Specifying Runtime Environments and Functionalities of Downloadable Components under the Sandbox Model

Haruhiko Kaiya
Shinshu University
4-17-1 Wakasato
Nagano City, 380-8553, JAPAN
kaiya@cs.shinshu-u.ac.jp

Kenji Kaijiri
Shinshu University
4-17-1 Wakasato
Nagano City, 380-8553, JAPAN
kaijiri@cs.shinshu-u.ac.jp

Abstract

In this paper, we propose a specification of both runtime environments and software components which can be loaded not only from your local system but also from the other systems over the computer network. Because components from the other system are not always enough reliable or safe to act freely in your own system, you should limit their activities to a certain context. Such assumption is based on the sandbox security model. Because such components are largely influenced by the runtime environments, users sometimes lose sight of the abilities and limitations of such components. Therefore, they fail to reuse the components in the right way. We provide a way to specify such properties, so that component users can precisely understand the abilities and limitations.

1. Introduction

Recently, we can use software components which can be downloaded from the other machines over the network, and can be linked dynamically during runtime. This kind of software components can be categorized into *mobile code* [10], and it enables our system to change itself dynamically in runtime. Even if a system can act dynamically and autonomously with mobile codes, we should rule its behavior. Therefore, we should also specify the behavior of mobile codes linked into the system.

Normally for specifying the properties of software components, documents in natural languages are used. Unfortunately, such documents tend to be long, incomplete, redundant or vague, component users sometimes can not use such components adequately. For example, the specification of RMI[8] consists of about 90 pages of documents and is hard to read it completely. But small manuals in many books are too simple to understand the components. One of the suitable tool for specification is formal method. Though writing a formal specification of components looks like expensive, we can retrieve the costs for using the components and the specification repeatedly [5].

We can not simply use the way of traditional functional specification, e.g. pre/post conditions and invariants, for mobile codes or components, because such codes are not always trusted. Moreover, when you want to run a system with mobile codes, e.g RMI or Jini, you should setup several kinds of servers, connect the networks suitably, deploy several pieces of codes on the suitable locations, and so on. In other words, you should prepare its runtime environment. As a result, a mobile component can not be fully specified without documents of such environment and its security model. One of the famous security model is called *sandbox model*, where local code is trusted to have enough access to vital system resources while downloaded code is not trusted and can access only the limited resources[9]. This model is used in Java1. There are another security models [7], and Java2 (JDK1.2 and later) has hybrid security model.

In this paper, we propose a way of specification for mobile components under the sandbox security model, which are not autonomous but downloaded by the other components or systems. We call such kind of mobile components as *downloadable components* in this paper. With our specification, users of downloadable components will be able to use the components and their environment adequately.

In section2, we introduce the properties and problems of downloadable components. In section3, we summarize what should be specified in addition for such properties, and we give examples in section4 to show the advantages of the specification. In the last section, we summarize our results and discuss the future works.

Proceedings of the International Symposium on Principles of Software Evolution (ISPSE'00)
0-7695-0906-1/00 $ 17.00 © 2000 IEEE

2. Properties and Problems of Downloadable Components

In this section, we introduce examples, underlying mechanism and problems of downloadable components. Because examples in this section are all about Java system, we regard the term '*class*' in the same light as the term 'component'.

2.1. What is Downloadable Component?

The most famous system using downloadable components is an Applet. Because classes used by an Applet are not enough reliable or safe, the classes are not allowed completely to operate system resources, e.g. file system or network connections, among the browser side system.

RMI (Remote Method Invocation)[8] is regarded as object-oriented RPC for Java, and its stub and skeleton are also downloadable components. In RMI system, in contrast with an Applet, programmers should explicitly manage the limitation by the `SecurityManager` in Java. Therefore, you can use the downloadable components without limitation, if you want.

2.2. Sandbox Security Model

The behaviors of an Applet and a RMI follow the sandbox security model. The essence of the model is that local code is trusted to have full access to vital system resources (such as the file system) while downloaded remote code is not trusted and can access only limited resources provided inside the sandbox [9]. This model is widely used by Java1 language system.

Because the sandbox model provides only two kind of spaces for codes, a trusted space for local code and another space in the sandbox for remote code, we can not flexibly represent many kinds of security limitation in one JVM. In Java2 system, we are free from this limitation and can represent many kinds of security limitation.

