Commons: resource sharing and protection in mobile appliance OS

Tetsuya Saito
Graduate School of Media and Governance
Keio University
5322 Endoh, Fujisawa, Kanagawa 252, Japan
saimune@tom.sfc.keio.ac.jp

Tatsuya Hagino
Faculty of Environmental Information
Keio University
5322 Endoh, Fujisawa, Kanagawa 252, Japan
hagino@tom.sfc.keio.ac.jp

1 Introduction

In this paper, we propose a new operating system abstraction, called a *common*, for mobile appliances. We can use a *common* to prevent the excessive use of computational resources on appliances.

Mobile appliances will have networked servers such as a http server. We need a new resource abstraction for mobile appliance OS to protect relatively precious and limited resources against service requests from the Internet.

The *common* abstraction can restrict accesses of machine resources and maintain stability on appliances against unknown users on the Internet. We describe two applications of the *common*: resource allocation for a WWW server and restriction on a bad influence of DoS attacks.

In the next section, we describe system requirements for mobile appliance OS and point out its needs. In Section 3, we describe the abstraction of a *common*. Section 4 describe the applications of *common*s. In Section 5 we describe current status of our project. Section 6 discusses related work and Section 7. concludes.

2 System requirements for mobile appliance OS

A mobile appliance OS should fullfill the following criteria.

- open access networked servers support
- efficient computational resources management

Mobile appliances such as mobile phones, PDAs, and portable game machines will have equivalent machine power to current laptop computers in near future. Because of widespread use of WWW-related applications, the appliances will support a http server. However, supporting networked servers on mobile appliances introduces a serious problem into the appliances' OS. Many service requests from the Internet are the cause of excessive use of machine resources such as CPU execution time. For example, if an appliance processes many http requests from network and the appliance's performance has been down. The excessive use of appliance's resources results in the level of availability degrading.

Various applications will be able to run on mobile appliances. A http server on appliances need to be a platform for many services such as a telephone, a multi-media mail, an MP3 player, and a personal data management system. Although appliances have relatively less machine resources, variaous applications such as multi-media applications and servers will run on appliances. An appliance OS needs efficient computational resources management.

On the other hand, conventional OS such as UNIX are designed to provide computational resources with its aplication programs in the way of high performance, high throughput, and unlimited consumption. As we described, this assumptions and criteria will need to change.

3 A new model for resource management

In this section, we propose a new model, for describing various types of resource usage in an operating system. We introduce a new abstraction, called a *common*, which represents a policy based on the way of using resources. The *common* abstraction extends the filesystem quota mechanism in UNIX for all applications and OS subsystems.

A *common* has two parameters: inter-*common*'s allocation policy and an upper limit variable. We describe the inter-*common*'s allocation policy later. A *common* has a quota policy. Each policy is hard limit, soft limit, and none. These quota's policies represents type of resource usage. The usage types are characterized by ownership and a sharing attribute. Each *common* is as follows.

We define the *public common* for sharing resources in multiple owners. Examples of resources in *public common* are shared libraries and file buffer caches read by multiple threads. The *public common* has a soft limit policy. The *kernel common* is for kernel processes and kernel's text and data areas. Because of kernel resources are allocated at boot time, this *kernel common*'s policy is none. The *application common* is for not shared resources in an application such as its data and stack areas. The *application common*'s policy is none too. Examples of resources in the *anonymous common* are interrupt handlers and a interrupt-driven protocol stack. The resources in *anonymous common* do not have an explicit owner and are not shared with other threads. The *anonymous common* has hard limit policy. Table1 illustrates each common.

	ownership	
	explicit	implicit
share	public	kernel
unshare	application	anonymous

Table 1: An attribute and name of each common

A user can set all *common*'s parameters. The user can assign an OS subsystem and an application to a *common* explicitly. An application's default *common* is *application common*. Default *common* in OS subsystems is *kernel common*.

We have allocation policies for inter-*commons*. We describe an example with a prioritized allocation policy. This example excludes *kernel common*. This reason is that the *kernel common*'s resources are fixed and do not changed. We can assign a priority to each *common*. An example is below. In this example, the *application common*'s priority is higher than the *public common*'s priority.

$$application > public > anonymous$$

When a thread in *public common* requests resources and free resources are empty, the thread can use resources in the *anonymous common* which has low priority.

The advantages of the *commons* abstraction are simplicity and applicable to OS subsystems. In a resource management mechanism on mobile appliances, simplicity is important. Because the appliances' machine resources are relatively small. A large and complex resource management system causes applications' performance overhead. Application of *commons* to OS subsystems is important too. A firewall mechanism has an effect on preventing denial-of-service (DoS) attacks. However, additional costs of using the firewall on mobile appliances will be relatively larger than that of desktop computers. We want to decrease a cause of performance overhead. On the other hand, *commons* are built in the kernel. We can use input traffic shaping for wasteful packets by only lowering upper limit variable with *commons*.

4 Applications of Commons

In this section, we describe two applications of commons, resource allocation for a WWW server and restriction on a bad influence of DoS attacks.

4.1 Resource allocation for a WWW server

A WWW server on mobile appliances has many services such as a telephone, a multi-media mail, a wallet for electronic commerce, and a personal data manager. This WWW server is a main server on the appliances.

A problem of this WWW server is to protect appliance's machine resources from the Internet requests. An appliance's owner and many unknown users can use this WWW server. They share the resources such as CPU time and memory. If many unknown people see a home page on your appliance's WWW server, many http requests from a network consume much CPU time. When you want to use your browser, its performance may be slow.

