represented by resource managers. Resource managers provide information about their resources and are able to steer them. Energy managers use this information and combine it with their knowledge on valid application component compositions to derive an energy-optimal configuration (component composition and deployment). Each software component has an own energy manager, which provides information about the component and offers functionality to deploy, redeploy or reconfigure the corresponding component. The top-most layer focuses on the users' needs, which are collected by so-called *contextors*. Besides collecting the users' expectations, they also collect

[1] http://www.cool-software.org/
[2] http://www.cool-silicon.de/

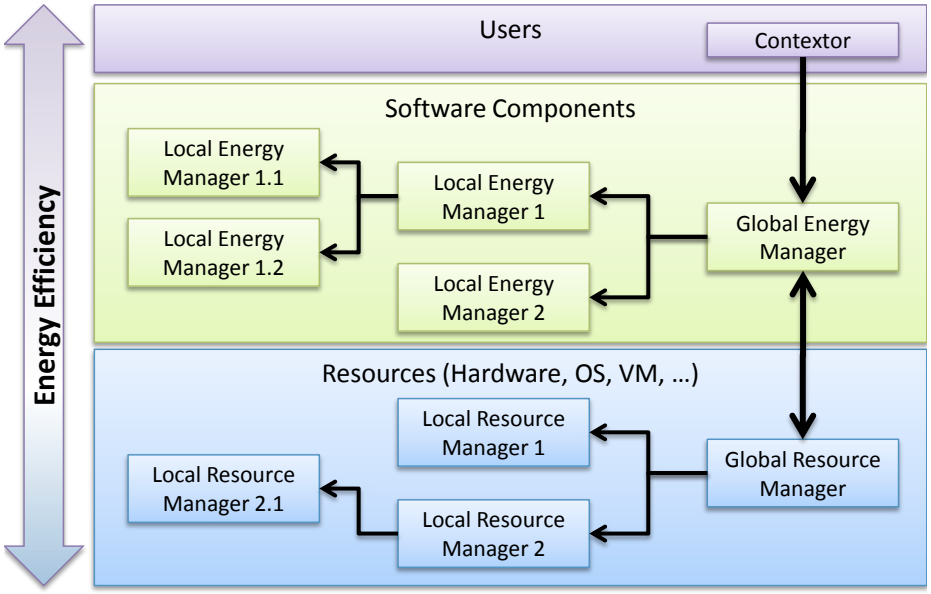


Fig. 1. The architecture of THEATRE.

information for workloads on the system. Such workloads are service requests created by users through the user interface of the software system. Energy managers know what software components require and provide in return. They know about valid system configurations, that is deployments of software components onto resources. After identifying all valid system variants, these variants are assessed in terms of their cost and utility. The variant providing the best combination of cost/utility will be the goal of the reconfiguration, which is forced by the energy manager. Reconfiguration includes the selection of another component implementation, whereas redeployment triggers the migration of a software component from one resource to another.

All resource specific information is aggregated up the hierarchy into the *global resource manager (GRM)*. The global resource manager is in turn able to disseminate requests from the *global energy manager (GEM)*. The migration of components might lead to under-utilization of resources. In such a case the global energy manager can request the global resource manager to shut down the under-utilized resource. Before the request can be processed remaining components are migrated to another resource. If all resources are fully utilized, but a further component has to be deployed, the global energy manager will request a new resource from the global resource manager. Hence, the global energy manager renders the decisions, which base on information provided by the resource managers and local energy managers.

