International Journal of Multimedia and Ubiquitous Engineering
Vol. 5, No. 1, January, 2010

An Efficient Bundle Replacement Algorithm for OSGi Platform

Ibrahim Kamel

Dept. Electrical & Computer Engineering
University Of Sharjah
kamel@sharjah.ac.ae

Abstract

OSGi is a middleware standard for home gateways, designed for smart home applications. OSGi models services as separate components, called bundles. Smart home applications might differ in their importance. For example, home security system is more important than Internet game. Bundles collaborate to provide the required service. This paper proposes a bundle replacement algorithm that takes into account the priority of the bundle and the interdependence between different bundles. Thus, given a home gateway that hosts several applications with different priorities and arbitrary dependencies among them. When the home gateway runs out of memory, which bundles will be stopped or kicked out of memory to start a new service. Because of the bundle dependencies, traditional memory management algorithms might not be efficient. Efficient replacement algorithm should stop the least important and a small number of bundles. The proposed algorithm takes into consideration the priority of the bundle or application and dependencies between different bundles, in addition to the amount of memory occupied by each service. We implement the proposed algorithms and performed many experiments to evaluate its performance and execution time. We used best fit and worst fit as yardstick to show the effectiveness of the proposed algorithms. The proposed algorithms are implemented as a part of the OSGi framework (Open Service Gateway initiative).

1. Introduction

Recently there have been a lot of interests to provide new applications for smart homes. Thanks to technologies like Fiber to Home that allowed the Internet to be used not only for connecting computers, laptops, and PDAs but also for home appliances like TV, refrigerators, and washers [20]. Remote diagnosis and remote configuration of home appliances are some of the most attractive applications. Power companies are also keeping an eye on home networking because it will allow them to provide value-added services such as energy management, telemetric (remote measurement), and better power balance that reduces the likelihood of blackout. Consumer electronics companies started to design Internet-enabled products. LG presented a smart Internet refrigerator, which has full Internet capabilities. Matsushita Electric showed during a recent Consumer Electronic Exhibition showed an Internet-enabled microwave, which can download cooking recipes and heating instructions from the Internet.

Multiple home network protocols like UPnP [11], Jini [9] [10] are expected to coexist in the home and inter-operate through the home gateway. The gateway acts also as a single point of connection between the home and outside world. OSGi [14] [15] (Open service Gateway initiative) is a consortium of companies that are working to define common specifications for the home gateway. According to OSGi model, the gateway can host services to control and operate home appliances. In the OSGi model, services are implemented in software bundles (or modules) that can be downloaded from the Internet and executed in the gateway [6]. For example, HTTP service is implemented as a bundle while security application would be implemented as another bundle. Bundles communicate and collaborate with each other through OSGi middleware and thus, bundles depend on each other. For example, a home security bundle uses an HTTP bundle to provide external connectivity [5].

Because of the need to keep the price of the gateway low, the gateway will be limited in computational resources, especially main memory and CPU. Home gateway main memory will be used by various service bundles and home applications. This paper proposes efficient replacement algorithms for managing bundles or services in home gateways. Memory management has been studied extensively in operating system field [13]. Memory management for software bundles executed in home gateways differs from traditional memory management techniques in the following aspects:

- Traditional memory management techniques, in general, assume that memory pages are independent while bundles may depend on each other.
- Many of the commercial gateways do not come with disks, which makes the cost of stopping applications or services relatively high; restarting a service might require downloading the service bundle from the Internet.

Terminating bundles [1]might result in aborting one or more other bundles if they depend on each other. Some home applications are real-time, thus, kicking a bundle from the memory may result in aborting the application or the service, while in traditional memory management model, kicking a page from the memory costs one disk I/O. However, in some applications it is possible to kick one service in the application and keep the application running. For example, Audio-on-demand might still work without the equalizer service. However, if the application considers the terminated service critical to its operation, it might terminate all other services in the tree as well. In this paper, although the proposed model and models works for the two cases mentioned above we assume that terminating a node or a sub-tree would terminate the whole application. Thus the main contributions of the paper are:

- Identifying difference between memory management in home gateway and traditional memory management problem in general computing environment.
- Introducing a novel replacement algorithm for managing bundles (or services) with different priorities. The proposed algorithm takes into consideration the priority of the application, the dependencies between applications, and the memory requirements for each application.

