A Platform for Supporting Dynamic Update and Resource Protection in an Embedded Operating System

MEI-LING CHIANG\(^{1}\) AND HSIANG-YU HSU\(^{2}\)

Department of Information Management
National Chi-Nan University
Nantou, 545 Taiwan
E-mail: \{joanna\(^{1}\); s96321506\(^{2}\}\}@ncnu.edu.tw

Recently, many researches focus on providing dynamic update functionality in embedded systems since dynamic update allows a system to enhance or update its functionality even without the need of rebooting the whole system or stopping system services. Dynamic update thus provides more flexibility in the development of embedded systems. However, an incautiously developed component once downloaded may corrupt the system or waste system resources. Because embedded systems’ resources are usually limited, protecting system resources against erroneous components is very important, especially in the systems providing dynamic update.

In this paper, we have implemented a platform which supports a remote component update mechanism for dynamically upgrading an embedded operating system at run time. Besides, a system resource protection mechanism is implemented for protecting system resources against downloaded un-trusted components. If our system detects misuses of system resources from an erroneous component, it will reclaim the wasted resources and remove the erroneous component out of our embedded client. Currently, our protection mechanism can reclaim lost memory space, ensure the normal execution of critical sections, and prevent null pointer access. The experimental results demonstrate that our platform can effectively support dynamic remote update and prevent incautiously developed components to misuse system resources with only little extra overhead.

Keywords: embedded operating systems, dynamic update, resource protection, embedded systems, garbage collection

1. INTRODUCTION

As the rapid development of hardware and maturity of computer technologies, embedded systems’ functions have become more and more complex. Usually, it is difficult or impossible to reclaim the deployed or sold embedded systems to upgrade their functionalities. Therefore, a dynamic update mechanism is needed for allowing embedded systems to update their software on-the-fly. Many embedded operating systems have already supported different kinds of dynamic update mechanism to upgrade or enhance their functionalities. Because embedded systems’ resources such as CPU, memory, and energy are usually restricted, an incautiously developed component that misuses system resources may corrupt system services or waste system resources once it is downloaded. For example, if components have allocated system resources but do not release them, this situation might cause deadlock problem. Therefore, in a system that supports dynamic update mechanism, providing resource protection is also a very important issue.
In this paper, we describe our experience in the implementation of a platform that can effectively support both remote dynamic update and resource protection in an embedded operating system. This platform provides two modes of dynamic update to extend or upgrade an embedded client’s functionalities. Under the pull (i.e. demand loading) mode, a requested component can be dynamically pulled and downloaded from the server into the embedded client. Under the push (i.e. multicast) mode, a dynamic update dissemination mechanism is implemented. The server would periodically broadcast new components’ information to the network. Embedded clients would listen to the broadcast message and then request for downloading the new components that they need. Meanwhile, administrator can also manually replace embedded clients’ components through set-up shell scripts. Besides, component dependency is maintained in the way that all dependent components would be downloaded together into the requesting embedded client. In particular, both the kernel components and application components can be updated or extended.

We have further implemented some schemes to protect embedded clients’ system resources. Each embedded client will record the information of allocated system resources. Through this system resource protection mechanism, our embedded client can detect whether a downloaded component has improperly used system resources. If system resource is misused by a downloaded component, this erroneous component will be removed out of the system. Currently, our mechanism can reclaim lost memory space, ensure the normal execution of critical sections, and avoid null pointer access.

In order to demonstrate the feasibility of our mechanisms, we have designed and implemented these mechanisms in LyraOS embedded operating system [1, 2] on ARM Integrator/CP920T development board [3]. Although the original LyraOS has already supported a dynamic update mechanism [2] to extend its kernel, it only supports demand loading functionality. After incorporating the implementation of our work, Lyra-OS kernel image is increased by only 3.5 Kbytes. The experimental results show that our platform can effectively protect embedded clients’ system resources against incautiously developed components with only little extra overhead.

