# RAIC: Architecting Dependable Systems through Redundancy and Just-In-Time Testing

Chang Liu
Information of Computer Science
University of California, Irvine
Irvine, CA 92697, USA
+1(949)824-2703
liu@ics.uci.edu

Debra J. Richardson
Information of Computer Science
University of California, Irvine
Irvine, CA 92697, USA
+1(949)824-7353
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. RAIC allows components in a redundant array to be added or removed dynamically during run-time, effectively making software components "hot-swappable" and thus achieves greater overall dependability. RAIC controllers use the just-in-time component testing technique to detect component failures and the component state recovery technique to bring replacement components up-to-date. This position paper gives a brief overview of RAIC and a proof-of-concept example to illustrate how problems occur during run-time can be masked by RAIC and would not affect smooth operations of the application.

## 1. INTRODUCTION

Software dependability has become a sufficiently important aspect of computer systems to warrant attention to the architectural level. Architectural representations capture overall designs of software systems while abstracting away low-level details [2,15]. Architectural representations can assist in improving software dependability in a number of ways. For example, architectural representations can be used in testing and verification of both the designs and the systems to help achieve higher dependability [3,13,14]. Furthermore, system dependability can be enhanced by adopting appropriate architectures and architecture styles. *Redundant arrays of independent components* (RAIC) is an attempt to achieve higher dependability and other desirable properties through a specific architecture style [8,9]. 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 any of those individual components.

With the introduction of Microsoft .NET platform and the release of tools such as Visual Studio .NET that bring the creation of XML web services to the masses, it is reasonable to expect more software applications to be built on top of remote third-party software components or XML web services. Unlike in-house components or off-the-shelf ones, these remote third-party components are not under the control of application developers. They can be upgraded without notice even when applications are running. This makes it even more important to ensure that application dependability is not affected by component failures. RAIC is designed to solve this problem through an architectural approach.

In this position paper, RAIC is briefly explained. 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 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 and *stateful* ones.

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* or *dynamic*. 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 [16]. 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. Note that components may be added to or removed from a static RAIC at runtime. The difference between a static RAIC and a dynamic RAIC is that in a dynamic RAIC, the RAIC controller takes responsibility of component discovery, whereas in a static RAIC, components are explicitly assigned to the RAIC controller.

**Component state recovery.** Component types and method properties help RAIC controllers to decide what to do in the event of *component state recovery* [10]. 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 *invocation-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 invocation-history-based approach assumes that placing an exact same call sequence to equivalent components results in the same component state. This implies that components are deterministic.

An invocation can have a method property of *state-defining*, *state-changing*, or *state-preserving*. Method properties help reduce the amount of call histories that are needed for state recovery purposes.

State-defining methods change the state of a component to specific states regardless of the previous state of the component. Different method parameters may bring the same components to different states. But same method parameters always bring components to the same states even though their previous state may be different.

State-changing methods may change the state of a component. Invocations of state-changing methods must be stored for future state recovery, unless invocations to state-defining methods are placed later.

State-preserving methods do not change the state of a component at all. Thus, it is not necessary to re-invoke calls to methods of this type. All state-preserving invocations can be safely trimmed off.

**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* ($\equiv$), *equivalent* ($=$), *similar* ($\approx$), *inclusionary* ($\leq$), or *incomparable* ($\neq$).

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.

The component relations are used to determine the integration strategy, i.e. to choose how the components interact. 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 the integration of components in RAIC. The following is a list of RAIC levels:

- RAIC-1: Exact mirror redundancy
- RAIC-2: Approximate mirror redundancy
- RAIC-3: Shifting lopsided redundancy
- RAIC-4: Fixed lopsided redundancy
- RAIC-5: Reciprocal redundancy
- RAIC-6: Reciprocal domain redundancy
- RAIC-0: No redundancy

RAIC controllers can also use different invocation models, including:

- RAIC-a: Sequential invocation
- RAIC-b: Synchronous parallel invocation
- RAIC-c: Asynchronous parallel invocation

RAIC controllers need to make judgment about the return values from individual components in the redundant array to determine whether or not to invoke another component, which result to select, or how to merge return values. To do that, RAIC controllers need to evaluate return values at run-time. Just-in-time component testing is designed for this purpose [6].

