A Graph-based Approach for Automatic Service Activation and Deactivation on the OSGi Platform

Chin-Yang Lin, Cheng-Liang Lin, and Ting-Wei Hou, *Member*, IEEE

Abstract —*More and more mobile and embedded devices, such as home appliances and network devices, have selected OSGi as the software management platform. As a result, the resource management of the OSGi platform has become a critical issue. This paper focuses on how to enhance the efficiency of resource utilization in terms of the service activation and deactivation. We propose the Service Activator (SA), which is designed as an OSGi bundle, to on-demand activate and deactivate OSGi services, so that the resources required by services can be allocated and deallocated automatically. This involves a graph-based representation of services dependencies and two new algorithms. We have implemented the SA on an OSGi implementation (Knopflerfish); a home network prototype with a home surveillance scenario is presented to demonstrate the feasibility. Furthermore, a simulator is developed to further evaluate the SA in terms of several scenarios; the results show that the SA performs well for a wide range of bundles, and the processing overhead is low.*[1]

Index Terms — OSGi, resource-constrained device, eager resource allocation, service activation, graph-based.

I. INTRODUCTION

Consumer electronics (CE) appliances are typically resource-constrained; they have limited computing power, storage capacity and communication capabilities [1], [2]. Therefore, efficient utilization of shared resources (i.e. allocation and de-allocation) for these appliances becomes a critical issue [3]. Increasingly, these devices employ a middleware-based architecture and rely on the middleware to manage the resources [2], [4]. However, most of these middlewares, including the OSGi (Open Service Gateway initiative) [5], adopt the eager resource allocation (ERA) scheme [6], [7]. Under this scheme, the resource allocation is made before the resource is to be used; all the services are activated at system initialization and consume a significant amount of system resources. ERA thus comes with two major drawbacks (1) long startup time and (2) unnecessary resource consumption as the number of services increases.

In this paper, the OSGi is considered as the target platform, since it is widely used in many applications, including personal devices, automobiles, industrial automation, application servers and resource-constrained devices [8]-[10]. Many recent studies have focused on building an OSGi-based context-aware infrastructure in smart space environments [11]-[13] and the security issues [14], [15], which normally rely on the OSGi framework to manage the resources. In the OSGi specification, there are two mechanisms that can be used to mitigate the potential problems of ERA. The first is the Lazy Activation Policy [6] that allows activating bundles lazily by simply specifying this policy for them. This mechanism, to a certain extent, can save resources and system initialization time because bundles can be activated as they are first used. However, these lazy activation bundles would not be automatically deactivated even though they are not being used.

The second is the Declarative Services (DS), a new service component model integrated in OSGi R4 specification [7], [16]. It is based on a centralized instance manager to manage the components and their life cycle; each service component is associated with an XML-based description file. Application developers can achieve the service (or service component) activation and deactivation by carefully declaring the component descriptions. Unlike the Lazy Activation Policy, which is only performed on the bundles layer, DS performs the activation and deactivation of services, thus enabling finer-grained resource management. (Note that an OSGi bundle logically can contain more than one service) However, this declarative model may introduce more complexity. With the expansion of services, writing the description files would be error-prone. Moreover, since the dependencies between service components must be predefined, it would not be adaptive to the dependency changes at runtime.

In this paper, we attempt to propose an efficient way to enhance both the resource utilization and the scalability, especially when resources are scarce. Our approach is aimed at freeing application developers from manually dealing with resource allocation and deallocation while developing service-based OSGi applications. We approach the resource allocation problem by automating the service activation and deactivation. Technically, the Service Activator (SA), designed as an OSGi bundle, is introduced to automate the on-demand activation and deactivation of OSGi services. This involves a graph-based approach that monitors the services runtime dependencies and helps to determine whether a service should be activated/deactivated. The proposed approach has the following characteristics:

- Automation: The SA automatically activates services on-demand and deactivates them when they are not being accessed; it also enables the possibility of deactivating the services involved in circular references.

[1] This work was partially supported by the TOUCH center project of the National Science Council of Taiwan, R.O.C., under Grants No. NSC98-2218-E-006-003.

The authors are with the Department of Engineering Science, National Cheng-Kung University, Tainan City 701, Taiwan R.O.C (e-mail: {chinyang, chengliang, hou}@nc.es.ncku.edu.tw). The corresponding author is Ting-Wei, Hou (e-mail: hou@nc.es.ncku.edu.tw).

Contributed Paper
Manuscript received July 10, 2009

- Simplicity: Our approach is built on top of the OSGi framework and does not introduce a new component model. Also, it is intuitive and very simple to use; developers need not handle any description files.
- Backward Compatibility: The approach keeps to the original programming model, so it can apply to any OSGi-compliant framework; it can also be easily applied to the existing service-based OSGi applications in comparison with using the DS model.
- Flexibility: Developers can decide which services should be put under the control of the SA, so the management effort is limited to the significant services.

We have conducted an experiment based on an OSGi implementation (Knopflerfish [17]) and a simulation for further evaluating the SA to demonstrate the effectiveness of the proposed approach.

The rest of the paper is organized as follows. Section II presents our system architecture. Section III describes the Service Activator in detail. Section IV is our implementation and experimental results, followed by the conclusions.

II. System Architecture

The OSGi specifications [5] define a standardized service platform for developing and deploying service-oriented applications. The core of the service platform is the OSGi framework that sits on top of a Java virtual machine and supports the deployment of extensible and downloadable service applications. This platform follows the well-known service-oriented interaction pattern [18] that consists of three main roles: service providers, service requesters and a service registry. Service providers publish their services into the service registry, and service requesters query the service registry to discover the published services.

In OSGi, a service is normally described as a Java interface and is packaged along with its implementations into a modular unit called a *bundle*. A bundle is physically a JAR file that contains the associated code and resources, as well as a manifest file describing the information about the bundle; it can be installed, updated, or removed on the fly. Each bundle can publish (register) or use (request) services. We call the bundle that publishes services the *service-providing bundle*, and the bundle that uses services the *client bundle*. From a developer's perspective, bundles interact with each other through providing and using services to form an application.