The rest of the paper is organized as follows. Section 2 introduces the background and related works. Section 3 presents our platform design and implementation. Section 4 shows our performance evaluation results. Finally, section 5 concludes this paper.

2. BACKGROUND AND RELATED WORKS

This section introduces some related works on embedded operating systems. Section 2.1 introduces LyraOS embedded operating system. Section 2.2 describes some dynamic update mechanisms and protocols. Section 2.3 introduces software-based resource protection and recovery.

2.1 LyraOS Embedded Operating System

LyraOS [1, 2] is a component-based embedded operating system. It consists of a set of well-designed system software components. Each system component is completely separated, self-contained, and highly modular. In addition to being light-weight system software, it is a time-sharing multi-threaded microkernel.
A PLATFORM FOR SUPPORTING DYNAMIC UPDATE AND RESOURCE PROTECTION

Fig. 1 shows the system architecture of LyraOS. It is designed to abstract the hardware resources of computer systems such that low-level machine dependent layer is clearly cut from higher-level system semantics. Thus, it can be easily ported to different hardware architectures [1]. On top of the LyraOS microkernel, a micro window component with Windows OS look and feel is provided. Besides, the LyraFILE component [4] which is a light-weight VFAT-based file system supports both RAM-based and disk-based storages. Especially, LyraOS provides the Linux device driver emulation environment [5, 6] to make use of Linux device drivers. Under this emulation environment, Linux device driver codes can be integrated into LyraOS without modification. Furthermore, the LyraNET [7] component, a TCP/IP protocol stack derived from Linux TCP/IP codes, is provided. In order to adapt into embedded systems, LyraNET is further implemented with the zero-copy mechanism for reducing protocol processing overhead and memory usage.

In the recent work [2], LyraOS supports dynamic component update and memory protection mechanisms. The dual-mode operation is used to provide protection to enhance its security against un-trusted components. Trusted components and the LyraOS kernel are located in the kernel mode, whereas, un-trusted components are located in the user mode. Especially, the server-side pre-linking mechanism is proposed and implemented for decreasing the overhead of dynamic updating.

2.2 Dynamic Update Mechanisms and Protocols

Dynamic update mechanisms and protocols are popular in the researches of embedded operating systems. Brown and Sreenan [8] proposed a new model for software updating in wireless sensor networks (WSN). The proposed model integrates a consolidated set of WSN software update requirements in the form of meeting the criteria. Fig. 2 shows the interactions between the three key elements (i.e. generation, propagation and activation) of the software update model. The host system will broadcast new software’s information to the network. If a node needs this new software for upgrading, it will send a request message to the host system for downloading the new software into the program memory for executing. Besides the software updating, each node will send its profile to the host system. The profile may consist of the node’s status or software’s version information.
According to the profile, programmers can plan the new software for updating the software of deployed nodes.

SOS [9] is a dynamic operating system for sensor nodes. It is composed of a static kernel and dynamic loadable modules. The loadable modules can dynamically update the functionalities of SOS without rebooting the system. The kernel provides a set of system services that are accessible to the modules through a jump table stored in the program memory. Modules can then look up the jump table and find the entry addresses of these system services. Module dependency is not considered and SOS supports only module insertion and removal.

The Linux Loadable Kernel Modules (LKMs) [10] are a set of object files for extending the kernel at run time. They are typically used to support new hardware, file systems, device drivers, or new system calls. Kernel modules can be dynamically loaded into or unloaded from the kernel at run time. When a module is loaded into the kernel, its dependent modules will be loaded too.

Crossbow In-Network Programming (XNP) [11] is a network reprogramming mechanism implemented on TinyOS with single-hop WSN. It can update the kernel’s functionality. The base station will broadcast program image to the network for updating neighboring sensor nodes. When these nodes receive program image packets, they will store the whole program image in their internal flash memories. The nodes will reboot after finishing the update procedure.