**Just-in-time component testing** is different from traditional software testing. Traditional software testing techniques use various methods to determine, through test execution, if a software application, a software component, or an even smaller unit of software code behaves as expected. Usually this is done by feeding the software-code-under-test with some pre-determined data, or test input, and comparing the result with pre-determined expected output, or test oracle. Traditional software testing happens in the development phase, when software is still under development and has not been deployed to the end user. Code that is used for testing purposes, or test harnesses, are usually removed or filtered out through conditional compilation or by other means before the final software product is deployed. Just-in-time component testing differs from traditional testing in the following aspects:

1. JIT testing happens even after application deployment. Code responsible for JIT testing is an integral part of the final software product and is shipped as such.

2. JIT testing uses mostly live input data that are unknown ahead of time. Thus it is difficult, sometimes impossible, to know if the result value is correct. Therefore, heuristics and other means must be used in place of traditional test oracles.

3. When in rare cases that predetermined test inputs are used in JIT testing, it is extremely important to ensure that test runs on these test inputs are very efficient, because any test execution on predetermined data is pure overhead during run-time and will directly place a negative impact on application performance. In

comparison, test case efficiency weighs much less in traditional software testing. In addition, any fabricated test data must not change the state of the component-under-test unless the pending invocation is a state-defining one with state-independent return results.

JIT component testing happens in run-time. This is very similar to another type of testing - perpetual testing. Perpetual testing is a class of software testing techniques that seeks seamless, perpetual analysis and testing of software products through development, deployment, and evolution [12]. The difference between JIT testing and perpetual testing is that perpetual testing is optional and removable, whereas JIT testing is an integral part of the final product. The purpose of perpetual testing is to obtain more insight of the software-product-under-test, which is usually under full control of testers, through monitoring in the real environment and thus gain data that are not available from laboratories. JIT testing, on the other hand, tries to determine on-the-fly if the result from a foreign software component is trustworthy. The foreign software component is usually not under control of the application programmer. Even their availabilities are not guaranteed.

JIT component testing is also different from certain self-checking built-in mechanisms [7,19]. The difference is that JIT testing code resides in the RAIC controller instead of the actual components.

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 dependability-enhancing purposes only, however, it is likely that all components in a RAIC have similar interface relations, identical domain relations, and non-incomparable functionalities so that they can be used interchangeably. It is also unlikely that there is a need to invoke different versions of components-under-upgrade simultaneously except when just-in-time testing needs component voting to verify return results. Therefore, for dependability purposes, "RAIC-2a[$\approx_i$,$\equiv_d$]", a special case of RAIC, is most commonly used [8].

## 3. THE *LIGHT* EXAMPLE[1]

There is a *Light* component that provides a simple software *light* service, which simulates an adjustable light. The *light* can be turned on and turned off. The intensity of the *light* can be adjusted through another method invocation. The following is a skeleton code in C# that defines the *Light* component [4]. 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 invocation.

[1] The *Light* component example was used in [18].

```
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
{
  // ...
}
```

There are two versions of the *Light* component. The first version allows arbitrary method invocations. An upgrade to the *Light* component, however, requires *TurnOn()* to be called before *SetIntensity()* or *TurnOff()* can be called[2]. 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);
    }
  }
}
```

[2] In this example, we only consider normal states when deciding method properties. Therefore, in the new version, all three methods are still state-defining.

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:

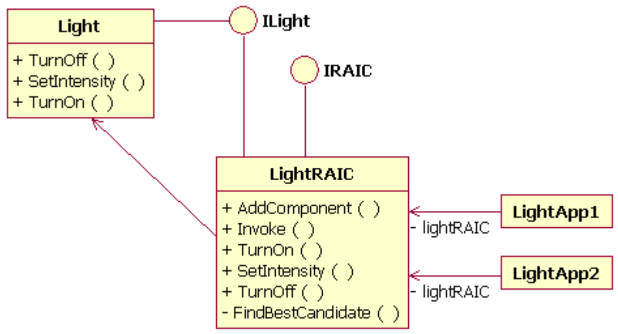

**Figure 1. With RAIC, the *Light* applications uses component *LightRAIC* instead of component *Light*.**

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*, as shown in Figure 1.

```
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 invocation 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 invocation history. In this case, since all three method invocations are state-defining, there is only one item in the trimmed invocation history no matter how long the invocation 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.

In the pre-.NET era, two versions of the same component (DLL) cannot appear on one system on Windows platforms, which means it would be impossible to have *LightApp1* using the upgraded version of the *Light* component and *LightApp2* using the original one on the same system, let alone upgrading the component at run-time.