Fig.1 illustrates the system architecture. The Service Activator (SA) plays the key role in achieving the automation of service activation and deactivation, which is implemented and deployed as an OSGi bundle. It contains two key parts: Service Dependency Graph (SDG) and Circular References Handler (CRH). The SDG is the core structure used to internally represent the services runtime dependencies; the CRH is used to deal with the circular dependencies according to the dependency information in the SDG.

In the proposed architecture, a new type of service called *m-service*, abbreviated from "managed service", is introduced; only the m-services would be put under the management of the SA; and each m-service bundle includes at least one m-service. Each m-service object must meet the requirement that the class of the object extends the abstract class *M-Service*, as shown in Fig. 2. It also implements service interfaces and behaves like standard OSGi service objects.

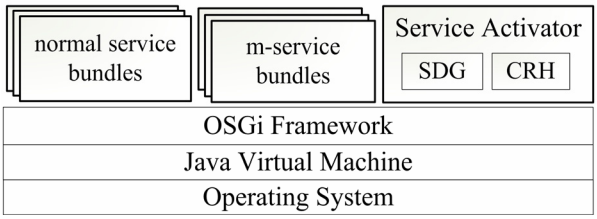

Fig. 1. System Architecture

One major difference is that each m-service object needs to implement two abstract methods, activate and deactivate, in *M-Service*. These two methods are designed for the SA to achieve the service activation and deactivation. For any m-service *s*, the activation/deactivation of *s* means the invocation of the activate/deactivate method of *s*. The activate/deactivate method is invoked by the SA and contains the resource allocation/deallocation code. The activation/deactivation of an m-service thus corresponds to the allocation/deallocation of the resources required by the m-service.

To capture the changes in runtime dependencies between bundles (that are not explicitly managed by the OSGi framework), each m-service is designed as a *service factory*. In OSGi, a service factory is essentially a service; any client can request/release it by invoking the standard getService/ ungetService method defined in the OSGi framework. It differs from a normal service in that: (1) it must implement the ServiceFactory interface (i.e. the getService and ungetService methods); (2) its getService method is invoked by the OSGi framework when a new client bundle requests it; and (3) its ungetService method is invoked by the OSGi framework when a client bundle releases it. Such an indirection mechanism in OSGi was originally designed to provide customized services. We utilize this mechanism to dynamically intercept the changes in dependencies and to separate m-services from normal services, which is accomplished by the customized methods getService and ungetService in *M-Service*.

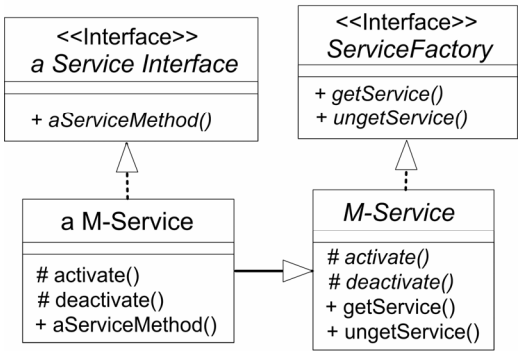

Fig. 2. The class diagram of M-Service

In Fig. 3, the sequence diagram of the getService operation is presented to illustrate how the above mechanism works, where a client queries the OSGi service registry to discover a

registered m-service by invoking the getService method. In this case, the m-service object is not returned directly. Instead, the getService method of the registered m-service (i.e. the getService method in *M-Service*) would be invoked by the OSGi framework, resulting in an invocation on the Service Activator (i.e. add dependency). The SA then updates the SDG accordingly and performs the service activation when needed, as shown in Fig. 3. Finally, the m-service object is returned to the client.

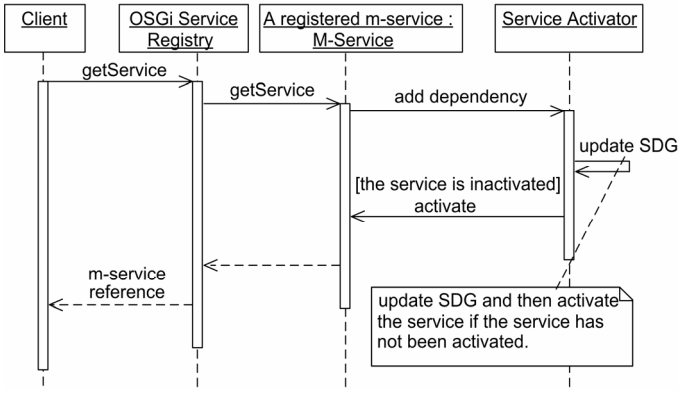

Fig. 3. Sequence diagram for get Service operation with Service Activator

Under this architecture, developers have the flexibility of choosing only the key services as m-services. All that is needed is to let each of the key services extend the *M-Service* and put the resource allocation/deallocation code in the activate/deactivate method.

III. The Service Activator

The Service Activator (SA) is designed as an OSGi bundle to provide two main functions: (1) the on-demand activation and deactivation of OSGi services and (2) the deactivation of the services involved in circular references. The key to these functions is the adoption of a service dependency graph (SDG) and two graph-based algorithms inspired by the algorithms used in garbage collection [19].

A. The Graph-based Approach

To efficiently accomplish the automation of service activation and deactivation, we need an efficient way to keep track of the dynamic service dependencies between bundles. Although the OSGi framework provides APIs for developers to look up the bundle-to-service dependencies, it is hard and actually time-consuming to obtain a whole picture of the dynamic dependencies among bundles and services. In order to satisfy our needs, we maintain a service dependency graph (SDG) to continuously monitor the runtime dependencies, in which the graph maintenance is carried out if any dependency changes during the running of the OSGi framework.