MOAP [12] named Multi-hop Over-the-Air Programming is also a software update mechanism for sensor nodes. It uses the Publish-Subscribe architecture. When the host system generates the new software, it would broadcast the Publish message to the whole network. If a sensor node receiving this Publish message needs this new software, it would send the Subscribe message to the host system. Then the new software will be downloaded into the requesting sensor node. Afterward this sensor node will broadcast the Publish message in the neighborhood after it accomplishes the update procedure. If another sensor node also wants this new software, it will send the Subscribe message to that sensor node for updating itself. Besides, MOAP adopts Ripple dissemination protocol that selects an appropriate node to disseminate code while avoiding network flooding.

Deluge [13] which enhances MOAP is also a network reprogramming mechanism of TinyOS. It uses a multi-hop data dissemination protocol for reliably propagating large amount of update data throughout a wireless sensor network. Deluge optimizes transmission throughput by adjusting packet transmission rate. It divides the program image to be updated to page frames, and each page frame is further divided into several fixed-sized packets for transmission of program image. If a packet loss or an error occurs, only those pages in question need to be retransmitted.

Impala [14] is an event-based middleware layer system and updates only the appli-
A Platform for Supporting Dynamic Update and Resource Protection

Maté [15] used in WSN is a virtual machine that supports an interface for programmers to communicate with operating system. Programmers can use byte code to develop applications. The byte code designing can reduce the transmission of packets during the update procedure. Therefore, it can save the bandwidth of the wireless network. However, this mechanism can update only the application-layer software.

2.3 Software-based Resource Protection and Recovery

There are many researches using software-based approach to protect system resources. When an erroneous component is downloaded, it will corrupt the system. The Kernel Resource Protector (KRP) [16] implemented in Linux 2.4.22 kernel can guard kernel heap space against misuses by kernel modules. If KRP detects memory wasted by kernel modules, it will reclaim the lost memory space.

Harbor [17] is a software-based memory protection mechanism implemented in SOS operating system for sensor nodes. Harbor will ensure the correctness of memory accesses. It uses sandboxing [18] concept which is a software based fault isolation mechanism to restrict user module’s memory accesses. Chiou and Chang [19] implemented an error detection and recovery scheme which modifies the original Harbor mechanism in SOS. This scheme aims at decreasing the errors caused by improper memory accesses and enhancing the system availability by automatically recovering the system from errors.

Wang [20] modified the eCos for providing high availability in sensor network operating system. The wrapper function implemented between the kernel and the modules will record the information about which system resources are assigned to which modules. When the wrapper function detects a failure, it will trigger the recovery manager to handle this error. According to the frequency of failures incurred by this module, the recovery manager will automatically update the system with the module of the new version, re-initialize this failure module, or remove this erroneous module out of the system.

XFI [21] uses a binary rewriter and a verifier to guard system address spaces, but the system correctness depends on the verifier. The XFI verifier performs the static analysis to ensure sufficient guard for all execution paths.

3. System Design and Implementation

We have designed and implemented a platform for supporting remote dynamic update and resource protection in LyraOS. In our platform, not only the kernel components but also the user components can be updated. When the new components are downloaded into embedded client, the component manager will maintain them. In order to prevent
components from misuses of system resources and thus corrupting our system, our protection mechanism can detect errors in components and then recover system from errors.

3.1 System Overview

Our platform uses a client-server model as shown in Fig. 3. In order to dynamically upgrade our embedded clients remotely, we have implemented an update dissemination mechanism that uses a broadcast-request architecture. With a carefully designed linker script, new components are pre-linked to Executable and Linking Format (ELF) format and stored in the component download server. The component download server will periodically broadcast new components’ information to the network. Our embedded clients will continuously receive the broadcast messages and determine whether the new components are needed. If our embedded client needs this new component, it will send a request message to the component download server. When the component download server receives this request, it will transmit the new component to the requesting embedded client.

When new components are downloaded into embedded clients, they can register their exported methods to the component manager by invoking component manager’s exported APIs. The component manager is responsible for maintaining component information. Besides, for the sake of component dependency, downloaded components can also ask for downloading their dependent components into embedded client by invoking these APIs. After the completion of updating, embedded clients need not reboot at all.