Figure 2 depicts the *autonomic control loop* [3], which outlines how THEATRE works and elaborates on the interaction between the global energy and resource

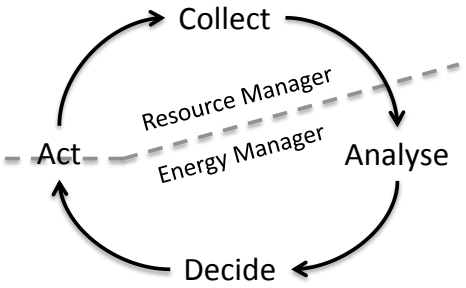


Fig. 2. Energy and Resource Managers in the Autonomic Control Loop [3].

manager. First, information about the energy behavior of resources needs to be collected. Second, this data has to be analyzed by finding mappings onto the valid system configurations, so an order in terms of energy efficiency is determined for these variants. Third, this order is used to decide for the best configuration. Forth and last, the system has to be reconfigured to the newly computed configuration.

Collecting data, as well as parts of the reconfiguration, are the responsibility of the resource managers. Analyzing the data, determining the best configuration and forcing the reconfiguration is the responsibility of the energy managers.

Determining the best configuration cannot rely solely on data collected by the resource manager. This is, because the users' utility (i.e. user metrics, perceptible qualities to the user [4], like application response time) is the key to energy efficiency:

$$efficiency = \frac{utility}{energy} = \frac{\sum_{i=0}^{n}(usermetric_i * w_i)}{energy} \tag{1}$$

$$\stackrel{e.g.}{=} \frac{response\_time * resolution}{energy} \tag{2}$$

Each user metric has a weight w_i, which defines the importance of that metric to the user. Like resource managers, required to collect energy related data, we need contextors, which collect data about the users' wishes and demands.

Furthermore, constraints provided by the application developer need to be considered. Developers need to be able to set quality constraints for their components, which cannot be deceeded. We propose to use ECL contracts [9], which specify all valid quality variations of a component.

Besides determining the optimal ratio of user utlity and energy consumption, our approach allows to compare user utilities in terms of their energy requirements. Such a comparison can be used to derive pricing models for software applications in accordance to different provided qualities.

The general architecture of THEATRE hence consists of contextors, which collect the users' demands, resource managers collecting information from and steering resources and energy managers, which analyze the collected data and derive according decisions. This paper focuses on resource managers and their interface required by the energy managers.

2 Resource Manager Interface Description

We focus on hierarchically structured resources, whose interconnections form a graph. Resources are hierarchical in the sense, that single resources are grouped by functionality for a given purpose. Classical servers are resources, but consist of many resources, too. Servers typically include several hard disks, volatile memory, a central processing unit (CPU), possibly a graphical processing unit and so on. CPUs, for example, consists in turn of multiple cores, again forming separate resources. We decide between resources, which can host component containers (`ContainerProvider`) and resources which cannot. This distinction is necessary to define deployments of software components onto resources. Moreover, we divide also resources into direct energy consuming resources and indirect energy consuming resources. Direct energy consuming resources (`PhysicalResource`s) are physical devices like CPUs or hard disks where indirect energy consuming resources (`Resource`s) are logical ones. They may group one or more physical devices (e.g. a servers) or purely logical like OS or virtual machines. Container providers are also logical resources too.

2.1 Requirements of Resource Managers

In THEATRE each resource is managed by its own resource manager. The resource managers have to fulfill the following five requirements:

/REQ1/ **infrastructure** - knowledge about the system infrastructure (which resources exists, how they are connected),
/REQ2/ **resource properties** - knowledge about device properties (from specifications and/or measurements),
/REQ3/ **energy behavior** - knowledge about energy behavior in correlation to offered services (e.g., how much energy is consumed for storing 200 MB of data),
/REQ4/ **resource control** - resource lifecycle management (power on/off, sleep-mode force),
/REQ5/ **deployment** - software components need to be deployed onto containers, which are resources, too.

The first three requirements are used by the GEM to rate the valid system configurations and thereby to determine the best ones. The forth and fifth requirement are used by the GEM to perform system reconfigurations. Figure 2 depicts the control loop of autonomic communication and the responsibilities of energy and resource managers.