In addition to helping identify the services/bundles not being used, the graph-based approach brings several advantages. For example, the detailed structures and relationships given in the graph can assist in further analysis of the dependencies among bundles and services (e.g. cyclic dependency). Moreover, we can easily enrich the graph by adding additional properties, constraints or other meta-information to the graph nodes and/or edges. Such an extensibility creates the potential for optimization, particularly for decision-making tasks.

Since the algorithms presented herein all rely upon the SDG, it is important to understand how the SDG is maintained and how the elements in the graph are mapped to the OSGi services and bundles.

- *Graph Structure*

The service dependency graph (SDG) is a directed graph representing dependencies among bundles and services, with the nodes in the graph being the bundles, and the edges being the bundle-to-service dependencies. A bundle node in the graph encapsulates (1) a set of service entries representing the m-services registered by the bundle, (2) the outgoing edges with each one representing a dynamic dependency from the bundle to a registered m-service, and (3) properties/states specific to our algorithms. Since an OSGi application is normally composed of a set of dependent bundles, under this approach an application can be represented as a sub-graph of the SDG.

The example in Fig. 4 illustrates the structure of SDG. The bundle nodes and service entries are, respectively, represented with circles and squares. Each directed edge points from a bundle node (client bundle) to a service entry (target service); the source of the edge, plotted as a solid black circle, is included in each bundle node to implicitly signify all of the outgoing edges from the bundle.

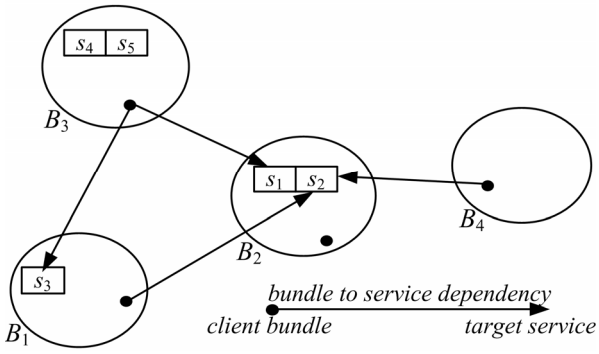

Fig. 4. An example of the service dependency graph

In this graph snapshot, SDG consists of four bundle nodes $\{B_1, B_2, B_3, B_4\}$ and four edges $\{(B_3, s_1), (B_3, s_3), (B_1, s_2), (B_4, s_2)\}$. Each of the nodes is a client and/or a service-providing bundle. It is easy to see that B_1 serves as both a client and a service-providing bundle, B_2 is a service-providing bundle, and both B_3 and B_4 are client bundles.

- *Graph Updates*

To keep the SDG up-to-date, additional care must be taken to deal with the situation where the runtime dependencies may change at any moment (due to the dynamic nature of OSGi). The SDG is initially empty and is updated through an event-driven adaptation algorithm that modifies the graph structure whenever one of the following events occurs.

1. START_BUNDLE Event: This event refers to the "start bundle event" defined in the OSGi framework. It causes a bundle node creation.

2. STOP_BUNDLE Event: This event refers to the "stop bundle event" in the OSGi framework. It causes the removal of a bundle node and the dependencies related to the node (i.e. the incoming and outgoing edges).
3. ADD_DEPENDENCY Event: This event, written AD(*B*, *s*), is triggered when a client bundle *B* asks for (via the getService method) an m-service object *s*. It causes the addition of the edge (*B*, *s*) and a target service entry for *s* (if the entry does not exist).
4. REMOVE_DEPENDENCY Event: This event, written RD(*B*, *s*), is triggered when a client bundle *B* releases (via the ungetService method) a service object *s*. It causes the removal of the edge (*B*, *s*).

Here, the events START_BUNDLE and STOP_BUNDLE are captured by simply listening to the changes in bundles' lifecycle states (notified by the OSGi framework). The events ADD_DEPENDENCY and REMOVE_DEPENDENCY are respectively mapped to the invocation of the methods getService and ungetService, which, as mentioned earlier, are specific to the m-services. Since in OSGi, stopping a bundle will cause all the services registered by this bundle to be automatically unregistered, triggering a STOP_BUNDLE event can therefore introduce a number of the REMOVE_DEPENDENCY events to be triggered.

B. Service Activation and Deactivation

The automation of service activation and deactivation means the ability to activate m-services on-demand and deactivate them when no clients are accessing them. To this end, we employ an algorithm modified from the reference counting algorithm used in garbage collection [19]. The algorithm is seamlessly incorporated into the SDG manipulation.

The basic idea is to keep a reference count for each m-service, say *s*; the reference count of *s*, denoted by *rc*(*s*), is used to reflect the number of bundles that are currently using *s*, and is stored in the service entry associated with *s*. Whenever a dependency to *s* is established, *rc*(*s*) is incremented, and whenever one is removed, *rc*(*s*) is decremented. In the case where *rc*(*s*) is decremented to zero, *s* would be eligible for deactivation.

Additionally, we keep a state per service entry to signify whether a given m-service has already been activated. Each m-service is always in one of two states: INACTIVE and ACTIVE. The INACTIVE state is an initial state and indicates that the service is not ready to serve. This implies that either the service has never been activated or the service has been deactivated. The ACTIVE state means that the service is ready to serve, which implies that the activate method of the service must have been invoked and the service has not been deactivated since then.

The SA makes decisions on whether to activate or deactivate an m-service according to the above runtime information. Given an m-service *s*, the activation and deactivation of *s* are performed, respectively, when *rc*(*s*) is incremented to 1 such that *s* is INACTIVE and when *rc*(*s*) drops to zero such that *s* is in the ACTIVE state. Since the increment is only attributed to the ADD_DEPENDENCY event, the service activation must be on-demand (i.e. postponed until a client actually wants to use the service). The service deactivation can occur while triggering the REMOVE_DEPENDENCY event or the STOP_BUNDLE event, because both of the events lead to the removal of edge.