At run time, the system resource protector will examine whether these downloaded components have misused system resources. If the resource protector finds any improper uses of system resources by the downloaded components, it will reclaim allocated resources and then recover the system from error conditions.

3.2 Dynamic Update Dissemination Mechanism

In the implementation of our dynamic update dissemination mechanism, we make use of the source codes of TFTP-Client and TFTP-Server [22], and then modify them for our purpose. Fig. 4 shows our system components. In the server-side implementation, the TFTP-Server is an application running on the component download server. It can receive
request messages from embedded clients, broadcast new components’ information to the network, and transmit components’ files to embedded clients. In the client-side implementation, the TFTP-Client is installed on the embedded client. It can send request message to the component download server and then receive new components from the component download server.

On the embedded client, the component download thread is implemented as a service thread and is responsible for downloading new components. This thread can continuously receive broadcast messages from network by invoking TFTP-Client’s exported functions. When the component download thread receives the broadcast message, it will verify whether the packet format is correct. If it needs this new component, it will send the request message to the component download server by invoking TFTP-Client’s exported functions. When the component download server has received the request message, it will start to transmit the new component into our embedded client.

Fig. 5 shows the server-side implementation in our platform. When a new component is created, the component download server will advertise its neighboring embedded clients by sending the broadcast message. The broadcast message contains the information instructing how to deal with this packet, the component name, and the component version. The format is described in the string “i/r-component name-component version”. In the format, “i” represents adding or updating a new component into embedded clients and “r” represents removing a component out of embedded clients. From the version information, embedded clients can decide whether they should add or update the new component.
To allow administrator to conveniently maintain embedded clients, we have also implemented a shell script on the component download server. Thus, administrator can manually update embedded clients’ components by issuing commands such as “inscom”, “rmcom”, and “lscom”. Besides adding new components, administrator can also remove useless components and update existing components to upgrade our embedded clients.

3.3 Component Manager

We have designed and implemented the component manager running on the embedded client to maintain all downloaded components. The component manager uses the component table and the method vector table to record the information about each component’s profile and its exported methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*<em>int CM::Add(char <em>add_com_name, unsigned int add_com_ver)</em></em></td>
<td>Adds a new component with the name <em>add_com_name</em> and version <em>add_com_ver</em> from the component download server and returns the added component’s ID.</td>
</tr>
<tr>
<td>*<em>int CM::Add_Dep_Com(char <em>dep_component)</em></em></td>
<td>Adds the information of the dependent component <em>dep_component</em> for the new component and returns component’s ID.</td>
</tr>
<tr>
<td><strong>int CM::GetPid(void)</strong></td>
<td>Looks up the component table, returns unused memory region, and then sets the PID register.</td>
</tr>
<tr>
<td>*<em>int CM::GetCV(char <em>componentname)</em></em></td>
<td>Gets the version of the requested component <em>componentname</em>.</td>
</tr>
<tr>
<td>*<em>int CM::GetCID(char <em>componentname)</em></em></td>
<td>Gets the component ID of the requested component <em>componentname</em>.</td>
</tr>
<tr>
<td><strong>int CM::GetMID(unsigned int p_cid, unsigned int mid)</strong></td>
<td>Gets the method ID of method <em>mid</em> and its parent <em>p_cid</em>.</td>
</tr>
<tr>
<td><strong>int CM::Invoke(unsigned int cid, unsigned int mid, unsigned long PTR arg)</strong></td>
<td>Invokes the method <em>mid</em> of the component <em>cid</em> with arguments <em>arg</em>.</td>
</tr>
<tr>
<td>*<em>void CM::Register(unsigned int mid, unsigned int cid, void <em>fptr)</em></em></td>
<td>Registers the method address <em>fptr</em> to the component manager for the method <em>mid</em> of the component <em>cid</em>.</td>
</tr>
<tr>
<td><strong>int CM::Remove(unsigned int cid)</strong></td>
<td>Removes the component <em>cid</em>.</td>
</tr>
<tr>
<td><strong>void CM::Update(unsigned int com_cid, unsigned int new_com_ver)</strong></td>
<td>Updates the component <em>com_cid</em> using the new component of version <em>new_com_ver</em>.</td>
</tr>
</tbody>
</table>