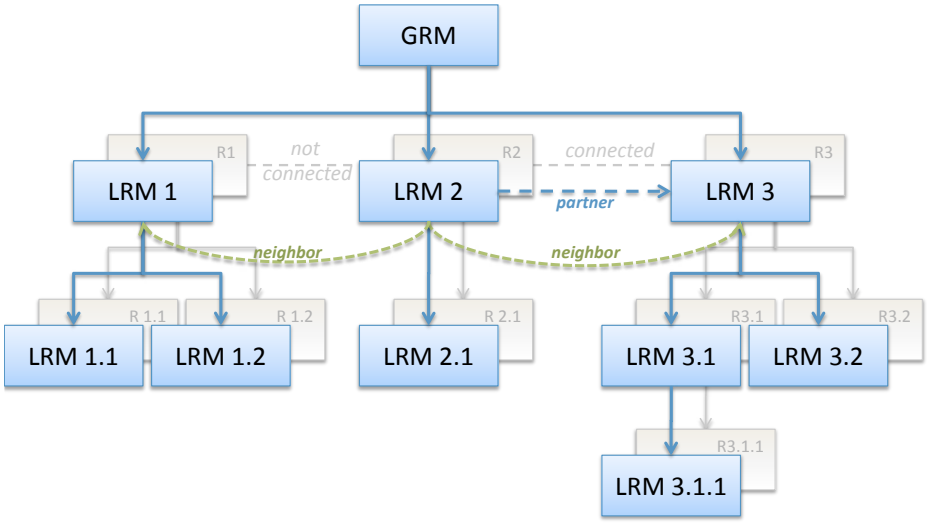


Fig. 3. Example Resource Manager Structure.

2.2 Interface Description for Local Resource Managers

Implementations of all common local resource managers in THEATRE for specific resources need to adhere to the interface specified in Listing 1. Client to this interface will be the GEM and resource managers playing the parent-role. As the Global Resource Manager and the Container Providers offer special functionally like maintaining the overall infrastructure or deployment of software components, their interface will be specified separately (see section 2.3 and 2.4).

/REQ1/ Infrastructure Resource managers provide information about the hardware infrastructure. Each resource has its own resource manager, which knows about adjacent resource managers. Not every resource manager needs to be attached to a physical resource. Instead virtual resources, grouping several physical or further virtual resources, are allowed. The top-most resource manager is the global resource manager (GRM). All other resource managers are *local resource managers (LRM)*. Figure 3 depicts an example structure of resource managers, comprising three different resources (R_1, R_2 and R_3). Each resource can have multiple sub-resources. Resource R_1 might be a central processing unit, where $R_{1.1}$ and $R_{1.2}$ are two cores. Resource R_2 can connect to R_1 and R_3. Hence the resources R_1 and R_3, as well as their resource managers, form the neighborhood of R_2. R_2 is currently connected to R_3. Therefore R_3 and its manager is a communication partner of R_1 (and its manager).

As resources are hierarchical, their resource managers have a parent and multiple child managers, which can be requested using the methods `getParentMgr()` and `getSubResourceMgrs()`. Resource managers of different resources are able to connect to each other in a neighborhood (`getNeighbours()`). Each

```
1  interface ResourceMgr {
2
3     /* /REQ1/ infrastructure */
4     ResourceMgr         getParentMgr();
5     List<ResourceMgr> getSubResourceMgrs();
6     List<ResourceMgr> getNeighbours();
7     List<ResourceMgr> getPartners();
8
9     /* /REQ2/ device properties */
10    List<String> getAllResourcePropertyNames();
11    List<State>  getResourceStates();
12    Map <String,Object>  getResourceProperties();
13    Object         getResourceProperty(String name);
14    List<Service>          getResourceServices();
15
16    /* /REQ3/ energy behavior per service */
17    double         getPredictedEnergy(Workload w);
18
19    /* /REQ4/ control */
20    State          getCurrentState();
21    StateChange    triggerState(State s, int priority);
22    StateChange    checkStateChange(State s);
23    boolean        isStateChangeComplete(StateChange sc);
24    void           setStateChangeComplete(StateChange sc);
25 }
```

Listing 1. Local Resource Manager Interface Specification

resource can have established connections to none up to all of its neighbors. The resource managers of currently connected resources can be requested using method `getPartners()`. The concept of neigbours enables the GEM and the GRM to determine similar resources for migrating software components onto them. Furthermore, the ability to express established connections between resources prevents to shut down under-utilized resources as long as they are required by others.