On the other hand, the SA introduces a new OSGi bundle header in the manifest file, called ALWAYS_ON, which developers can use to designate the bundles whose m-services are not intended to be deactivated. That is because, in some cases, a bundle may include only the services that are supposed to be always running, e.g. a service which encapsulates a thread of control and continuously monitors some real-time information. It can also be expensive if a considered m-service is frequently unregistered and re-registered or if the cost of activating and deactivating it is high. Application developers can prevent these kinds of services from being deactivated by simply tagging the bundles registering them with the ALWAYS_ON header. The content of the header is read while the START_BUNDLE event is triggered. One can think of this header as a means to reduce the unnecessary computation and resource reallocation.

Fig. 5 outlines the procedures for the service activation and deactivation, both of which are embedded in the graph modification. Each time an ADD_DEPENDENCY event is triggered, the SA invokes the procedure service_activation. It creates a graph edge from a client bundle node to a target service entry, followed by the operations related to the service activation. The procedure service_deactivation is aimed at removing a graph edge and performing the service deactivation if needed. Note that the SA does not deactivate any service whose enclosing bundle is tagged as ALWAYS_ON (checked in the if-statement), even though the reference count of the service is zero.

```
service_activation(clientBundleID, targetMService)
1. clientBundleNode := SDG.getBundleNode(clientBundleID);
2. targetServiceEntry := SDG.getServiceEntry(targetMService);
3. SDG.addEdge(clientBundleNode, targetServiceEntry);
4. targetServiceEntry.rc++;
5. if !targetServiceEntry.isActive() then
6.   targetMService.activate();
7.   targetServiceEntry.setActive(true);
8. endif

service_deactivation(clientBundleID, targetMService)
1. clientBundleNode := SDG.getBundleNode(clientBundleID);
2. targetServiceEntry := SDG.getServiceEntry(targetMService);
3. SDG.removeEdge(clientBundleNode, targetServiceEntry);
4. targetServiceEntry.rc--;
5. if targetServiceEntry.rc = 0 and targetServiceEntry.isActive()
6.    and !targetServiceEntry.isALWAYS_ON() then
7.   targetMService.deactivate();
8.   targetServiceEntry.setActive(false);
9. endif
```

Fig. 5. The algorithm for service activation and deactivation

Unlike the traditional reference counting, our algorithm does not carry out the service deactivation in a recursive manner. That is, deactivating a service cannot result in more

services being deactivated; the service deactivation time is hence bounded. Furthermore, the deactivation work can also be performed on the bundles layer. When a bundle is stopped, a group of services registered by the bundle would be considered for deactivation as a whole. For example, consider the SDG in Fig. 4, and suppose that a STOP_BUNDLE event for B_2 is then triggered. In this case, stopping B_2 involves triggering the events RD(B_1, s_2), RD(B_3, s_1) and RD(B_4, s_2), so the services s_1 and s_2 will be deactivated eventually (i.e., both $rc(s_1)$ and $rc(s_2)$ reach zero).

C. Circular References Handler

The Circular References Handler (CRH), designed as part of the SA, is targeted at dealing with the circular dependencies, which enables the possibility of deactivating the services involved in circular (or cyclic) references and hence reduces the unnecessary resource consumption.

A circular dependency is a relation between two or more bundles which either directly or indirectly reference each other. For example, when a bundle B_1 references a service provided by another bundle B_2 and B_2 references a service provided by B_1, we say that B_1 and B_2 form a circular dependency. Both B_1 and B_2 are called circular bundles. In OSGi, a bundle essentially can be a client and/or a service-providing bundle, so any circular dependency is considered normal.

To illustrate the potential problem, consider the SDG in Fig. 6, constructed from the following scenario: (1) service request from B_3: AD (B_3, s_1) is first triggered, followed by AD (B_1, s_2); and (2) service request from B_4: AD (B_4, s_3) is then triggered, followed by AD (B_2, s_1).

It can be seen that AD (B_3, s_1) and AD (B_4, s_3) result in the circular dependency between B_1 and B_2, as well as the activation of s_1, s_2 and s_3. Assume that RD (B_3, s_1) and RD (B_4, s_3) are then triggered. The SA will only deactivate s_3, while leaving both s_1 and s_2 as they are, because the circular dependency cannot be broken by either of these two events. More precisely, both s_1 and s_2 would remain ACTIVE until the occurrence of another event that can break the circular references (e.g. stopping B_1 or B_2). In the worst case, s_1 and s_2 would never be deactivated, even though it may be safe to deactivate both of them (this is application dependent).

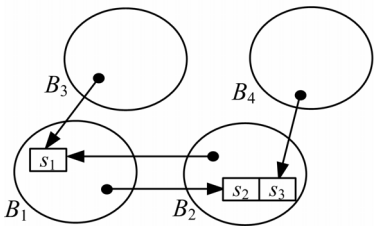

Fig. 6. An example of circular dependency

To automate the deactivation of the services involved in this situation, two points must be addressed: (1) identifying the circular bundles and (2) determining whether the identified bundles are eligible for deactivation. With SDG, the former is trivial, since finding cycles in a directed graph is a typical problem and would not be a key issue here. However, the latter is technically hard, since the OSGi framework does not provide any runtime information about whether a given bundle can be safely deactivated. That is the reason why the bundles B_1 and B_2 in the above case can only be deactivated by manually breaking the circular references (e.g. stopping B_1).

We propose a method, realized as the CRH, to facilitate this automation. The basic idea is to predefine candidates (bundles) for possible deactivation, followed by a runtime analysis of the SDG for identifying which of these candidates can be safely deactivated, and then to deactivate the identified candidates. Here, deactivating a candidate bundle means deactivating all the services in the bundle, i.e., each of the services in the bundle would be in the INACTIVE state.

The step of predefining candidates is performed at design time, where developers designate part or all of the bundles as candidates for possible runtime deactivation. Each of the candidates must meet the requirement that it can be safely deactivated when it is not being accessed from any application entry. The application entry is the entry (starting) point of an application. An application is normally composed of a set of bundles and includes at least one application entry.