Table 1 lists component manager’s exported APIs. By invoking these APIs, our embedded client can maintain component profiles in the component table and the method vector table. When a component has been downloaded into our embedded client, the component manager will initialize it, record its profile, register its exported methods, and check its component dependency. If the component has some dependent components, it will also download them into our embedded client by invoking these APIs. Besides, components can communicate with each other by invoking these APIs.

In our platform, we can upgrade our embedded client’s functionalities by replacing the old version’s component with the new version’s component. The component manager will invalidate these old components, wait for the reference counts of their exported methods becoming zero, and then update or delete them.
A PLATFORM FOR SUPPORTING DYNAMIC UPDATE AND RESOURCE PROTECTION

Fig. 6 shows the component manager for maintaining downloaded components. At first, when the component download thread running on the embedded client receives the broadcast message that advertizes the existence of new components, it will invoke CM::GetCID() and CM::GetCV() to search the component table for the component with the specified component name and version. If the requested component does not exist in the embedded client, the component download thread will send request message to the component download server for downloading the desired component. Secondly, the component download thread will get a PID number through invoking CM::GetPID() and then set the PID register [3]. Each downloaded component can be loaded into different memory region by setting the PID register, thus each component has its own protection domain. Thirdly, the embedded client invokes CM::Add() to record new component’s information in the component table. Fourthly, the downloaded component can invoke CM::Add_Dep_com() to download its dependent components during its initialization. Fifthly, the downloaded component also will invoke CM::Register() to register its exported methods to the component manager during its initialization. Finally, when the component needs to call other components’ exported methods, it can invoke CM::Invoke() and then invoke CM::GetMID() to look up the method vector table for getting the requested method’s address.

After a component is downloaded into our embedded client, it can invoke component manager’s exported API to download its dependent components in our embedded client during the component initialization. We use ARM FCSE mechanism [3] and divide virtual memory into 128 regions. Components are initially downloaded into the PID 0 region and then moved to other unused memory regions by setting the PID register. The PID 1 region is used to store LyraOS kernel image. The remaining memory regions are used for downloaded components. The component download thread can get unused PID
regions in our embedded client for the downloaded components by invoking component manager’s exported API.

3.4 System Resource Protection

We have designed and implemented a system resource protector in our platform to protect our embedded clients against incautiously developed components. Currently, it can reclaim lost memory space, ensure the normal execution of critical sections, and prevent components from accessing null pointer. Basically, when downloaded components obtain system resources, the system resource protector will record the allocation of each system resource. After these components have released system resources, the corresponding allocation records will be removed by the system resource protector.

The system resource protector has several exported APIs. By invoking these APIs, our embedded client can register or unregister a component’s profile, and record or remove the information for the allocated resources. When a component is removed, the system resource protector will check the system status whether the component has misused system resources. For example, if its allocated resources have not been normally released, they will be released automatically by our system and the system status will be recovered through the invocation of these APIs.

3.4.1 Garbage collection

In our system, downloaded components can freely acquire additional memory spaces by invoking kernel’s exported function. An erroneous component might have allocated memory spaces but do not release them. Then after the component is removed out of our system or the thread finishes its execution, these allocated memory spaces will be lost and not available for the use of the kernel and other components. Thus, incautiously developed components may have the danger to exhaust system memory.

We have implemented the system resource protector to guard our embedded client’s memory space. When a component is removed out of our embedded client, the system resource protector will reclaim the lost memory spaces used by this component.