However, the sandbox model is still basic concept of downloadable components because the extension in Java2 can be regarded as the multiple-sandboxes model. Moreover, sandbox model is used as default model still in Java2 when the programmer do not explicitly specify a security policy [2]. Therefore, we only focus on the sandbox model of Java1 in this paper.

2.3. Class Loaders

Components of Java are running on the Java Virtual Machine (JVM). We focus on the functionality of the components during their runtime in this paper. JVM has two different kind of class loaders, *system class loader* and *user-defined class loaders* [6]. System class loader is used for local codes, and user-defined class loaders are used for remote codes. Figure1 is an example of an user-defined class loader.

```
1  import java.net.*;
2  import java.io.*;
3
4  public class DLoader extends ClassLoader{
5      // several definitions are omitted .....
6
7      protected Class loadClass(String name, boolean res)
8      throws ClassNotFoundException{
9          Class clazz = null;
10
11         try{ return findSystemClass(name); }
12         catch(ClassNotFoundException e1){}
13         catch(NoClassDefFoundError e2){}
14
15         clazz=findLoadedClass(name);
16         if(clazz != null){ return clazz; }
17
18         clazz=findClass(name);
19         if(clazz == null ){
20           throw new ClassNotFoundException(name);
21         }
22         return clazz;
23     }
24
25     private synchronized Class findClass(String name){
26         // finding byte code from the network resource
27         // and define class.
28         // .....
29               return defineClass(name, data, 0, total);
30         // .....
31     }
32
33     // several definitions are omitted .....
34
35     public static void main(String args[]){
36         try{
37             DLoader loader=new DLoader(new URL(args[0]));
38             Class cc=loader.loadClass(args[1]);
39             Runnable cmd=(Runnable)cc.newInstance();
40             cmd.run();
41         }catch(Throwable e){
42             e.printStackTrace();
43         }
44     }
45 }
```

Figure 1. Simple User-defined Class Loader

Security policy of a JVM runtime is decided in the following steps.

1. Decide whether your class loader takes account of the `SecurityManager` or not. For example, the loader in Figure1 does not, but loaders of RMI and an Applet does.

2. Decide each security policies listed in class `SecurityManager`. The manager class has about 30 numbers of check lists for limiting the operations for the system resources, e.g. file systems and network connections[8]. You can implement your own manager as a subclass of `SecurityManager`. For example, the manager for RMI, i.e. `RMISecurityManager` disallows most of all system operations.

Then, the decision is reflected to the component functionality as follows.

- If a class loader is designed to require its `SecurityManager` but the manager is not given, the class loader itself can not run. As a result, the classes loaded by the loader also can not run.

- If a loader is not designed to require its `SecurityManager`, the loader itself and the classes loaded by it can run.

- Methods in Java class libraries for operating the system resources are designed to refer the related check lists in the `SecurityManager` of their system, if the manager is defined. Therefore, the activities of each method is limited by the manager.

  As a result, if an user-defined class loader does not require its manager but a manager is defined, its limitation is applied to the system operations.

Though the security model is extended and generalized in Java2, class loading mechanism above still plays a large role in Java2.

2.4. The Problems of Downloadable Components

As mentioned above, Java has flexible but complex mechanism for loading its components. As a result, without deep understanding of class loading and security system, component users tend to lose sight of the abilities and limitations of such components, especially those downloaded from the other system over the network. Here we summarize the problems.

2.4.1 Loader Selection Problem

As explained above, a component of Java changes its behaviors along with its class loader. In other words, a class loader limits the behaviors of classes loaded by the loader. Therefore, component users should know the specification of components with their loaders. But it is not so easy because

- application programmers normally do not care about the class loaders,
- and the *defining loader* is dynamically decided according to both
  - the logic of the *initiating loader*.
  - and the deployment of components

If a component `C` is the result of `L.loadClass()`, a loader `L` is called initiating loader of C. If a component `C` is the result of `L.definClass()`, a loader `L` is called defining loader of C [4].

The logic of the initiating loader specifies the order for selecting defining loader. For example, `DLoader` in Figure1 uses the following order to select its defining loader,

1. system class loader (line 11-13),
2. cache of this loader (line 15-16),
3. `defineClass` method of this class via the method `findClass` (line 18).

As a result, components in the local system, which will be loaded by the system class loader, have the priority in `DLoader`. Loaders of RMI and an Applet have similar logic of selection. We can also develop a loader which gives a priority to the components loaded via the network.