/REQ2/ Resource States, Properties and Services Each resource has at least two states: `On` and `Off`. But many resources provide further sleep or performance states, like defined in ACPI [6]. A resource manager needs to know about all states provided by its resource (`getResourceStates()`), because these states are used for energy use prediction (/REQ3/) and to control the resource (/REQ4/). Current hard disk drives usually provide five power states: `Off`, `Sleep`, `DeepSleep`, `Idle` and `Active`. Some drives even provide more states, like `BurstWrite` or special low-power read modes. Besides knowledge about the states, the transitions between them need to be known, too.

To access the properties of a resource, there exists severall methods. `getAllResourcePropertyNames()` returns a list containing the names of all properties of a resource. Given a property's name, `getResourceProperty(String name)` returns the value for the specific property. `getResourceProperties()` combines both methods returning a map of name-value pairs of the resource's properties.

The properties of resources are best described using the concept of quality characteristics as introduced in *CQML* [1] and *CQML+* [7]: our *Energy Contract Language (ECL). Quality characteristics* describe functional and non-functional properties of resources and software components. Such characteristics are named, have a domain and optionally a value clause. The following code snippet shows an example quality characteristic for a network connection: `dataRate`.

```
1 characteristic dataRate (connection : Network) {
2         domain numeric real bit / second;
3         value connection.getDataRate();
4 }
```

Listing 2. Exemplary ECL Contract.

Resources provide `Services` to their clients. The method `getResourceServices()` has to return all offered services of the resource, which in turn can be used to describe workloads for that resource. `Services` are named and are allowed to have an arbitrary set of `Parameters`, as describe by the interface definition given in Listing 3.

```
1  interface Service {
2     String           getName();
3     List<Parameter> getParameters();
4  }
5
6  interface Parameter {
7     String    getName();
8     String    getType();
9     Boolean   setValue(Object v);
10    Object    getValue();
11    Direction getDirection();
12 }
13
14 enum Direction {
15    IN, OUT, INOUT;
16 }
```

Listing 3. Service and Parameter Specification

`Parameters` have a `name`, `type`, `value` and a `direction`. The `direction` determines whether the `Parameter` is an input, an output or a bidirectional `Parameter`. The corresponding parameter types are described in the enumeration `Direction`.

/REQ3/ Energy Behavior with Energy State Charts The base for requirement /REQ3/ is an energy state chart based simulator, which is able to predict energy use for a specified workload (`getPredictedEnergy(Workload)`.

`Workloads` are sequences of workload items (`Occurences`), where each `Occurence` has a specified point in time (when will it occur), a specified amount of work to be done and refers to the Pin which handles the work. Resources are components (i.e. are named and explicitly describe what they require/provide). They expose the sleep/performance states they offer and the values of certain quality characteristics, like *bytes read per second*. Resource properties are allowed to change over time, but currently the decisions made by the GEM rely on the values collected at a single point time.

```
1  interface Workload extends Iterable {
2     Iterator<WorkloadItem> iterator();
3  }
4
5  interface Occurence {
6     Pin   getPin();
7     int   getTime();
8     int   getAmount();
9  }
```