On .NET platforms, with the support for side-by-side execution of different versions of the same component, it is now possible to do so. To achieve this, however, extra efforts are required from component developers, application developers, or system administrators to explicitly specify which application should use which version of the component. In addition, to avoid problems that may be created by over-paranoid component developers, application developers, or system administrators, .NET platform allows them to override decisions made by each other, which undoubtedly could further require more efforts from all of them. In short, even on the currently state-of-art .NET platforms, this is achievable but not pain-free.

With RAIC, this scenario is not just achievable, it is trivial with the help of just-in-time testing and component state recovery.

In this example, the two *Light* components used are two versions of the same component. It demonstrates that problems in on-line upgrading can be avoided by using RAIC [11]. RAIC, however, is not limited to arrays of different versions of the same components. In fact, the two *Light* components here can be regarded as two different components that provide similar services and all the results still hold. Examples that use different Light components can be found at [5].

## 4. LIMITATIONS AND PENDING TASKS

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 invocation-history-based state recovery technique can fully recover component states[3]. 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 improved techniques. We are working to add better heuristics to just-in-time testing and more approaches to component state recovery. For example, we are considering using component dependency information to broaden applicability of the component state recovery technique [17].

## 5. SUMMARY

In summary, dependability-through-redundancy can be achieved by adopting a special RAIC architecture style. Just-in-time component testing and component state recovery techniques can be used to coordinate redundant components so that applications are not exposed to the complexity of the integration of redundant components.

## 6. REFERENCES

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

[2] L. Bass, P. Clements, and R. Kazman, "Software Architecture in Practice", SEI Series, Addison-Wesley January 1998. ISBN: 0201199300.

[3] A. Bertolino, F. Corradini, P. Inverardi, H. Muccini, "Deriving Test Plans From Architectural Descriptions", *Proceedings of the 22nd international conference on Software engineering*, p.220-229, June 04-11, 2000, Limerick, Ireland.

[4] ECMA, "Standard ECMA-334: C# Language Specification", December 2001. http://www.ecma.ch/ecma1/STAND/ecma-334.htm.

[5] C. Liu, "The RAIC Web Site," 2002, http://www.ics.uci.edu/~cliu1/RAIC.

[6] C. 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] C. Liu and D. J. Richardson, "Software Components with Retrospectors," International Workshop on the Role of Software Architecture in Testing and Analysis, Marsala, Sicily, Italy, July, 1998.

[8] C. Liu and D. J. Richardson, "Redundant Arrays of Independent Components", *Technical Report 2002-09, Information and Computer Science, University of California, Irvine*, March 2002.

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

[10] C. Liu and D. J. Richardson, "Specifying Component Method Properties for Component State Recovery in RAIC", *Accepted by the Fifth ICSE Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly (ICSE2002), Orlando, Florida, USA,* May 19-20, 2002.

[11] C. Liu and D. J. Richardson, "Using RAIC for Dependable On-line Upgrading of Distributed Systems", *Submitted to the Dependable On-line Upgrading of Distributed Systems Workshop held in conjunction with COMPSAC 2002 (August 26-29 2002, Oxford, England)*, March 2002.

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

[13] M. D. Rice, S. B. Seidman, "An Approach To Architectural Analysis And Testing", *Proceedings of the third international workshop on Software architecture*, p.121-123, November 01-05, 1998, Orlando, Florida, United States.

[14] D. J. Richardson and A. L. Wolf, "Software Testing At The Architectural Level", *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, p.68-71, October 16-18, 1996, San Francisco, California, United States.

[15] M. Shaw and D. Garlan, "Software Architecture: Perspectives on an Emerging Discipline", Prentice-Hall, Englewood Cliffs, NJ, 1996. ISBN: 0131829572.

[16] UDDI, "UDDI 2.0 Specification", 2001. http://www.uddi.org/specification.html.

[17] M. Vieira, M. Dias, D. J. Richardson, "Describing Dependencies in Component Access Points", *Proceedings of the 4th Workshop on Component Based Software Engineering, 23rd International Conference on Software Engineering (ICSE'01, Toronto, Canada)*, May, 2001 pp.115-118.

[18] C. H. Wittenberg, "Testing Component-Based Software", *International Symposium on Software Testing and Analysis (ISSTA'2000),* Portland, Oregon, 22-25 August 2000.

[19] S. S. Yau and R. C. Cheung, "Design of Self Checking Software," In Proceedings of the International Conference on Reliable Software, April, 1975, pp. 450-457.

[3] This problem was addressed by Phoenix [1].