2.4.2 Deployment Problem

The deployment also affects the selection of the defining loader. For example, if you use `RMIClassLoader` for loading the components over the network and the same components is also deployed in your local system, `RMIClassLoader` is never used as their defining loader but system class loader is used. As a result, the security manager gives no effect to the loaded components. Therefore component user should know whether a security manager is active or not, by checking the deployment of components.

3. Additional Specification for such Properties

For describing a specification of a downloadable component concretely, we represent the states related to the component and the functions for changing the states. We describe the following states.

- Deployment map of files or byte streams which can generate runtime components. This map is shared by all JVM runtimes.
- Access flags of each system resources from a component in the sandbox. This flags are defined in each JVM runtime.
- Search path of files or byte streams keeping component's code. This path is defined in each class loader.
- Identifier of runtime where each component is activated.
- Relationships between a runtime component and its sources.
- Component specific attributes.

We describe the following functions.

- Set or modify the flags, the map, the path and the relationships.
- Pre and post conditions for each method of the component. In contract to the traditional specification, the conditions are defined not only the component specific attributes but also the flags, the map, the path and the relationships above.

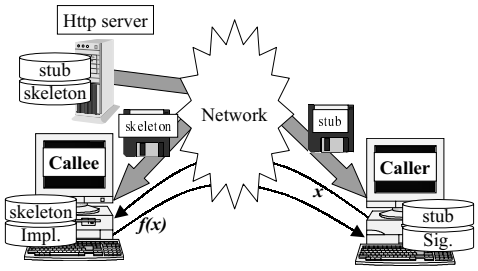

Figure 2. RMI with stub and skeleton both in the local system and over the network

4. Examples

In this section, we introduce examples to show how the specification in section3 is useful for component users to understand the properties of downloadable components. We use Z notation for our specification in this paper because it is widely known in the software engineering field, and because it is fit for object-oriented mechanism.

4.1. Stub with Cracking Code: Counter Example

4.1.1 Story

Even though we know an RMI call contains cracking codes which will steal our password file, we should use the RMI call for our system. Therefore we use `SecurityManager` which limits such stealing so that we can guard our system against the cracking. Moreover, we carefully deploy current version of stub codes so as to stop the progress of cracking in the codes. Here we check the safety of this situation as shown in Figure2. Note that we only take account of access flags in Section3 in the following specification.

4.1.2 Specification

First, we specify the system resources and their security limitation in *SysRes*. The security limitation can change only at once by `SetSecurityManager` method in Java, we model this method in *SetLimit*.

SysRes

$$res : R \pfun Bool;\ limit : \mathbb{P}\, R$$

$$limit \subseteq \mathrm{dom}\, res;\ \forall x : limit \bullet (x, \mathrm{false}) \in res$$

SetLimit

$$\Delta SysRes;\ l? : \mathbb{P}\, R$$

$$limit \neq \emptyset \Rightarrow l? = limit'$$

R is basic type which represents a set of resources. Schema *SysRes* represents a security level of this machine.

A method with cracking codes itself and the cracking codes can represented as follows;

Func

$$x?, y! : \mathbb{Z}$$

$$y! = f(x?)$$

Crack

$$pas! : R;\ \Xi SysRes$$

$$(pas!, \mathrm{true}) \in res$$

The method will be observed from the component users as follows;

$$F \mathrel{\widehat=} (Crack \fatsemi Func \land \Xi SysRes) \setminus \{pas!\}$$

4.1.3 Inference

Then we check the following schema,

$$SetLimt \fatsemi Crack \fatsemi (Func \land \Xi SysRes) \mid pas! \in l?$$

This schema tells that cracking is established even if corresponding operation is protected by the security manager.

This schema is inconsistent because both $(pas!, \mathrm{true}) \in res$ and $(pas!, \mathrm{false}) \in res$ are satisfied at the same time even if *res* is function. Therefore, we can conclude that the cracking is never established and our protection is enough safe.

4.1.4 Discussion

The specification above does not mention that the policy of security checking is changing along with the birthplace of components. As a result, inference above does not reflect the fact. Unfortunately, in fact, the cracking can be established in the story above. We will resolve this problem in the next.

4.2. Stub with Cracking Code: Resolved

4.2.1 Specification

We introduce two additional basic types, *Loc* for location of byte codes, and *ByteCode* for realizing a class and an instance in run time. Then we define the deployment of byte codes over the network (the deployment map in Section3) as follows;

$$\mid deploy : Loc \pfun \mathbb{P}\, ByteCode$$

We extend *SysRes* schema with the location where the components are running (identifier of runtime in Section3), and introduce *Class* schema for representing the birthplace, byte code and the logic for selecting the birthplace of the class (the search path and relationship in Section3).