Listing 4. Workload Specification

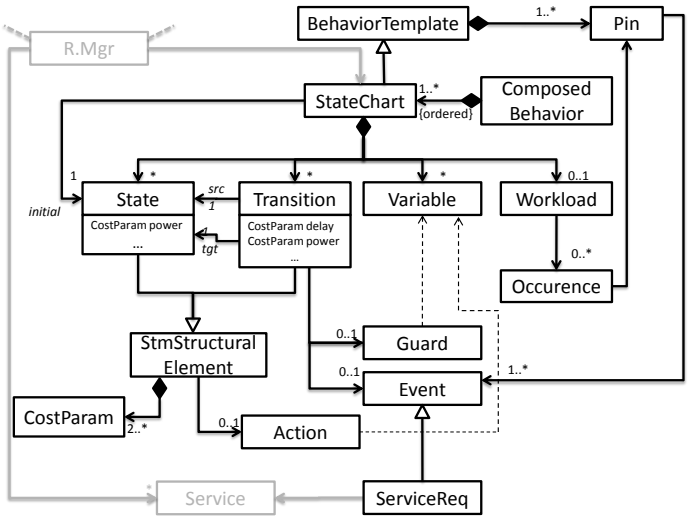


Fig. 4. Energy State Chart Meta Model.

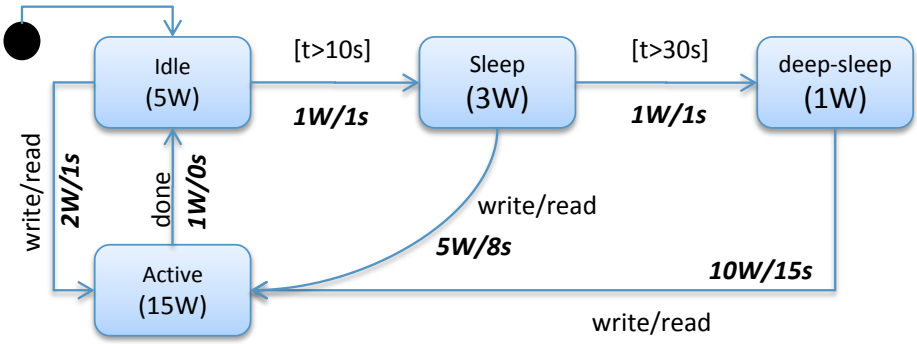


Fig. 5. Energy State Chart Example: Hard Disk Drive.

```
1  interface State {
2     String getName();
3     double getPower();
4     Action getInAction();
5     List<Transition> getFanOut();
6  }
```

Listing 5. Public State Interface

The `State` interface (c.f. Listing 5) is defined in energy state charts (introduced by Benini et al. [2]) as we use an extended version in THEATRE. Figure 4 depicts the energy state chart meta model. A StateChart is a specific BehaviorTemplate which may consists of several Pins encapsulating behavior internals. In case of energy state charts a request to a Pin triggers a transistion by reffering an Event. Which transitions is triggered depends on the current state. An other difference to usual state machines is, that states and have a defined power consumption rate and transitions are delayed and power-consuming, too. This is expressed by defining `CostParameters` for states and transtistion. In order to simulate such an energy state chart all cost parameters has to be bound to concrete values. In the current version, we do not support complex states. Mutliple state charts can be run in parallel using a `ComposedBehavior`. Our simulator is able to handle these charts fully concurrently. This is possible, because workloads define a concrete path through the charts, and these paths can be partitioned. Figure 5 shows an exemplary energy state chart for a hard disk drive with four power states: `Idle`, `Sleep`, `DeepSleep` and `Active`. Note, that a state change from `DeepSleep` to `Active` takes much more power and time than from `Idle` to `Active` (because the disk needs to spin up).

Energy state charts can be derived using experimental measurements or (rarely) from resource specifications. As mentioned above workloads are a sequence of occurences (workload items) with a specified time and an amount of work to be done as well as a reference to pin responsible to handle requests.

The occurences base on the services offered by the resource. Depending on the service parameters and the type of service the amount of work is determinable. There are three kinds of services [8]:

1. Services always behaving the same
2. Services behaving with respect to their parameters
3. Service behaving independently to their parameters

Although the first two kinds allow to automatically derive the amount for a service request, the third does not. In consequence the developers' knowledge is required. An example for the third kind of service are database requests, whose behavior depends on the contents of the database.