Fig. 7 shows our garbage collection mechanism. If a new component is downloaded into our embedded client, the component download thread will invoke RD::record_component() to register the component’s profile (step 1). Later when the downloaded component invokes malloc() to acquire additional memory space, our embedded client will record this allocation into a hash table by invoking RD::record_memory() (steps 2 and 3). In order to decrease system overhead, this allocation information is not immediately removed when the component invokes free() to release allocated memory space (step 4). Instead this allocation information is marked invalid (step 5). Later, the component download thread can invoke RD::remove_memory() to clear all invalid allocation information (step 6). When the component is removed out of our embedded client, the component download thread will invoke RD::remove_component() to remove the component’s profile, and then the system resource protector will examine whether this component has allocated memory spaces but not released them. The system resource protector will reclaim these lost memory spaces used by this component (step 7).
3.4.2 Critical section

For ensuring the normal execution of a critical section in our platform, a thread is required to invoke Lock::Acquire() function to disable interrupts when it wants to enter a critical section. Meanwhile, it should invoke Lock::Release() function to restore interrupts after it leaves the critical section. If a thread does not normally invoke Lock::Release() function when it exists the critical section, a deadlock situation might occur. In other words, interrupts will be disabled and the following interrupts will be ignored.

Our system resource protector is also designed to ensure the normal execution of critical sections, by setting some check points to detect whether interrupts have been normally disabled. When a thread wants to enter a critical section, it will disable interrupts by invoking Lock::Acquire(). Then the system resource protector will record its access time. If the thread does not invoke Lock::Release() to restore interrupts after a certain period of time since it has disabled interrupts, the system resource protector will release this lock and enable interrupts, and then removes this erroneous component out of our embedded client.

Fig. 8 shows how our embedded client ensures the normal execution of a critical section. When a component is downloaded into our embedded client, the component download thread will register the component’s profile to the resource protector (step 1). When the component invokes Lock::Acquire() to lock the critical section and disable interrupts, RP::record_lock() will be invoked to record the locked resource’s information in the lock array table (steps 2 and 4). Before LyraOS stores the locked resource’s information, it will invoke get_clock() to acquire the current system time (step 3). When the component leaves the critical section, it will invoke Lock::Release() to restore interrupts (step 5). Similarly, LyraOS will invoke RP::remove_lock() to clear the corresponding recorded information in the lock array table (step 6). The component download thread
then will invoke RP::check_lock() to check the lock array table (step 7). After the system resource protector invokes get_clock() to get the current system time, it will compare the locked resource’s access time. It will release locked resources that have been locked for a certain long time (step 8). When the component is removed, the system resource protector will check the lock array table and then release resources locked by this component (step 9).

### 3.4.3 Null pointer access

Components can allocate system memory spaces by invoking kernel’s exported function. When a component needs to acquire a larger memory space that exceeds system memory size, the kernel will return a null pointer. If the component does not check this returned pointer, it will access the memory space located at the address zero. However, in our design, a downloaded component is initially downloaded into the virtual memory located at PID 0 region [3] and then is relocated into one unused virtual memory region by the setting of the PID register. In other words, the memory addresses that a component actually accesses are the modified virtual addresses located at the component’s own 32MB virtual memory region. Thus, the component has the potential risk to overwrite its code section when accessing the null pointer.

In our work, we have modified the memory access permission in MMU’s section table [3] to avoid null pointer access. When an erroneous component does not check the returned pointer of memory allocation function and directly access the null address, the data abort exception will be triggered. Then our data abort exception handler’s service routine will remove this erroneous component out of embedded client.
4. PERFORMANCE EVALUATION

This section presents the performance evaluation of our platform for supporting remote dynamic update and resource protection mechanisms in LyraOS embedded operating system. Section 4.1 describes our experimental environment including hardware and software facilities. Section 4.2 shows program code analysis including LyraOS kernel image size and the object code sizes of components. Section 4.3 shows our experimental results. Finally, section 4.4 presents extra system overhead of our platform for supporting remote dynamic update and system resource protection.

