Using RAIC for Dependable On-line Upgrading of Distributed Systems

Chang Liu Debra J. Richardson
Information and Computer Science, University of California, Irvine
{liu, djr}@ics.uci.edu

Abstract

Redundant Arrays of Independent Components (RAIC) is a technology that uses groups of similar or identical distributed components to provide dependable services [1,2,3]. RAIC allows components in the redundant array to be added or removed dynamically during run-time. A special case of RAIC can be used to perform dependable on-line upgrading of distributed systems. This position paper gives a brief overview of RAIC and discusses its application in on-line upgrading of distributed systems. A proof-of-concept example is given to illustrate how problems occur during upgrading can be masked by RAIC and would not affect smooth operations of the system-under-upgrade.

1. Introduction

Several problems arise when performing on-line upgrading of distributed component-based software systems. First, how to keep the overall system functional while individual components are being upgraded? Second, if a newly upgraded component causes problems in the system, how to detect the failures and revert to the original component without disrupting system operation? Third, if a newly upgraded component causes problems in a part of the system, how to allow that part of the system to revert to the original component while the rest of the system uses the upgraded one?

While certain technologies such as late-binding, server-side component lifetime management, and side-by-side execution of different versions of the same component make it possible to switch components or perform on-line upgrading during run-time, significant knowledge and preparation are required for systems and applications to be enabled for on-line upgrading. *Redundant arrays of independent components* (RAIC) uses groups of similar or identical distributed components to provide higher dependability, better performance, or greater flexibility than what can possibly be achieved by using any of those individual components. By putting different versions of a component-under-upgrade in a redundant array and routing all connections in the system to that component via a RAIC controller, it is possible to leverage on the RAIC technology and address the three problems of on-line upgrading listed above without complicating application or system logic.

In this position paper, RAIC is briefly explained with emphasis on its aspects related to the on-line upgrading problems. A proof-of-concept *Light* example is given to illustrate the functions of RAIC controllers and how failures in *Light* components are detected and masked while the *Light* applications run smoothly.

2. RAIC Overview

A *redundant component array* (also referred to as RAIC) is a group of similar or identical components. The group uses the services from one or more of components inside the group to provide services to applications. Applications connect a RAIC and use it as a single component. Applications typically do not have any knowledge of the underlying individual components.

Depending on the types and relations of components in a RAIC, it can be used for many different purposes under different types of RAIC controllers. A *RAIC controller* contains software code that coordinates individual software components in a RAIC. Not all types of RAIC controllers apply to all combinations of component types and relations. It is essential to determine component types and relations prior to configuring a RAIC.

Component Types. There are mainly two types of components in terms of whether or not they maintain internal states: *stateless* components, denoted by "( )", and *stateful* components, denoted by "[ ]".

In a stateful component, each public method can be either *state-preserving*, *state-changing*, or *state-defining*. The return value of a method can be either *state-dependent* or *state-independent*.

A RAIC can be either *static*, denoted by "-", or *dynamic*, denoted by "~". Components in a static RAIC are explicitly assigned by mechanisms outside the RAIC, whereas components in a dynamic RAIC may be discovered and incorporated by the RAIC controller during run-time. Dynamic RAIC controllers may use directories such as UDDI to locate new components [4]. Either way, RAIC controllers allow addition or removal of components during run-time and take care of component state recovery when necessary as new stateful components are added.

Component state recovery. Component types and method properties help RAIC controllers to decide what to do in the event of *component state recovery*. For stateless components, no state recovery is necessary. A

newly created component can be used in place of another component right away. For stateful components, their states must be restored before they are used in lieu of other components. There are primarily two ways to perform state recovery: *snapshot-based recovery* and *call-history-based recovery*. The snapshot-based approach assumes that the state of a component is represented by its snapshot, which is a copy of all of its internal variables. The call-history-based approach assumes that placing an exact same call sequence to equivalent components results in the same component state. Method properties help reduce the amount of call histories that are need for state recovery purposes. For example, all state-preserving calls can be trimmed off because these calls do not change component states at all.

Just-in-time component testing. RAIC controllers need to know when a component fails and when to trigger component state recovery. *Just-in-time component testing* does just this. Different from traditional software testing and perpetual testing [5], as an integral part of RAIC controllers, just-in-time testing tries to determine if a component functions as intended during run-time without using extensive test data [6].

Component relations. There are many aspects of relations between components. Nearly universally applicable are aspects such as interfaces, functionalities, domains, and snapshots. Not applicable to all components, but important nonetheless, are aspects such as security, invocation price, performance, and others. Relations of multiple components can be derived from binary relations among components.