To set low scheduling priority or allocate less resources for the WWW server do not avoid this problem. The cause of this problem which is to process many requests from network remains. Your browser's performance is down with the server's priority and allocated resources.

We can use commons to address the problem of unlimited resource accesses. For example, We set a WWW browser on your appliance in the application common, the network subsystem in the anonymous common, and the WWW server and the filesystem in the public common. An advantage of this allocation policy can limit consumed resources against service requests through a network with anonymous common. Unknown users' requests do not interfere with processing owner's requests, and moreover, the owner and unknown users can share resources such as file buffer caches which file transfer requests from a network. Figure1 illustrates this example.

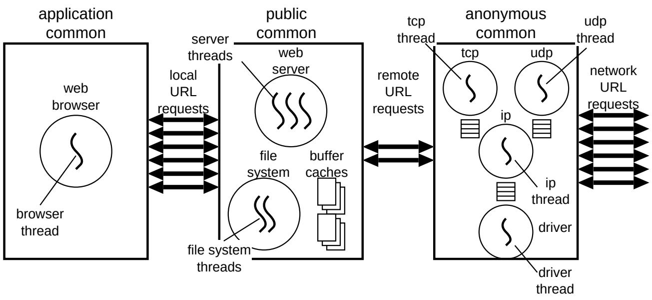

Figure 1: An example showing how a user can allocate commons to a WWW browser, a WWW server, the filesystem and the network subsystem.

In general, the commons can apply to application processes and OS subsystems. A user sets applications and OS subsystems in different commons based on the way of being used resources.

4.2 Restriction on a bad influence of DoS attacks

Bad influences of DoS attacks for mobile appliances are serious. As we describe above, many users of good will can cause performance degrading. Malicious DoS attacks can cause less available network bandwidth. DoS attacks have various types such as the SYN flood attack, the SMURF attack and the ICMP echo reply (ping) attack. We will describe to prevent the ping attack as an example.

To amortize those bad influences, we can use commons. We set commons in the network subsystem a.k.a the TCP/IP protocol stack. A network device driver, IP threads in the IP layer, and the IP queue are in the public common. TCP threads, UDP threads, and their queues are in the application common. To process ICMP packets, a ICMP thread and its queue are in the anonymous common. Each thread processes an incoming network packet and places it on the destination queue. Figure 2 illustrates this example.

In case of detecting the ping attack, allocated resources in anonymous commons are reduced. For example, the ICMP thread's priority is lowered and the length of the ICMP's queue is shortened. When ICMP packets filled with the ICMP's queue, successive received ICMP packets are dropped. The wasteful packet processing cost is saved.

The commons can restrict bad influences of DoS attacks to allocated resources. In other words, these packets are shaped traffic by allocated resources in commons.

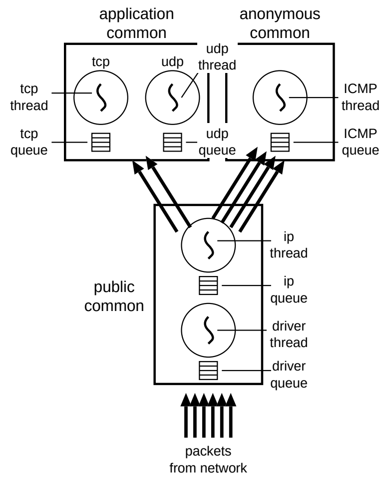

Figure 2: An example of allocating commons to the TCP/IP network protocol stack.

5 Current Status

We are implementing a Java operating system which has the *commons* resource allocation mechanism. OS service classes which manage resouces in our JavaOS have a interface for *commons*. We extend our Java virtual machine which is running on native hardware and has PCMCIA disk and network device drivers. Currently our JavaOS has a filesystem, a process manager and UDP/IP protocol stack.

We describe our JavaOS with the process-oriented model. All interrupts such as disk interrupts and network interrupts are hidden in our extended JavaVM. Therefore, device drivers are implemented by C. However, device handlers which read and write data from the device are Java threads.

6 Related Work

Banga et al.[1] proposed the resource container abstraction for fine-grain resource management in server systems. An application which uses the resource container can control machine resources consumed by the application's activity in OS. However, it requires knowledge of detailed application behavior and OS kernel. We propose a simple resource allocation mechanism based on the way of using resources.

Verghese et al.[3] introduced the performance isolation model and the Software Performance Unit(SPU) abstraction in shared-memory multiprocessors. A SPU has processes which access to the resource of the SPU, allocated computational resources of the multiprocessor, and a sharing policy. The SPU guarantees isolation. from other SPUs. In contract to the SPU, we aim at sharing and protecting mobile appliances' resources against service requests from the Internet.

7 Conclusion

This paper proposes a operating system quota abstraction that allows th user to protect resources in OS subsystems. Such quota abstraction is necessary to support flexible resource allocation for a http server and restriction on bad influences of DoS attacks on mobile appliances.

Unlike a conventional file system quota mechanism, *common*s take advantage of quota mechanism in all OS subsystems. *common*s have simplicity and application to all resource management in OS.

We are implementing the *common* abstraction in our JavaOS.

References

[1] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. 3rd Symp. on Operating Systems Design and Implementation*, Feb. 1 999.

[2] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proc. 17th Symp. on Operating System Principles*, Dec. 1999.

[3] B. Verghese, A. Gupta, and M. Rosenblum. Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors. In *Proc. 8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.