The rest of the paper is organized as follow; section 2 presents a formal definition of the problem and the dependency model. In section 3 we describe the proposed replacement algorithms. Experimental results are presented in section 4. Section 5 describes prior works. Finally, conclusions and future works are outlined in section 6.

---

[1] In this paper, the terms application, service, and bundle are used interchangeably.

2. Problem description

The gateway might need to free memory space to accommodate new services that are triggered by connecting a new device to the network or upon explicit local or remote requests. Although the amount of memory required to execute a service might change with time, the application service provider (or the author who provides the bundle) can give approximate statistical estimates of the amount of memory required to execute the services such as average, median, or maximum. Moreover, extra memory space might be requested by any one of the service instances (inside the residential gateway) to continue its service. If such memory is not available, the gateway picks a victim service instance (or instances) to terminate to allow the new application to start. Given that many of the smart home applications are real-time in nature, thus, the gateway tends to terminate the victim service rather than suspending it.

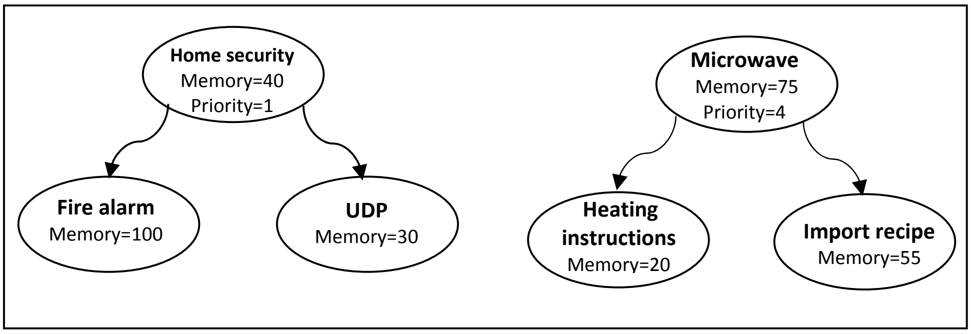

Figure 1. A gateway that hosts two applications:
home security and smart microwave

The following is a typical example that explains the problem in hand. Suppose that there are two applications that are already running in the gateway namely, home security and microwave applications. One application is the home security which uses fire alarm and UDP as a dependent services; it has a priority level 1 (highest priority). The second application is the microwave service, which has a priority level 4 and it uses two subservices: heating instructions and import recipe. The details of the memory requirement for each application and service are shown in Figure 1. Now we would like to start the refrigerator application, which requires a total of 90 memory units. The priority level of the refrigerator application is 3, which means it is more important than the microwave application but it is less important than the home security. The fire alarm service (which is a part of the home security application) has the required memory but it will not be kicked out, because it has the highest priority. Instead it can replace the Microwave application because it has the least priority level. Notice that the required space can be fulfilled by terminating several services. The challenge is to select those services to kick out from the memory gateway such that the services will be with least priority and the number of applications/services affected is minimal.

2.1. Application dependency model

OSGi is a framework and specifications for services that can be deployed and managed over wired home network [4] [5] and wireless networks [4]. It provides the standardized primitives that allow applications to be constructed from small, reusable and collaborative components. The core component of the OSGi specifications is the OSGi framework that provides a standardized environment to applications (called bundles), and is divided into four layers: Execution Environment, Modules, Life Cycle management, and Service Registry. The Execution Environment is the specification of the Java environment. The Module layer defines the class loading policies and adds private classes for a module as well as controlled linking between modules. The Life Cycle layer adds bundles that can be dynamically installed, started, stopped, updated, and uninstalled. Bundles rely on the module layer for class loading but add an API to manage the modules in run time. The life cycle layer introduces dynamics that are normally not part of an application. The Service Registry provides a cooperation model for bundles that takes the dynamics into account. Moreover, the Service Registry layer provides a comprehensive model to share objects between bundles. A number of events are defined to handle the coming and going of services. Services are just Java objects that can represent anything. Many services are server-like objects, like an HTTP server, while other services represent an object in the real world, for example a Bluetooth phone that is nearby.