As our proposed software architecture explicitly models the relationship of software components and resources, our runtime environment is able to determine the amount of work for methods of the third kind. This is, because the amount of work of such methods depends on the nature and size of external resources. The direct relationship pointing to these external resources allows to derive the amount of as if these resources are parameters, i.e. like for methods of the second kind.

/REQ4/ Resource Lifecycle Management Finally, resources need to be controllable by their resource manager. The set of states provided by the resource may vary depending on the resource's nature. Nevertheless, as stated before, all resources have to support at least two states: `On` and `Off`, so that the resource manager can trigger these state changes. Changing the state of a resource implies a cost in terms of time and energy, as described by the transitions in energy state charts. The methods for /REQ4/ hence have to return a `Cost` object (Listing 6).

```
1  interface Cost {
2     Double        getReqEnergy();
3     Float         getReqDelay();
4  }
5
6  interface StateChange {
7     Cost   getCost();
8     Time   getStartTime();
9     State  getSrcState();
10    State  getTargetState();
11 }
```

Listing 6. Cost and StateChange Interface Description

The decision, whether to put a resource in a given state or not, depends on the cost for the state change. Hence, higher order managers needs to be able to get the cost for a state change without forcing the resource to perform the state change. For this purpose method `checkStateChange(State)` can be used. Forcing a state change means to hardly, that is immediately, switch to the new state. In many cases an immediate change is not possible. To softly switch to a

new state the `triggerStateChange(State s, int priority)` can be used. It returns a `StateChange` object, which comprises the time, when the change will start, besides the costs for the change (c.f. Listing 6). Additionally, the method `getCurrentState()` allows higher order managers to reflect on the current state of the resource, especially in regard to its power consumption rate.

Futhermore, a method `isStateChangeComplete(StateChange sc)` exists, which returns `true`, if the given `StateChange` has been done yet. This method might be needed to define startup and shutdown scripts. Imagine two resources R_1 and R_2 and R_1 uses R_2. To shutdown these resources correctly, R_1 needs to be shut down before R_2, because else R_1 is likely to get into an error state. Starting up both resources, requires R_2 to be started before R_1 is.

Finally, the method `setStateChangeComplete(StateChange sc)` can be used to enable the resource manager to (asynchronously) inform about a succesful excecution of a specific state change, so that `isStateChangeComplete(StateChange sc)` do not has to be called periodically.

2.3 Interface Description for Global Resource Manager

The global resource manager (Listing 7) extends the local resource managers' interface regarding functionallity to register and unregister other resource managers and, thus, control the whole infrastructure.

/REQ1/ Infrastructure The global resource manager needs to implement the `register(URI)` and `unregister(URI)` method. This methods are used by server-level resource managers to register/unregister themselves at the global resource manager. This way resources can be added to and removed from the infrastructure.

```
1 interface GlobalResourceMgr extends ResourceMgr{
2
3     /* /REQ1/ infrastructure */
4     boolean            register(URI resource);
5     boolean            unregister(URI resource);
6 }
```

Listing 7. Global Resource Manager Interface Specification

2.4 Interface Description for Container Provider Managers

The interface of the container provider managers (Listing 8) extends the local resource managers' interface regarding functionallity to deploy and undeploy software components.

/REQ5/ Deployment of Software Components Component containers (`ContainerProviders`) are special resources since they provide functionality to deploy and undeploy software components. In our current architecture, each

server hosts a single container. In consequence, resource managers responsible for servers provide an implementation for the methods grouped under /REQ5/.

These managers are of high importance to our overall approach, because the reconfiguration plans of the global energy manager are comprised of migration and redeployment commands. Managers, which are able to (un-)deploy software components need to implement `isContainer()` returning `true`.