4.1 Experimental Environment

Fig. 9 shows our experimental environment. The experimental hardware consists of a component download server and an embedded client connected through a 100 Mbits/sec Ethernet. The component download server is a Pentium IV 3.4 GHz machine with 1GB RAM, running the Linux kernel 2.6.18. The embedded client is an ARM Integrator/CP920T development board [3] with 128MB RAM, running LyraOS v. 2.1.13.

![Diagram](image)

Fig. 9. Experimental environment.

| Table 2. Sizes of LyraOS kernel image. |
|-----------------|-----------------|
| **Items**       | **Size**        |
| Original LyraOS kernel image | 35,752 bytes    |
| Modified LyraOS kernel image  | 39,344 bytes    |

4.2 Program Code Analysis

In order to implement a platform for supporting remote dynamic update and resource protection mechanisms, we not only modify and extend the functionalities of the embedded client but also add functionalities in the original component download server. Table 2 shows the kernel image sizes of LyraOS for the embedded client. We compare the sizes of the original and the modified LyraOS kernel images. For our work of enhancement, the size of LyraOS kernel image is increased by only 3.5 Kbytes.
4.3 Experimental Results

4.3.1 Garbage collection

In our platform, the lost memory spaces will be reclaimed after an erroneous component is removed out of the protected embedded client. We have designed a test component \(i.e.\) memory_test-1 that will acquire 12MB memory space but does not de-allocate it when it is not needed. Fig. 10 shows the screenshot for this test component that is downloaded into our protected embedded client. This test component later is removed out of the embedded client after the component has successfully allocated memory space. According to the result, the protected embedded client can successfully reclaim this unused lost memory space after this erroneous component has been removed.

![Screenshot of garbage collection](image)

Fig. 10. The screenshot for testing garbage collection.
4.3.2 Critical section

To prevent more than one thread from simultaneously accessing a shared resource, in our platform, interrupts will be disabled before a thread enters a critical section and restored after the thread exits the critical section. The system resource protector will ensure interrupts to be normally disabled and restored by setting some check points to examine whether interrupts have been normally disabled and enabled by a component. The erroneous component will be further removed out of the protected embedded client.

In this experiment, we have designed two test components (i.e. lock_release-1 and lock_test-1) to verify whether our platform can correctly disable and restore interrupts. The lock_release-1 component’s exported method will execute in a critical section and acquire a lock to prevent more than one thread from simultaneously accessing a shared resource. However, this erroneous component will incur the deadlock situation since its exported method has disabled interrupts before entering the critical section but does not enable interrupts when it exits the critical section. Besides, the lock_test-1 component depends on the lock_release-1 component since it invokes the lock_release-1 component’s exported method to complete its task.

Fig. 11 shows the screenshot for the lock_test-1 component that invokes the exported method of the lock_release-1 component. After the lock_test-1 component has invoked the lock_release-1 component’s exported method, the protected embedded client will detect that interrupts have been previously disabled but are not correctly restored. Our protected embedded client then will restore interrupts and remove this erroneous lock_release-1 component out of our embedded client.

![Fig. 11. The screenshot for testing the execution of a critical section.](image-url)
4.3.3 Null pointer access

We have designed a test component (*i.e.* null_access-1) to verify whether our platform can prevent from accessing null address. Fig. 12 shows the screenshot for the null_access-1 component that is downloaded into our protected embedded client. This downloaded component tries to allocate 17MB memory space. Because our embedded client does not have enough memory space, so the memory allocation function will return a null pointer. This erroneous component does not check this returned null pointer and then directly accesses the returned null memory address. The result shows that the data abort exception is incurred and this erroneous component is removed out of our protected embedded client by the resource protector.

![Fig. 12. The screenshot for testing null pointer access.](image)

4.4 System Overhead

To evaluate the extra system overhead incurred by our implementation, we have designed three test components (*i.e.* calculate-1, null_access-1, and not_access-1) to measure average execute time and system overhead of the original LyraOS and the modified LyraOS. The calculate-1 component is used to measure the extra overhead for garbage collection and ensuring the normal execution of a critical section. This component will
be respectively downloaded into the original and the modified LyraOS. The execution is performed 30 times and the average time of execution is then calculated. The null_access-1 component is developed to measure the elapsed time of handling null pointer access. Besides, we download the not_access-1 component into our embedded client and then remove it for measuring the system overhead of component downloading and removing.