As an example, interfaces of two components can have the following relations: *identical* (≡), *equivalent* (=), *similar* (≈), *inclusionary* (≤), or *incomparable* (≠).

While it is possible to programmatically determine interface relations by analyzing interface specifications, other relations, such as functionality relations, sometimes can only be manually determined.

Component relations are the basis of integration strategies that decide how the components are used together. For example, RAIC controllers can partition components inside a RAIC into equivalent classes and use only components inside the same class to replace each other until they run out.

RAIC levels. Most of these RAIC strategies and policies are configurable. RAIC levels describe the level and the purpose of integration among components in a redundant array:

- RAIC-1: Exact mirror redundancy
- RAIC-2: Approximate mirror redundancy
- RAIC-3: Shifting lopsided redundancy
- RAIC-4: Fixed lopsided redundancy
- RAIC-0: No redundancy

Invocation models. RAIC controllers can also use different invocation models, including:

- RAIC-a: Sequential Invocation
- RAIC-b: Synchronous Invocation
- RAIC-c: Asynchronous Invocation

RAIC can be used for purposes such as fault-tolerance, result refinement, and performance enhancement, to name just a few, where it is desirable to put components with incomparable interfaces or exclusionary domains in the same RAIC. When used for dependable on-line upgrading, however, it is likely that all components in a RAIC have identical interface relations, identical domain relations, and non-incomparable functionalities. Otherwise, the upgraded components are certain to break existing applications if no RAIC controller is present to serve as bridges.

Therefore, on-line upgrading of distributed component-based systems concerns mostly "RAIC-2a[$\equiv_i$, $\equiv_d$]", a special case of RAIC.

3. The Light Example

There is a *Light* component that provides a simple software *light* service, which simulates an adjustable light [7]. The *light* can be turned on and turned off. The intensity of the *light* can be adjusted through another method call. The following is a skeleton code in C# that defines the *Light* component [8]. The *MethodProperty* attributes specify that all three methods are *state-defining*, meaning that they change the state of the component to a specific state regardless of which state the component was in prior to the method call.

```
public interface ILight
{ [MethodProperty(MthdProperty.StateDefining)]
  int TurnOn();
  [MethodProperty(MthdProperty.StateDefining)]
  int SetIntensity(int intensity);
  [MethodProperty(MthdProperty.StateDefining)]
  int TurnOff();
}
public class Light: MarshalByRefObject, ILight
{ // ...
}
```

The first version of the *Light* component allows arbitrary method calls. An upgrade to the *Light* component, however, requires *TurnOn()* to be called before *SetIntensity()* or *TurnOff()* can be called. Similarly, *TurnOff()* cannot be called if the *light* is already off. An exception would be thrown if these requirements are not met.

There are also two applications that use the *Light* component. The first application, *LightApp1*, simply calls *TurnOn(), SetIntensity(),* and *TurnOff()* repeatedly.

```
public class LightApp1
{ public static void Main(string[] args)
  { int pause in seconds = 3;
    Light light = new Light();
    for (int i=1; i<=100; i++)
    { light.TurnOn();
      Thread.Sleep(pause_in_seconds * 1000);
      light.SetIntensity(50);
      Thread.Sleep(pause_in_seconds * 1000);
      light.TurnOff();
```

```
      Thread.Sleep(pause_in_seconds * 1000);
    }
  }
}
```

The second application, *LightApp2*, is similar to *LightApp1*. The difference is that *LightApp2* does not call *TurnOn()* at all.

```
public class LightApp2
{ public static void Main(string[] args)
  { int pause_in_seconds = 3;
    Light light = new Light();
    for (int i=1; i<=100; i++)
    { light.SetIntensity(50);
      Thread.Sleep(pause_in_seconds * 1000);
      light.TurnOff();
      Thread.Sleep(pause_in_seconds * 1000);
    }
  }
}
```

Apparently, both *Light* applications work well with the first version of the *Light* component. The upgrade of the *Light* component would break *LightApp2* but would not affect *LightApp1*.