SysRes

$res : R \pfun Bool;\ limit : \mathbb{P}\, R;\ here : Loc$

$limit \subseteq \mathrm{dom}\, res$
$\forall x : limit \bullet (x, \mathrm{false}) \in res$
$here \in \mathrm{dom}\, deploy$

Class

$birth : Loc;\ byte : ByteCode$
$lslctr : \mathrm{seq}\, Loc$

$birth \in \mathrm{ran}\, lslctr$
$birth \in \mathrm{dom}\, deploy$

Then the attribute *birth* in schema *Class* is defined the following schema.

SetLoader

$sl?;\ \mathrm{seq}\, Loc;\ \Delta Class$

$lslctr' = sl?$
$\forall x, y : \mathbb{N} \bullet byte \in deploy\, lslctr'\, x \land$
$\quad x \in \mathrm{dom}\, lslctr' \land lslctr'\, y = birth' \Rightarrow y \leq x$

The second and third lines of predicate part (the search path in Section3) are slightly complex, but it simply says that the first location which involves *byte* in *lslctr* is its *birth*.

The schema *Crack* is modified as follows for representing the effect of birthplace.

$$Crack \mathrel{\widehat{=}} [pas! : R;\ \Xi SysRes;\ \Xi Class \mid here \neq birth \Rightarrow (pas!, \mathrm{true}) \in res]$$

4.2.2 Inference

The following expression becomes consistent,

$$SetLimit \fatsemi (SetLoader \land \Xi SysRes \fatsemi Crack \fatsemi Func \land \Xi Class \land \Xi SysRes) \setminus Class \mid pas! \in l? \land sl? = \langle here, there \rangle$$

under the situation of

$$deploy = \{(here, \{byte, \cdots\}), (there, \{byte, \cdots\}) \cdots\}.$$

In Figure2, *here* corresponds to 'Call', *there* to 'Http server', and *byte* to 'stub'. As a result, cracking can be established even if corresponding operation is protected by the security manager under this situation. So if you want to stop the cracking, you should change *sl*? or *deploy*.

5. Discussion

In this paper, we discuss and propose how to specify the environments and the functionalities of downloadable software components for suitable reuse, through the case study of Java. Though the style of specification here is not different from the style of traditional specification, we can clarify how and what kind of properties should be described. Formalization for JVM is already proposed [3] and the security issue of Java is also reported [1], but these researches are not intended to encourage the reuse of components.

While component users can design their own system in detail by the flexibility of security policy, it becomes harder to specify and coordinate the system. For example, if each component in a system has different security policy, it is not so easy to identify the functionality of each component in runtime. In Java2, `Permission` and `AccessController` class are introduced for flexible security policies [9]. Moreover, many kinds of security model are available for mobile codes now, e.g. sandbox, code signing, proof-carrying code and so on [7], so that we can develop more flexible and safe codes on these models. However, it will become difficult to identify the functionality of each component in runtime under the extended security model and mechanism. Therefore, specification for using mobile codes should be also extended together.

References

[1] D. Dean, E. W. Felten, and D. S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 190–200, May 1996.

[2] L. Gong. Secure Java Class Loading. *IEEE Internet Computing*, 2(6):56–61, Nov. and Dec. 1998.

[3] T. Jensen, D. L. Metayer, and T. Thorn. Security and Dynamic Class Loading in Java: A Formalization. In *Proceedings of International Conference on Computer Languages*, pages 4–15, May 1998.

[4] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 36–44, Oct. 1998.

[5] B. Meyer. The Next Software Breakthrough. *COMPUTER*, 30(7):113–114, Jul. 1997. IEEE/CS.

[6] J. Meyer and T. Downing. *Java Virtual Machine*. O'Reilly, first edition, Mar. 1997.

[7] A. D. Rubin and J. Daniel E. Geer. Mobile Code Security. *IEEE Internet Computing*, 2(6):30–34, Nov. and Dec. 1998.

[8] Sun Microsystems, Inc. *Java Remote Method Invocation Specification*, Feb. 1997. Revision 1.4, JDK1.1 FCS.

[9] Sun Microsystems, Inc. *Java Security Architecture (JDK1.2)*, Oct. 1998. Version 1.0.

[10] T. Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, Sep. 1997.