The OSGi framework is completely based on Java technology. In fact, the specification itself is just a collection of standardized Java APIs plus manifest data. Bundles or services are implemented as plug-ins modules called bundles. These bundles can be downloaded from the application service providers through the Internet. Examples for services that are used for application development are Java development tools, J2EE monitor, crypto services, bundles that provide access to various relational database management systems (e.g., DB2, Oracle, etc.), HTML creation, SQL, Apache, Internet browser, XML plug-ins, communication with Windows CE, etc. Other system administration bundles like core boot, web application engine, event handling, OSGi monitor, file system services, etc. Bundles for various Internet and network protocols, like, HTTP service, Web services, SMS, TCP/IP, Bluetooth, X10, Jini, UPnP, , etc. There are many bundles that are already implemented by OSGi partners [15].

2.2. Formal definition of the problem

More formally, our problem can be described as follows. Let $G=\{g_1,g_2,...,g_j\}$ present the set of graphs (applications), and let $S=\{s_1,s_2,...,s_i\}$ be the set of service instances currently resident in each graph in the main memory. Service instance s_i occupies $M(s_i)$ memory, and each s_i may have other services depending on it. $T(s_i)$ is the set of services that depend on s_i, and the memory occupied by s_i and its dependants is denoted as $M(T(s_i))$. The services in the memory gateway have three levels of priorities High, Medium and Low (*H*, *M* and *L*).

Given that a new service instance s_i, with memory requirement $M(s_i)$ has to be created, it might be required to remove some of the currently existing instances in order to free room for the new instance. Assume that the extra required memory for this operation is M_t units, that is $M_t=M(s) - M_f$, where M_f is the current amount of available memory. Here we assume that, when a service instance is terminated, all instances depending on it will be terminated and removed as well. Our goal is to reduce the quality of removed (stopped) services. More precisely, it is desired to find a service with least priority, whose ejection, together with all its dependents, will make available a total memory of at least M_t units.

In this paper we discuss two approaches to achieve our goal in preserving the quality of services in the memory gateway and present two algorithms The *Relative Weights* (*RW*), and the *Strict Priority* (*SP*) algorithms.

2.3. Traditional replacement algorithms

The traditional memory management techniques, like *Best Fit* and *Worst Fit* make selection based on the amount of memory used and ignore the dependencies. In fact these algorithms can be used to solve the problem in hand. We use *Best Fit* and *Worst Fit* algorithms as yardstick to evaluate the performance improvement achieved by the proposed algorithms. We modified *Best Fit* and *Worst Fit* to take into consideration the total accumulative memory of each service (bundle) resulting from stopping one or more service(s). ***Best Fit*** chooses the service, $s \in S$, with the smallest total memory that is $\geq M_t$. While ***Worst Fit*** chooses the service, $s \in S$, with the largest total memory that is $\geq M_t$.

3. The new bundle replacement algorithms

The algorithm mainly visits all the nodes in sequential manner. Note that the node *X* can be a root of a tree (an application), a leaf node, or a non-leaf that acts as a root of a sub-tree. Recall, leaf and non-leaf nodes represent services that belong to that application. If *X* is the root node then the gateway will stop the corresponding application. But if *X* is non-leaf node, then deleting *X* delete the sub-tree under *X*. This will result in stopping some features of the application. In many cases applications can continue to run at reduced functionality. For example, stopping the “Equalizer” service in an Audio-on-Demand application would not stop the audio delivery and the Audio-on-Demand service can still continue working without the “Equalizer” service. In our experiments, without loss of generality, we assume that stopping a service will stop all dependent services in its sub-tree but will not stop the hosting application. We implemented two flavors of the service management algorithm depending on how the priority is handled.