Since the application entries actually reflect the sources of service requests, the criterion for designating candidate bundles is that each candidate bundle must not include any application entries. In general, a candidate should include only the services designed as utility routines. The utility routine is used as needed; it performs a single task or a small range of tasks; and particularly, it is often considered as a "passive" service that does not own a thread of control and must rely on another thread to execute its code. It is difficult to identify such candidates at runtime, but it is intuitive for the developers.

Similar to the ALWAYS_ON header, CRH introduces a new header, called PASSIVE, which the developers can use to specify the candidates. These candidates are also called PASSIVE bundles. Additionally, it is obvious that any bundle tagged as ALWAYS_ON cannot be tagged as PASSIVE.

At runtime, the CRH is launched (when appropriate) for deactivating the candidates that are not being accessed from any of the application entries. We consider all the non-PASSIVE bundles as the application entries (i.e. the maximal set of application entries), called Roots in SDG. The target problem of CRH is thus to identify the PASSIVE nodes that are unreachable from any of the Roots in SDG.

To this end, we devise an algorithm based on the mark-scan garbage collection technique [19], which can correctly find the graph nodes unreachable from a given set of nodes in the graph through a simple tracing procedure, even though the nodes are involved in cyclic structures.

The new algorithm consists of three phases: MarkPhase, which traces the SDG, starting from the nodes in Roots, to mark all the visited nodes; ScanPhase, which scans all the

nodes in SDG for identifying the unmarked (i.e. unreachable) PASSIVE nodes; and finally CollectPhase, which deactivates the bundles identified in ScanPhase and updates the SDG accordingly.

To avoid repeated deactivations, we add a boolean flag per bundle node to signify whether a given bundle is deactivated by CRH. For any bundle, this flag is initially false and is set to true whenever the bundle is deactivated by CRH; it is reset to false as soon as any of the included services is activated via the ADD_DEPENDENCY event. The CollectPhase will exclude all the bundles deactivated by CRH.

Fig. 7 depicts the pseudocode of the algorithm. First, the procedures mark_phase and mark collaborate to find and mark all the nodes reachable from Roots by performing *depth-first search* from each node in Roots. The procedure scan_phase then scans all the nodes in SDG to obtain the unmarked PASSIVE nodes and reset the mark flag for each marked node, where each of the unmarked PASSIVE nodes whose associated bundles are not deactivated by CRH will be put in a list, termed the DeactivationList. Finally, the procedure collect_phase deactivates the bundles whose corresponding nodes are contained in the DeactivationList. Recall that deactivating a bundle means deactivating all the services in the bundle. By removing all the incoming edges of a bundle node, all the services provided by the bundle can be naturally deactivated. Here, each removal of an edge corresponds to the procedure service_deactivation in Fig. 5.

In the algorithm, both the MarkPhase and ScanPhase are dedicated to the detection effort. The detection time is dominated by the MarkPhase because it needs to traverse the entire *transitive closure* of Roots; the complexity is thus limited by the size of that transitive closure. As for the collection time (i.e. the time for CollectPhase), it entirely depends on the number of bundles identified in the ScanPhase.

```
mark_phase()
1.  for B in Roots do
2.    if !B.isMarked() then
3.      mark(B);
4.    endif

mark(B: BundleNode)
1.  B.setMarked(true);
2.  for all edges (B, T) do
3.    if !T.isMarked() then
4.      mark(T);
5.    endif

scan_phase()
1.  for B in SDG do
2.    if B.isMarked() then
3.      B.setMarked(false);
4.    else if B.isPASSIVE() and
5.       !B.isDeactivated() then
6.      DeactivationList.add(B);
7.    endif

collect_phase()
1.  for B in DeactivationList do
2.    SDG.removeIncomingEdges(B);
3.    B.setDeactivated(true);
```

Fig. 7. The algorithm for Circular References Handler

Consider the example in Fig. 6 again and assume that RD (B_3, s_1) and RD (B_4, s_3) have been triggered. All that is needed under this approach is to tag the bundles B_1 and B_2 with the PASSIVE header. Then, at runtime, s_1 and s_2 would be automatically deactivated while performing the CRH, since both B_1 and B_2 are unreachable from Roots in this case.

Our approach only requires developers to determine which of the bundles should be considered for deactivation; it does not matter whether these bundles are involved in circular references. Ideally, the CRH should be triggered on-demand. Our current design of the trigger is simply based on the available memory, which is checked after each service deactivation to determine whether to perform the CRH. The triggering occurs only if the available memory falls below a preselected threshold.

IV. Experimental Results

We implemented the proposed approach on an OSGi implementation, Knopflerfish 2.2.0 [17], and conducted a simulation for evaluating the SA to demonstrate the effectiveness of the proposed approach. All the experiments are performed on an x86-based platform with a VIA EPIA N-Series Nano-ITX platform, using the operating system Debian kernel 2.6 and Java 2 Platform Standard Edition version 1.6. This platform is equipped with a 1G VIA processor and 512MB RAM with only required services. We first demonstrate our implementation through a prototype and then present the simulation.

A. Implementation

Our implementation (based on Knopflerfish [17]) is designed as an OSGi service gateway in a home network, which employs a remote control interface modified from the httpconsole bundle developed by Knopflerfish. Coupled with a web server on the OSGi platform, users can access the registered services via the remote control interface (i.e. a web-based user interface). This interface also displays memory usage information.

A scenario based on a "home surveillance service" is presented to demonstrate how the Service Activator on the OSGi platform works. The home surveillance service is implemented as an m-service. It allows pictures captured by the cameras in the house to be monitored remotely, where the cameras are connected to the home network via wireless LAN and are controlled by the service.

A likely scenario is that the parents may need to remotely watch over their children and house while they are away. By installing and starting the bundle that publishes the home surveillance service on the gateway, the parents can easily use this service through the web browser of a mobile device (e.g. a smart phone) to take pictures of their kids or house, thus achieving the remote control/monitor capability.