The functionality of a such a manager divides into two parts: act and check. Software components can be deployed or undeployed using method `deploy(..)` and `undeploy(..)`. To determine, whether deploying or undeploying a certain component variant is possible, the methods `checkDeploy(..)` and `checkUndeploy(..)` can be used. The migration of a software component is not subjected to the local resource managers, as each manager knows only about its own server/container. The global resource manager, which knows all server-level resource managers, implements `migrate(..)` and `checkMigrate(..)`.

```
1  interface ContainerProviderMgr extends ResourceMgr {
2
3      /* /REQ5/ deployment */
4      boolean      isContainer();
5      boolean      deploy(String id,
6                          URI softwareComponentImpl,
7                          URI container);
8      boolean      undeploy(String id, URI container);
9      boolean      migrate(String id,
10                          URI srcContainer,
11                          URI targetContainer);
12     boolean      checkDeploy(String id,
13                             URI softwareComponentImpl,
14                             URI container);
15     boolean      checkUndeploy(String id, URI container);
16     boolean      checkMigrate(String id,
17                              URI srcContainer,
18                              URI targetContainer);
19 }
```

Listing 8. Container Provider Manager Interface Specification

3 The Global Energy Manager in a Nutshell

Client to the Global Resource Manager (GRM) is the Global Energy Manager (GEM), which acts in five phases:

/P1/ **identification** - derive all valid system variants,
/P2/ **assessment** - assess these variants,
/P3/ **selection** - determine the best variant,
/P4/ **planning** - construct a reconfiguration plan,
/P5/ **acting** - communicate the plan to the GRM.

3.1 /P1/ identification

The derivation of all valid system variants requires the knowledge of the GRM about all existing containers and what they offer. A system variant is valid, if the deployment of concrete software component implementations onto containers for a given workload adheres to the resource requirements stated in the ECL contracts. Imagine a software component `Sort`, which has two different implementations: `QuickSort` and `BubbleSort`. Further imagine there are two servers. Server 1 has 512 MB RAM, whereas Server 2 has 2 GB RAM. Furthermore, a user wants to sort a list, which has a size of 1 GB. The system variant identification should return two valid system variants: (`QuickSort`, Server 2) and (`BubbleSort`, Server 2). Theoretically two further system variants exist. Namely those of the two implementations onto Server 1. But these are not valid, because Server 1 cannot handle a list of 1 GB size with the two implementations of `Sort`.

3.2 /P2/ assessment

To assess the variants the GRM needs to be able to predict the energy consumption for a specified workload. The assessment phase also uses information provided by contextors from the user layer. The assessment of the valid system variants leads to two additional values for the variant: the cost and utility. Whereas the utility is determined using information provided by the contextors, the cost is to be returned by the global resource manager. The energy use is evaluated according to the system variant *and* the workload. Remind the example from above. The two variants will be extended to (`QuickSort`, Server 2, 400 J, 5) and (`BubbleSort`, Server 2, 300 J, 3). The users utility is (in the end) a priority starting with `1` as the lowest priority. Thus, in the example, quick sort on server two has a higher priority than bubble sort. (remind it's just an example)

3.3 /P3/ selection

The determination of the best variant is independent of the GRM and contextors, because it relies on the assessment information from the former phase. Heuristics are a simple approach to determine the best system variant, but linear approximination, neural nets and further approaches are possible, too. An easy way to determine the best variant in the example from above is to divide the cost by the utility. The higher the utility, the lower the ratio of energy costs and utility will be. In the example quick sort on Server 2 evaluates to 80 J and bubble sort on Server 2 to 100 J. Hence quick sort on Server 2 is choosen.

3.4 /P4/ planning

The construction of the reconfiguration plan does not directly use the GRM, too. Instead it uses the information from the derived variants (first phase), which includes the information, onto which containers each software component shall be deployed. Imagine, that in the example from above, an instance of quick sort

is running on Server 2. A third server is added, which has more than 1 GB RAM and this can be used to handle the example problem. Due to more modern hardware the system variant (quick sort, Server 3) is assessed to only 250 J and the same utility. The resulting 50 J (250 J divided by 5) lead to the selection of this variant. The current variant is (quick sort, Server 2). A reconfiguration plan for this scenario would simple state: migrate(quick sort instance 1, Server 2, Server 3). More complex scenarios lead to scripts of migrate, deploy and undeploy statements.