3.1. The relative priority replacement algorithm

RW Algorithm

```
for each g_j in set G       // graphs loop
   for every s_i in graph g_j     //services loop
          if (M(T(s_i))>M_t )
               //s has enough  memory
               if(W(T(victim))  > W(T(s_i))  )
                    victim= s_i;      // total  weights for s_i < victim
               end if
          end if
   end for // services loop
 end for // graphs loop
if (victim!=NULL)
   delete(victim);        // delete  victim service
else
   return  "no solution found"
end if
```

Figure 2. RW replacement algorithm

Some of the real life scenarios represent priority by weight values that reflect the importance of the application. Relative Weight (*RW*) algorithm treats priorities as weights. Large weight values are assigned to high priority services and small weight values are assigned to low priority services. In this algorithm, $W(s_i)$ is assigned to each root node to the priority level that the corresponding application. Subservices, which are represented by leaf and non-leaf nodes, inherit the priority from their parents.

$W(T(s_i))$ is the total weight for the service with its dependants. $W(T(s_i))$ is calculated by adding up the weights of the node s_i and all the nodes in its sub-tree. The terminated service (victim) will be the one with the least weight and of course its termination frees enough space for the new coming application.

The algorithm in

Figure **2** checks if the service has the required memory for the new coming service, then we check for the service with least weight. So the *RW* algorithm traverses all the services available in the gateway and checks if the service has the required memory. If the service does have the required memory the algorithm checks if its weight is less than that of the victim; if this is true, the victim is updated. Note that the *RW* model may not find a service with enough memory space; in this case, the new service cannot start.

3.2. The strict priority replacement algorithm

The other way to treat applications with different priority is to give an unprecedented attention to high priority applications before serving applications with lower priorities. We refer to this algorithm as the *Strict Priority* algorithm. The difference between the strict treatment and the relative weight treatment of the priority appears when there is a need to delete more than one low priority service, say *c* low priority services. If the total weight of the *c* low priority services is larger than the weight of a high priority service, then the *Relative Weight* algorithm will remove the high priority service. While the *Strict Priority* algorithm will remove the *c* low priority services regardless of the value of *c.*

Strict Priority model assumes that the priority is a property of the application; all services and subservices inherit their priorities from their parent applications. The model assumes that there are *k* different priority levels assigned values from 1 to *k,* where 1 refers to the highest priority and *k* refers to the lowest priority.

To minimize the number of services terminated, we select to terminate the node with minimum number of dependents. To account for the number of dependent services (that will be terminated by kicking the sub-tree root) we use the $Ratio(s_i)$ formula:

$$Ratio(s_i) = \frac{[illegible]}{[illegible]}$$

The terminated service (victim) will be the one with least priority and has low *Ratio value.* The *SP* algorithm performs one pass through the services in the memory gateway. Since the new service cannot kick out a service of higher priority, the *SP* algorithm simply considers services of equal or less priority than the new services. So the *SP* algorithm traverses all services in the gateway to select the candidate victim. The algorithm will check if the priority of s_i is less than the priority candidate victim. If true, s_i is added to the candidate victim list. Among all candidate victims with the same priority, the algorithm chooses the one with the least *Ratio*. This process is repeated until all services are processed.

SP Algorithm

```
for each Gi in set I   //graphs loop
      for every s in graph Gi   //services loop
          if (M(T(s))>Mt )
             //s  has enough memory
             if (P(victim) < P(s) )
                 victim = s ;    // priority of s  < priority of the victim
             else if (P(victim)==P(s) )
                    If (Ratio(victim) < Ratio(s))
                       victim=s;     //equal priority and different AMS
                    end if
             else
                  // don't update victim
             end if
          end if
      end for // services loop
   end for // graphs loop
  if (victim!=NULL)
     delete(victim);        //this function  deletes the victim service
  else
   return "no solution found"
   end if
```

Figure 3. Pseudo code for the strict priority algorithm

4. Performance evaluation

We carried extensive simulation experiments to evaluate the performance of the proposed algorithms in terms of the number and priority of the removed services. We also measured the algorithm execution time. The amount of memory required by each bundle (or services) is assumed to be uniformly distributed. Initially, services are generated with random sizes and loaded into gateway memory, until the memory becomes almost full; in our experiments we filled the gateway with 100 services. Each service can be dependent on a number of other services. Dependent service sizes are selected according to uniform distribution from a pool of available services. The sizes of memory required for the execution of a bundle (or service) are in the range from 1MB to 5MB. Services have three levels of priorities High, Medium and Low (*H*, *M* and *L*).