Table 3 shows that our implementation incurs only little overhead. For example, the extra overhead of garbage collection mechanism for memory allocation and free is less than 5 microseconds. To ensure the normal execution of a critical section, the extra overhead is less than 11 microseconds. The extra overhead for downloading a component into embedded client is 66 microseconds and is 190 microseconds for removing a component out of embedded client.

<table>
<thead>
<tr>
<th>Items</th>
<th>Original LyraOS (µs)</th>
<th>Modified LyraOS (µs)</th>
<th>Overhead (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>malloc()</td>
<td>15.76</td>
<td>18.93</td>
<td>3.17</td>
</tr>
<tr>
<td>free()</td>
<td>9.26</td>
<td>10.93</td>
<td>1.67</td>
</tr>
<tr>
<td>Lock::Acquire()</td>
<td>3.36</td>
<td>4.86</td>
<td>1.5</td>
</tr>
<tr>
<td>Lock::Release()</td>
<td>3.16</td>
<td>4.26</td>
<td>1.1</td>
</tr>
<tr>
<td>RP::check_lock()</td>
<td>–</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Handle Null Pointer Access</td>
<td>–</td>
<td>13,915</td>
<td>13,915</td>
</tr>
<tr>
<td>Download a Component</td>
<td>136,536</td>
<td>136,602</td>
<td>66</td>
</tr>
<tr>
<td>Remove a Component</td>
<td>9,690</td>
<td>9,880</td>
<td>190</td>
</tr>
</tbody>
</table>

5. CONCLUSION

This paper describes our experience in the design and implementation of a platform that can not only dynamically upgrade the functionalities of embedded clients remotely but also protect system resources of embedded clients. Downloaded components can add new functionalities into the system, can be removed when they are not needed at all, or can update the existent components. Furthermore, downloaded components can also ask for downloading their dependent components by invoking well-designed APIs. With our resource protection mechanism, protected embedded clients can reclaim lost memory space, ensure the normal execution of critical sections, and prevent null pointer access. Our mechanisms have been practically implemented in LyraOS embedded operating system to demonstrate the effectiveness of the design. The experimental results demonstrate that our resource protection mechanism can effectively protect system resources of embedded clients with only little overhead.

Although the current implementation of our resource protection mechanism protects only some system resources, we will extend our mechanism to protect more types of resources. Currently, our dynamic update dissemination mechanism can upgrade embedded clients’ functionalities without rebooting the whole system. To further reduce the update overhead, we will address the issues on the diff-based dynamic component update and reconfiguration mechanism. Besides, a better system recovery scheme should be developed if the task of downloading dependent components is failed.
REFERENCES


17. R. Kumar, E. Kohler, and M. Srivastava, “Harbor: Software based memory protection for sensor nodes,” in *Proceedings of the 6th International Conference on Infor-
A PLATFORM FOR SUPPORTING DYNAMIC UPDATE AND RESOURCE PROTECTION


Mei-Ling Chiang (姜美玲) received her B.S. degree in Management Information Science from National Cheng Chi University, Taipei, Taiwan, in 1989. She received the M.S. degree in 1993 and her Ph.D. degree in 1999 in Computer and Information Science from National Chiao Tung University, Hsinchu, Taiwan. Now she is an Associate Professor in the Department of Information Management at National Chi-Nan University, Puli, Taiwan. Her current research interests include operating systems, embedded systems, and clustered systems.

Hsiang-Yu Hsu (徐翔宇) received his M.S. degree in Information Management from National Chi-Nan University, Puli, Taiwan, in 2010. He is currently working as a software engineer in WinMate Communication Inc., Taiwan. His research interests include embedded systems, operating systems, and computer architectures.