Fig. 8 illustrates this scenario by showing screenshots of the web-based interface on a smart phone, where screenshot (a) is the central home page where users can access services; screenshot (b) is a console showing the states of the bundles and services; screenshot (c) shows a result of using the home surveillance service (by clicking the link named "HomeSurveillance" in the home page); and finally the user goes back to the home page, as shown in screenshot (d). This figure reveals the difference in the memory consumption before and after using the home surveillance service. That is because this service is an m-service which is activated on-demand and deactivated by the Service Activator when it is not being accessed.

In this study, we use this prototype with a simple scenario to demonstrate the feasibility of our approach. In practice, the effectiveness would be increased with the use of more bundles and services on the home gateway.

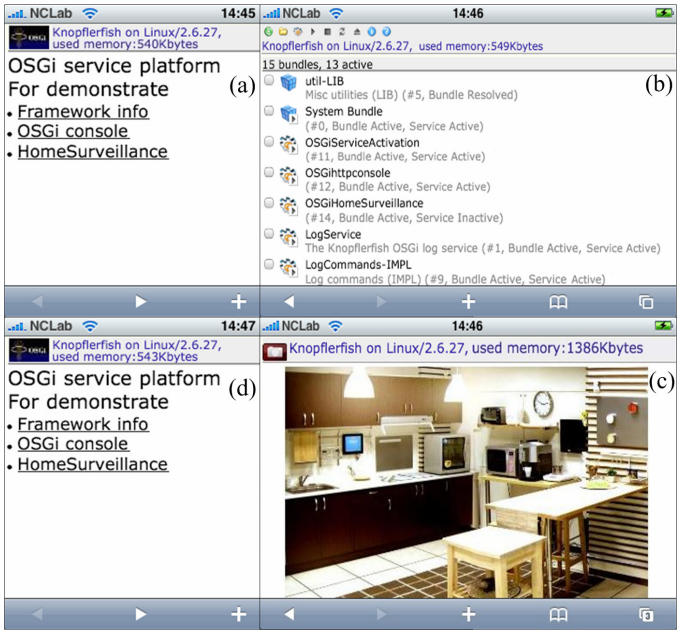

Fig. 8. Illustration of the scenario

B. Simulation

Since there is no standard benchmark for evaluating the proposed approach, we conducted a simulation to get a better understanding of the approach. The simulation setup consists of two independent phases, as shown in Fig. 9. First, the event generator produces the event sequence files based on a set of predefined configurations. Each file contains a sequence of the four types of events presented in Section III-A to represent a scenario. In the second phase, the simulator that serves as the Service Activator reads and interprets these event sequence files to run scenarios in an automated way.

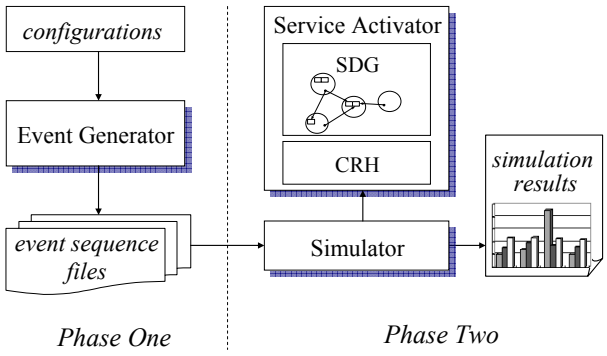

Fig. 9. Experimental Setup

In practice, the resources required by a service may include CPU, memory, files, network and database connections. In order to simulate the service activation and deactivation, each service (represented by an ID number) in these sequence files is assigned a quantitative value, in the range from 1 to 10, to indicate the amount of resource required by the activation of a service; the higher the value, the more the resources consumed by the service. The system resource consumption with the Service Activator is evaluated in terms of the amount contributed by the service activations and deactivations, where the amount is increased and decreased, respectively, for the activation and the deactivation.

The event generation process is based on the random selection. Each event sequence file is associated with a configuration that specifies the ranges of random selections and the proportion of both event types and bundle tags. A configuration is defined by the following parameters:

- Bundle numbers (n): Bound of the size of SDG.
- Maximum number of services per bundle (p): The number of services per bundle is in the range from 0 to p.
- Event numbers (m): Size of the generated event sequence (say E), where $m \geq n$, and the first n events in E are START_BUNDLE events.
- Proportion of deactivation events (α): α = (the number of deactivation events in E)/(m - n), where a deactivation event is either a REMOVE_DEPENDENCY event or a STOP_BUNDLE event, each of which accounts for half of the number of deactivation events. The proportion of activation (i.e. ADD_DEPENDENCY) events is (1 - α).
- Proportion of ALWAYS_ON bundles (β): β = (the number of ALWAYS_ON bundles)/n.
- Proportion of PASSIVE bundles (γ): γ = (the number of PASSIVE bundles)/n(1 - β), where the ALWAYS_ON bundles are excluded from being the PASSIVE bundles.

All the configurations in this experiment use the same values of the parameters: n = 1000, m = 2000 and p = 10. The varying parameters are α, β and γ. Each presented result is an average over 10 runs, each of these runs uses a different event sequence file generated from the same configuration. Since n = 1000 and m = 2000, only the second half of the entire event sequence is displayed.

Fig. 10 displays the resource impact of different event distributions, measured on varying α. The ratio of the resource consumption with SA to that without SA (i.e. normal) is plotted; the lower the ratio, the greater the resources saved by the SA. It can be seen from Fig. 10 that the resource consumption with SA decreases with increasing the proportion of deactivation events (i.e. α).