The expected output of the simulation is to find out which service(s) should be kicked out to make room for a incoming service. To measure the quality of the deleted services we calculate the total weight of the stopped services using the equation below. *V* is the set of stopped services. W_v denotes the total weight of all services that are stopped to start the new service.

$$W_v = \sum W(s)$$

We conducted experiments to compare the performance of the traditional algorithms, namely, Best-fit and Worst-fit with the proposed algorithms *RW* and *SP*. Each experiment is repeated 100 times and the average of the results is calculated.

4.1. Evaluation of the relative weight algorithm

In this experiment we compared the *RW* algorithm with the well-known best fit and worst fit algorithms in terms of the quality of victim services, as the size of the new coming bundle increases from 1 MB to 10MB. To measure the quality of the deleted services we assign weights {400, 200, 1} according to the priority these services obtain (High, Medium and Low) respectively. The performance of the service management algorithms is evaluated by measuring the total weight of the stopped services as a function of the size of the new coming service. Figure 4 shows the total weight of the stopped services in the Y-axis and the size of the new services in the X-axis. The total weight of the stopped services is increasing as the size of the new coming service increases because of the need to terminate more services. The results show that the *RW* outperforms the traditional algorithms in preserving the services with high priority. The performance gain increases with increasing the size of the new service.

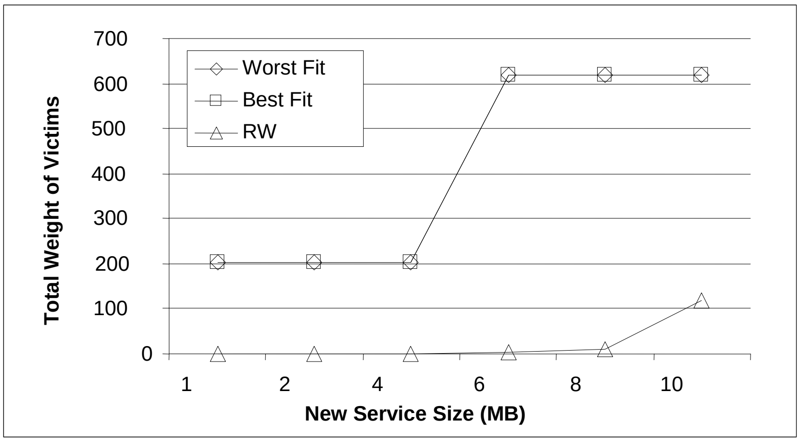

Figure 4. Quality performance of algorithms while increasing the new service size

Table 1 compares the execution time of the *RW* algorithm with the execution time of the best fit and worst fit algorithms as a function of the number of services that exists in the gateway. The size of the new coming service is fixed to 5 MB. The costs of the three algorithms increase with increasing the number of services in the gateway because of the sequential nature of the algorithms. The results show that the cost of the *RW* algorithm is higher than (but close to) the best fit and worst fit algorithm. The difference in the execution time is always less than 6% and it significantly decreases as the number of services in the gateway increases. This makes the proposed algorithms suitable for practical applications.

Table 1. Comparing the execution time of the RW with best and worst fit

No. of existing services	Worst Fit(μs)	Best Fit(μs)	*RW*(μs)
100	18	18	19
200	35	35	36
300	51	51	53
400	68	68	69
500	85	85	86

4.2. Evaluation of the strict priority algorithm

To check the performance of the *SP* algorithm, we performed experiments that count the number of services with high, medium and low priority levels that are deleted. The *SP* algorithm uses the *Strict Priority* model, which does not use weights that relate between various priority levels. **Figure 5** illustrates the accumulated number of deleted services for each priority level.

The size of new coming service is set to 15MB. Each experiment is repeated 100 times. The y-axis shows the accumulated number of terminated services over the 100 experiments for each algorithm used. One can easily observe that the *SP* algorithm outperforms *Best Fit* and *Worst Fit*. The *SP* algorithm protects services with high priority from being kicked out. In addition, the *SP* algorithm terminates less total number of services (regardless of the priority level) when compared with the traditional algorithms, *Best Fit* and *Worst Fit*. **Figure 5** shows the total number of deleted services for *Best Fit* (or *Worst Fit*) is 606 while the total number of deleted services for the *SP* algorithm is only 303, which accounts for 50% improvement.