3.5 /P5/ acting

The last phase requires the GRM to process the migrations and deployments. Additionally, underutilized resources shall be forced to sleep and over utilized resources shall lead to the activation of new servers, virtual machines or containers.

4 Conclusion

The direct consumers of energy are hardware resources. Nevertheless, these resources are utilized by many abstraction layers on top of it: Virtual Machines, Operating Systems, Component Containers and Software Components. The user interacts with software applications, composed from software components running in component containers, and thereby indirectly causes energy consumption. Our goal is to bridge the gap between the users' needs (utility) and energy consumption taking place on the hardware level to build an energy auto tuning runtime environment (THEATRE) which is self-optimizing in terms of energy efficiency. THEATRE bases on an energy-aware software architecture, whose basic building blocks are components, which are connected to each other and to resources by contracts.

This paper focused on a central part of THEATRE: the resource manager. We stated the requirements for resource managers and derived the functionality, which has to be provided by them. In essence five requirements exist. First, resource managers need to provide knowledge about the resource infrastructure. Second, resource managers need to provide knowledge about their resources. Third, resource managers need to be able to predict future energy consumption of their resource in accordance to a service provided by the resource or a specified workload. Fourth, resource managers need to be able to steer their resources. That is, they need to be able to force their resources into a specified state (at least On and Off). Finally, resource manager must be able to allow deployment and migration of software components to enable reconfiguration at runtime.

This document is meant to serve the purpose of decoupling the realization of resource managers for specific hardware resources from the development of THEATRE and energy oriented software development (EOSD). The current version of this document does not present a final specification, but rather a specification, which is subject to forthcoming changes.

5 Acknowledgments

This work emerged from the research project CoolSoftware, part of the CoolSilicon cluster and has been funded by the Bundesministerium für Bildung und Forschung. We would like to thank our advisers Prof. Aßmann, Prof. Meißner and the Silicon Saxony e.V.

References

1. J. O. Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University of Oslo, Norway, 2001.
2. Luca Benini, Robin Hodgson, and Polly Siegel. System-level power estimation and optimization. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED-98)*, pages 173–178, New York, August 10–12 1998. ACM Press.
3. Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A survey of autonomic communications. *ACM Trans. Auton. Adapt. Syst.*, 1(2):223–259, 2006.
4. Jason Flinn. *Extending Mobile Computer Battery Life through Energy-Aware Adaptation*. PhD thesis, School of Computer Science, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, December 2001.
5. Sebastian Götz, Claas Wilke, Matthias Schmidt, Sebastian Cech, and Uwe Aßmann. Towards energy auto tuning. In *Proceedings of First Annual International Conference on Green Information Technology (GREEN IT)*, 2010.
6. Hewlett-Packard, Intel, Microsoft, Phoenix Technologies, and Toshiba. Advanced configuration and power interface specification, revision 4.0a. http://www.acpi.info/spec.htm, April 2010.
7. Simone Röttger and Steffen Zschaler. Cqml+: Enhancements to cqml. In *In Proceedings of the 1st International Workshop on Quality of Service in Component-Based Software Engineering*, pages 43–56. Cpadus-ditions, 2003.
8. Chiyoung Seo, Sam Malek, and Nenad Medvidovic. An energy consumption framework for distributed java-based systems. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, Atlanta, Georgia, USA*, New York, NY, USA, 2007. ACM Press.
9. Claas Wilke, Sebastian Cech, Sebastian Götz, Matthias Schmidt, Johannes Waltsgott, and Ronny Fritzsche. Energy contract language (ecl) specification v0.1. Technical report, Technische Universität Dresden, Dresden. Germany, 2010.