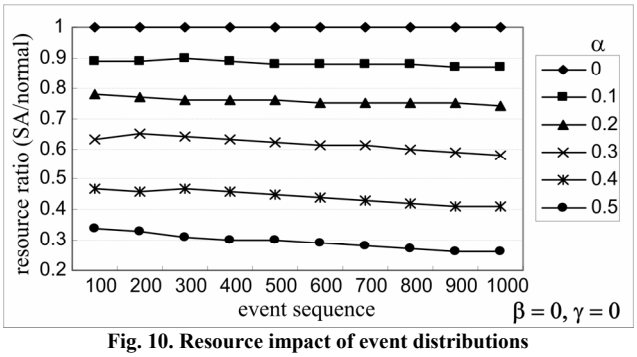

Fig. 10. Resource impact of event distributions

Based on the same settings, the overall SDG processing time is presented in Fig. 11. The node creation time does not obviously vary with the α values, which is actually limited by *n*. Both the creation and removal of edges depend on α, in which the higher the α value, the more time the SA spends on removing edges. Particularly, the results of the total processing time are all around 25 ms, which implies that the overhead of the service activation and deactivation is quite low. (Note that these operations would be spread out during the running of the OSGi framework.)

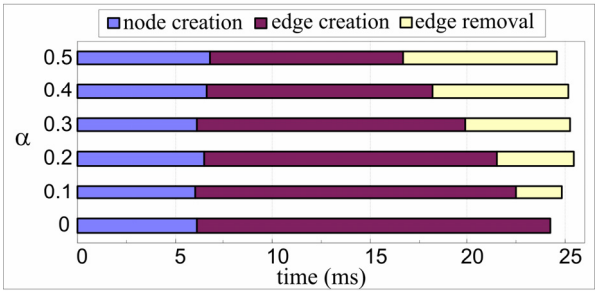

Fig. 11. SDG Processing Time

Fig. 12 shows the resource impact of using ALWAYS_ON header, where α = 0.2 and γ = 0. It is easy to see that the resources saved by the SA decrease with increasing β. That is because ALWAYS_ON bundles would essentially prevent services from being deactivated.

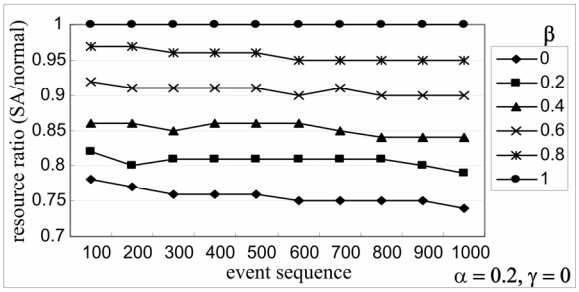

Fig. 12. Resource impact of using ALWAYS_ON header

In Fig. 13, we trigger the CRH per 100 events to observe how the CRH would affect the resource consumption, where α = 0.2, β = 0 and the PASSIVE header is taken into account (i.e. γ). It is not surprising that the resources saved by the SA increase with increasing the number of PASSIVE bundles (i.e. candidates). This can also be seen in the column "Services Deactivated" of TABLE I, where the number of services deactivated increases with increasing γ.

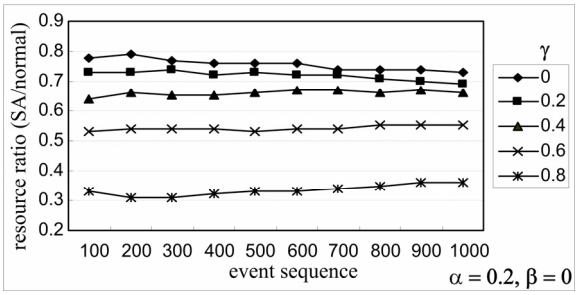

Fig. 13. Resource impact of using PASSIVE header (with CRH)

TABLE I presents more details about the CRH, where "CRH Time" represents the overall execution time of CRH; "Detection Time" represents the time required by the MarkPhase and ScanPhase of CRH, normalized to "CRH Time"; and "Collection Time" represents the time required by the CollectPhase of CRH, normalized to "CRH Time". The overall execution time is composed of the detection time and the collection time. According to this table, the detection time depends on the number of bundles marked, and the collection time hinges on the number of services deactivated. Since the overall execution time clearly depends on the detection time (according to the trend), it is thus still dominated by the MarkPhase of CRH (i.e. the traversal of the SDG).

TABLE I
BEHAVIORS OF THE CRH

γ	Services Deactivated	CRH Time (ms)	Detection Time (%)	Collection Time (%)	Bundles Marked
0	0	34.9	100%	0%	4782
0.2	21	32.7	96%	4%	4202
0.4	76	29.3	90%	10%	3566
0.6	187	27.9	78%	22%	2723
0.8	389	27.6	59%	41%	1543

Finally, we measure the services deactivated by CRH based on the settings in Fig. 13 with different values of β. As shown in Fig. 14, the number of services deactivated decreases with increasing β. That is because ALWAYS_ON bundles are always considered as part of Roots (i.e. non-candidates).

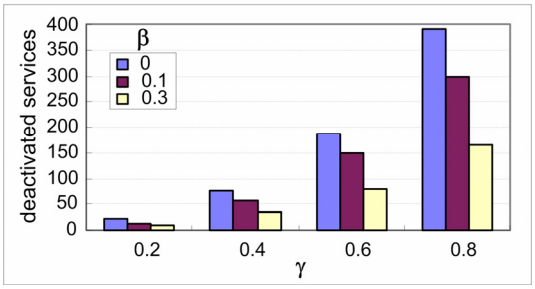

Fig. 14. Deactivated services with a mixture of ALWAYS_ON and PASSIVE headers

Overall, the simulation results show that the SA behaves well for a wide range of event sequences and bundle numbers, which significantly decreases the resource consumption by automatically performing the service activation and deactivation at runtime. By using the CRH, more bundles or services can be deactivated, thus further reducing the resource consumption. Moreover, the processing overhead of the SA is almost negligible, since the execution time is very short.

V. CONCLUSIONS AND FUTURE WORK

We have presented a novel approach for automating the service activation and deactivation on the OSGi platform, where services are activated on-demand and are deactivated when no clients are accessing them. The activation and deactivation of a service correspond to the allocation and deallocation of the resources required by the service, thus enhancing the efficiency

of resource utilization. This automation essentially provides fine-grained resource management, performing activation and deactivation on the services layer. In particular, our approach is simpler in both design and use in comparison with existing mechanisms; it is built on top of the OSGi framework, involves no complex component model and keeps to the original programming paradigm. This approach can significantly ease the difficulties of handling complicated resource allocation and deallocation while developing OSGi applications.