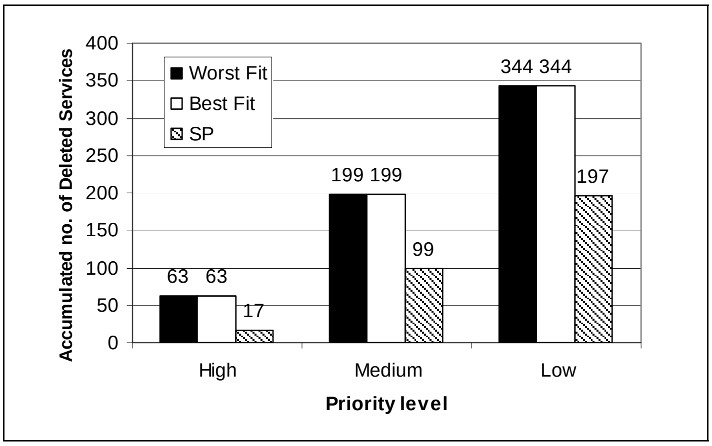

Figure 5. Average Number of Deleted Services, *Single Pass* algorithm using *Strict Priority* model

Table 2. Execution time comparison between the SP and the Best Fit and Worst Fit

No. of existing services	Worst Fit(μs)	Best Fit(μs)	*SP*(μs)
100	18	18	24
200	35	35	46
300	51	51	69
400	68	68	95
500	85	85	116

Table 2 shows the cost of the *SP* algorithm in terms of execution time and compare it with the execution time of the best fit and worst fit algorithms as the number of services in the gateway changes. The size of the new coming service is fixed to 5 MB. The execution time of the SP algorithm is slightly larger than the execution time of the traditional algorithms (as well as the *RW* algorithm). However, the difference is small, which makes the SP algorithm a viable option for real-time solutions.

5. Prior work

There are a lot of research works that addressed the memory management problem extensively in the past. However, the service model is different than that of the home applications. The most efficient traditional memory management algorithms are best-fit, worst-fit. In the experiment section, we compared them with our proposed algorithms in section 2. One of the main differences between memory management for smart home applications and general computer applications memory management in that the first one takes into account the priority of the application and subservices and the dependencies among the different services or bundles.

Vidal et.al. [19] addressed QoS in home gateway, they proposed a flexible architecture for managing bandwidth inside the home; however they have not addressed memory management in home gateways. In [24] we addressed memory management in home gateway but this work did not take priorities of the applications into consideration. To the best knowledge of the authors there is no study related to the memory management in the context of smart home applications. Ali et.al. [8] proposed architecture based on OSGi for wireless sensor network where data is processed in distributed fashion. They showed how to execute simple database queries like selection and join in a distributed fashion. [17] addresses protocol heterogeneity, interface fragmentation when connection several devices to OSGi-based gateway at home. The paper describes different scenarios and challenges for providing pervasive services in home applications.

6. Conclusions

This paper studied the problem memory management in the context of smart homes. One of the main differences between our problem and the traditional memory management is the priority of the applications and the dependencies among different services.

We proposed two algorithms; the first one is the *Relative Weights* algorithm that uses weight vector to represent the priority between applications. Furthermore subservices inherit the priority of the parent application. The second one is the *Strict Priority* algorithm, which assumes that high priority service is more important than any number of low priority services. We compared the proposed algorithms with the traditional memory management algorithms like best fit and worst fit. Simulation results indicate that *RW* and *SP* are much better than best fit and worst fit in terms of the total number of services kicked out and their priorities. At the same time, the proposed algorithms execution time is comparable to the execution time of the best fit and worst fit. In the future, we will study the optimal solution for the memory management problem within the above constraints.