In a distributed system where *LightApp1* and *LightApp2* run side-by-side, if an on-line upgrading of the *Light* component is attempted, *LightApp2* will undoubtedly be interrupted. An attempt to revert the *Light* component to its original version would fix *LightApp2*, but would deny *LightApp1*'s access to upgraded features of the *Light* component. By using RAIC, these problems can be avoided. Here is what happens with RAIC:

First, instead of using the concrete *Light* component directly, the *light* applications use a new component *LightRAIC*, which has the same interface *ILight* as *Light*.

```
public class LightRAIC
  : MarshalByRefObject, IRAIC, ILight
{ //...
}
```

```
LightRAIC light = new LightRAIC();
for (int i=1; i<=100; i++)
{ //...
  light.SetIntensity(50);
  //...
}
```

Second, in a system-wide configuration, *LightRAIC* is defined as "RAIC-2a[]", which means it uses the sequential invocation model and treats all components inside as stateful. Its policy is set to "latest version first". Then, the first version of the *Light* component is added to the RAIC as its only member component. After that, both *LightApp1* and *LightApp2* can run smoothly using their own instances of *LightRAIC*.

Third, during the on-line upgrading, the upgraded version of the *Light* component is added to *LightRAIC*. In *LightApp1*, the RAIC controller switches to the new component because its policy asks it to always try to use the component with the latest version. It first brings the status of the new component up-to-date by placing all calls in its trimmed call history to the new component. Then it places the current call to the new component and thus switches the application to the new component. *LightApp1* only experiences a brief delay during the switch. The operation of *LightApp1* continues without any disruption. The length of the delay depends on the number of items in the trimmed call history. In this case, since all three method calls are state-defining, there is only one item in the trimmed call history no matter how long the call history is.

In *LightApp2*, the RAIC controller also tries to switch to the new component because of the same "latest version first" invocation policy. Its just-in-time component testing mechanism detects an exception when the first *SetIntensity()* method call is placed without a preceding *TurnOn()* call. JIT testing treats the exception as a failure. The RAIC controller then tries the next available component in the RAIC, which is the original *Light* component. Since the state of that component is already up-to-date, the RAIC controller goes ahead and places the current method call and returns the result to *LightApp2*. During the on-line upgrading, *LightApp2* does not experiment any failure at all. The exception in the upgraded component was masked by the RAIC controller. *LightApp2* notices only a brief delay, the length of which is approximately one method call to the upgraded component. After that, all subsequent calls go to the original component without delay. To *LightApp2*, the on-line upgrading never happened.

Note that in this scenario, there is no application-or component-specific configuration definition that specifies which application works with which component.

4. Limitations and Conclusions

Currently, both the just-in-time component testing technique and the component state recovery technique have significant limitations. For example, if a component is connected to a persistent external storage such as a database, neither snapshot-based nor call-history-based state recovery technique may fully recover component states [9]. While some limitations are fundamental to the approach and cannot be removed by improving these two techniques alone, we feel that both techniques work or could work under broad enough circumstances that this work could produce practical results. In addition, many limitations may be lifted by adding better heuristics the two techniques.

In summary, RAIC addresses the three problems listed in the beginning of this position paper by: first, allowing run-time addition or removal of components in RAIC and automatically bringing the state of newly added component up-to-date using component state recovery techniques; second, using just-in-time component testing to detect component failures and to fall back on the original components when failures are detected in upgraded components; and third, allowing different instances of the same RAIC controller to select different components.

5. References

[1] Chang Liu and Debra J. Richardson, “Redundant Arrays of Independent Components,” *Technical Report 2002-09, Information and Computer Science, University of California, Irvine*, March 2002.

[2] Chang Liu and Debra J. Richardson, "The RAIC Architectural Style," Submitted to the 10th International Symposium on the Foundations of Software Engineering (FSE-10), March 2002.

[3] Chang Liu, "The RAIC Web Site," http://www.ics.uci.edu/~cliu1/RAIC.

[4] UDDI, “UDDI 2.0 Specification”, 2001. (http://www.uddi.org/specification.html)

[5] L.J. Osterweil, L.A. Clarke, D.J. Richardson, and M. Young. “Perpetual Testing,” *Proceedings of the Ninth International Software Quality Week*, May 1996.

[6] Chang Liu, “Just-In-Time Component Testing and Redundant Arrays of Independent Components”, *Doctoral Dissertation, Information and Computer Science, University of California, Irvine* (in progress).

[7] Craig H. Wittenberg, “Testing Component-Based Software”, *International Symposium on Software Testing and Analysis (ISSTA'2000)*, Portland, Oregon, 22-25 August 2000.

[8] ECMA, “Standard ECMA-334: C# Language Specification”, December 2001. (http://www.ecma.ch/ecma1/STAND/ecma-334.htm)

[9] R. Barga and D.B. Lomet, “Phoenix: Making Applications Robust” *Proceedings of 1999 ACM SIGMOD Conference*, Philadelphia, PA (June 1999) (562-564).