The proposed approach has been designed and implemented as an OSGi bundle, called Service Activator. The key is the adoption of a service dependency graph and two graph-based algorithms inspired by the algorithms used in garbage collection. The conducted experiments, based on an OSGi implementation, a home network prototype and a precise simulation, have shown the feasibility and effectiveness of this novel approach.

In future work, we plan to enrich the dependency graph by introducing more properties concerning the services or bundles, so that the Service Activator can more intelligently determine whether to activate/deactivate a service. We also plan to enhance the mechanism of triggering CRH applied in this study by considering more runtime resource information, such as CPU loading and available memory. This will help the Service Activator to more accurately identify the triggering points for the CRH.

REFERENCES

[1] A. Moller, M. Akerholm, J. Fredriksson, and M. Nolin, "Evaluation of component technologies with respect to industrial requirements," *Proc. of EUROMICRO'04*, pp. 56-63, 2004.

[2] S. Malek, M. Mikic-Rakic, and N. Medvidovic, "A style-aware architectural middleware for resource constrained, distributed dystems," *IEEE Trans. Software Eng.*, vol. 31, no. 3, pp. 256-272, March 2005.

[3] J. S. Rellermeyer and G. Alonso, "Concierge: a service platform for resource-constrained devices," *Proc. of EuroSys'07*, pp. 245-258, March 2007.

[4] F. Guidec, Y. Maheo, and L. Courtrai, "A java middleware platform for resource-aware distributed applications," *Proc. of ISPDC'03*, pp. 96-103, 2003.

[5] Open Service Gateway initiative Alliance, http://www.osgi.org/.

[6] Lazy Activation Policy, OSGi Service Platform Core Specification, Release 4, section 4.

[7] Declarative Services Specification Version 1.0, OSGi Service Platform Service Compendium, Release 4, section 112.

[8] A. Ibrahim and L. Zhao, "Supporting the OSGi service platform with mobility and service distribution in ubiquitous home environments," *The Computer Journal*, vol. 52, no. 2, pp. 210-239, 2009.

[9] J. S. Rellermeyer, O. Riva, and G. Alonso, "AlfredO: an architecture for flexible interaction with electronic devices," *Proc. of the ACM/IFIP/USENIX 9th International Middleware Conference*, pp. 22-41, Dec. 2008.

[10] J.-E. Lim, O-H. Choi, and D.-K. Baik, "An evaluation method for dynamic combination among OSGi bundles based on service gateway capability," *IEEE Trans. Consumer Electron.*, vol. 54, no. 4, pp. 1698-1704, 2008.

[11] T. Gu, H. K. Pung, and D. Zhang, "A service-oriented middleware for building context-aware services," *Journal of Network and Computer Applications*, vol. 28, no. 1, pp. 1-18, 2005.

[12] R. P. Diaz Redondo, A. F. Vilas, M. R. Cabrer, J. J. Pazos Arias, J. G. Duque, and A. Gil-Solla, "Enhancing residential gateways: a semantic OSGi platform," *IEEE Intelligent Systems*, vol. 23, no. 1, pp. 32-40, 2008.

[13] R. P. Diaz Redondo, A. F. Vilas, M. R. Cabrer, J. J. Pazos Arias, and M. R. Lopez, "Enhancing residential gateways: OSGi services composition," *IEEE Trans. Consumer Electron.*, vol. 53, no. 1, pp. 87-95, 2007.

[14] P. Parrend and S. Frenot, "Security benchmarks of OSGi platforms: toward Hardened OSGi," *Software: Practice and Experience*, vol. 39, no 5, pp. 471- 499, 2009.

[15] P. H. Phung and D. Sands, "Security policy enforcement in the OSGi framework using aspect-oriented programming," *Proc. of COMPSAC'08*, pp. 1076-1082, 2008.

[16] H. Cervantes and R. S. Hall, "Automating service dependency management in a service-oriented component model," *Proc. of the 6th International Workshop on Component-Based Software Engineering (CBSE)*, pp. 91-96, 2003.

[17] Knopflerfish, http://www.knopflerfish.org/.

[18] R. S. Hall and H. Cervantes, "Challenges in building service-oriented applications for OSGi," *IEEE Communications Magazine*, vol. 42, no. 5, pp. 144-149, May 2004.

[19] R. E. Jones and R. D. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, John Wiley and Sons: New York, 1996, pp. 19-41.

Chin-Yang Lin received the B.S. and M.S. degrees in Engineering Science from National Cheng Kung University, Taiwan in 1999 and 2001, respectively. He is now a Ph.D candidate in Engineering Science, National Cheng Kung University. His major research interests include ubiquitous computing, automatic memory management, embedded Java platform and software engineering.

Cheng-Liang Lin received the B.S and M.S. degree in Computer Science and Information Engineering from Shu-Te University, Taiwan, in 2003 and 2005. And now he is a Ph.D candidate in Engineering Science, National Cheng Kung University. His major research is in software methodologies, middleware collaboration, services design for ubiquitous computing environments, Interactive Digital TV and Java related technology.

Ting-Wei Hou received BS, MS, and Ph.D degrees all in Electrical Engineering, National Cheng Kung University, Taiwan, in 1983, 1985, and 1990 separately. He has been an associate professor in Department of Engineering Science, National Cheng Kung University since 1990. He was a visiting scholar of CSRD of University of Illinois at Urbana-Champaign, Illinois, U.S.A, during 1993-1994. His major research is in embedded systems and system integration. He has been the project leader of the pilot project on Healthcare IC cards in Penghu, Taiwan from 1998 to 2003, which encouraged the National Healthcare IC card project. He is currently working on Java based embedded systems, such as Java Virtual Machines, Java obfuscators, MHP, OSGi, Java card applications, and RFID applications.