References

[1] Zigbee Alliance: Zigbee specification: Zigbee document 053474r06 Version 1.0" (2004)

[2] Watanabe, K., Ise, M., Onoye, T., Niwamoto, H., Keshi, I.: An Energy-efficient Architecture of Wireless Home Network Based on MAC Broadcast and Transmission Power Control. In: IEEE Transaction on Consumer Electronics, vol. 53, no. 1, pp. 124—130 (2007)
[3] King, J., Bose, R., Pickles, S., Helal, A., Vander Ploeg, S., Russo, J.: Atlas: A Service-Oriented Sensor Platform. In: The 4th ACM Conference on Embedded Networked Sensor Systems (Sensys), Boulder, Colorado, USA (2006)
[4] Helal, A., Mann, W., El-zabadani, H., King, J., Kaddoura, Y., Jansen, E.: Gator Tech Smart House: A programmable pervasive space. In: IEEE Computer, vol. 38, no. 3, pp. 50—60 (2005)
[5] Lee, C., Nordstedt, D., Helal, A.: OSGi for Pervasive Computing. In: The Standards, Tools and Best Practice Department, IEEE Pervasive Computing, A. Helal, Dept. Editor, vol. 2, no. 3 (2003)
[6] Maples, D., Kriends, P.: The Open Services Gateway Initiative: An introductory overview. In: IEEE Communication Magazine, vol. 39, no. 12, pp. 110—114 (2001)
[7] Jansen, E., Yang, H., King, J., AbdulRazak, B., Helal, A.: A context driven programming model for pervasive spaces. In: The 4th International Conference on Pervasive Computing (2006)
[8] Ali, M., Aref, W., Bose, R., Elmagarmid, A., Helal, A., Kamel, I., Mokbel, M.: NILE-PDT: a phenomenon detection and tracking framework for data stream management systems. In: Proc. of the Very Large Data Bases Conference (2005)
[9] Sun Microsystems Inc.: Jini Architectural Overview ", http://www.jini.org/
[10] Sommers, F.: Dynamic Clustering with Jini Technology, http://www.artima.com/lejava/articles/dynamic_clustering.html
[11] Microsoft Corporation: Universal Plug and Play Device Architecture Reference Specification, Version 2.0, http://www.upnp.org/
[12] Jain, K., Vijay V. Vazirani: Approximation algorithms for metric facility location and k-Median problems using the primal-dual schema and Lagrangian relaxation. In: Journal of the ACM (JACM), vol. 48, Issue 2 (2001)
[13] Silberschatz, A., Peterson, J.: Operating System Concepts, Addison Wesley (1989)
[14] The OSGi Service Platform Release 4 Core Specification Ver 4.1, http://bundles.osgi.org/browse.php
[15] Binstock, A.: OSGi: Out of the Gates. Dr. Dobb Portal (2006)
[16] Ryu, I.: Home Network: Road to Ubiquitous World. In: International Conference on Very Large Databases (VLDB) (2006)
[17] Bottaro, A., Gérodolle, A., Lalanda, P.: Pervasive Service Composition in the Home Network. In: The 21st International IEEE Conference on Advanced Information Networking and Applications, Niagara Falls, Canada (2007)
[18] Margherita2000, The first washing machine on the Internet, http://www.margherita2000.com/sito-uk/it/home.htm
[19] Vidal, I., Garc´ıa, J., Valera, F., Soto, I., Azcorra, A.: Adaptive Quality of Service Management for Next Generation Residential Gateways. In: The 9th International conference on Management of Multimedia and Mobile Networks and Services, Irelanda, Dublin (2006)
[20] Ishihara, T., Sukegawa, K., Shimada, H.: Home Gateway enabling evolution of network services. In: Fujitsu Science Technical Journal, vol. 24, no. 4, pp. 446—453 (2006)
[21] Ishihara, T.: Home Gateway Architecture Enabling Secure Appliance Control Service. In: The 10th International conference on intelligence in network (ICIN'06) (2006)
[22] Garey, M., Johnson, D.: Computers and Intractability, Freeman, New York (1979)
[23] Johnson, D.S., Niemi, K.A.: On Knapsacks, partitions, and a new dynamic programming technique for trees. In: Mathematics of Operations Research, vol. 8, pp. 1—14 (1983)
[24] Kamel, I., Chen, B.: A Novel Memory Management Scheme for Residential Gateways. In: Special issue on Intelligent Systems and Smart Homes, International Journal Information System Frontiers, Springer (2008)