ICS

TECHNICAL REPORT

UCI-ICS-TR-02-09

Redundant Arrays of Independent Components

Chang Liu, Debra J. Richardson

Information of Computer Science
University of California, Irvine
Irvine, CA 92697, USA

{liu,djr}@ics.uci.edu

Information and Computer Science
University of California, Irvine

UCI-ICS-TR-02-09

Redundant Arrays of Independent Components

Chang Liu, Debra J. Richardson

Information of Computer Science
University of California, Irvine
Irvine, CA 92697, USA

{liu,djr}@ics.uci.edu

ABSTRACT

Component-based software development technologies have been advocated for years [31]. Recent developments in the software industry are posing to make it as easy to develop a distributable software component as it is to code a traditional software module [21, 23]. An abundance of standard-binding inexpensive software components is about to emerge. It will soon be possible to use redundant software components to enhance application reliability or to improve system performance without doubling or tripling component costs. Component integration cost, however, remains high. Before average software developers can take advantage of the coming abundance of low-cost software components, component integration techniques must be improved so that the benefits of adopting redundant components outweigh component integration cost.

Redundant Arrays of Independent Components (RAIC) is a technology that uses groups of similar or identical distributed software components to provide reliable services to software applications. The RAIC architectural style is a special architectural style designed to take advantage of redundant independent components in a systematic way. The types and relations of components in a RAIC is the basis of how they should be integrated. After component types and various component relations are determined, an appropriate RAIC level and an invocation model can be adopted. The RAIC level and the invocation model dictate how a RAIC controller functions.

RAIC controllers use the *just-in-time component testing* technique to check component status and detect component failures. *RAIC feelers* provide other status information to assist decisions in component selection. RAIC allows components in a redundant array to be added or removed dynamically at run-time. *Component state recovery* techniques are used to bring replacement components or newly added components up-to-date.

Together, these systematic strategies and supporting technologies enable software developers to integrate redundant software components in an array with no or little coding. They can then use the array as a single component. Thus, by following the guidance of the RAIC architectural style, component integration cost can be lowered.

This technical report describes of RAIC and the RAIC architectural style. It presents categorizations and definitions of component types, component relations, RAIC levels, and invocation models. It also discusses the just-in-time testing and component state recovery techniques used in RAIC. A number of examples and scenarios are given to illustrate different types of RAIC. Future research directions in RAIC are also outlined.

Table of Contents

List of Figures

List of Tables

1. INTRODUCTION

Component-based software development technologies [18, 31] have been advocated for years. Various component services have also been available in commercial software for several years [16, 17, 36]. With the introduction of Microsoft .NET platform [21, 28] and the release of tools such as Visual Studio .NET [23] that brings the creation of XML web services [10] to the masses, it is reasonable to expect distributed software components to boom just as the explosion of HML web pages fostered by industry standards and tools a few years ago [3]. More application will be built on top of third-party software components or XML web services, which will likely be free or very inexpensive to use, just as today's HTML web pages. Unlike in-house components or off-the-shelf ones, however, these third-party XML web services are not under the control of application developers. They can be upgraded without notice even when applications are running. Such uncontrollable upgrades would undoubtedly increase the chance of component failures, in which case it becomes necessary to seek alternatives. In addition, the upcoming abundance of these inexpensive components will probably make alternatives available at very low cost. Using redundant components to enhance application reliability becomes a natural solution.

Reliability-through-redundancy is not new. From redundant cooling systems of space shuttles in the non-computer world, to redundant hard disks of RAID in the hardware world, to redundant voting aircraft control systems in the software world, redundancy has been used to enhance reliability. Software redundancy has not been popular outside safety-critical systems mainly because of the high cost of both creating redundant systems or components and integrating them. The coming abundance of XML web services can solve the former problem, but not the latter one. When the cost of integration exceeds the benefits of using redundant components, few will adopt redundancy even when all the components are free.

Redundant Arrays of Independent Components (RAIC) is a technology that attempts to achieve higher software reliability by using more than one identical or similar software components redundantly. The basic idea is that when one software component fails for whatever reason, another one can be used in place of it so that software applications that use the components remain operational. As discussed above, RAIC could potentially come with high component integration cost. The *RAIC architectural style* is a way to address this problem [13]. It attempts to lower integration cost by using a clearly defined systematic approach. The RAIC architectural style describes the assumptions and constraints that various fundamental RAIC supporting technologies impose on the architecture of systems [2, 30]. These RAIC-supporting technologies include component selection strategies, the just-in-time component testing technique, and component state recovery techniques. The RAIC architectural style enables system designers to perform preliminary analyses on whatever components they intend to use and decide on the best way to integrate them. The RAIC-supporting technologies enable system developers to implement a RAIC style system with ease.

In this technical report, RAIC is defined and explained. Various aspects of RAIC, such as component types, component relations, RAIC levels, and invocation models, are described. Two techniques behind RAIC controllers, just-in-time component testing and component state recovery, are presented. Finally, future research directions in RAIC are discussed.

Through this report, a *Light* example is used. 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 call. Several *Light* applications use the *Light* components in various fashions.

Table 1 lists a skeleton code in C# that defines the *ILight* interface and the *Light* component [5]. The *Light* component was used in [35].

Table 1. *ILight* and *Light* in C#.

```
public interface ILight
{
  int TurnOn();
  int SetIntensity(int intensity);
  int TurnOff();
}

public class Light: MarshalByRefObject, ILight
{
  // ...
}
```

2. THE RAIC ARCHITECTURAL STYLE

Under the RAIC architectural style, redundant software components are grouped into an array. 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. On the other hand, applications under the RAIC architectural style connect to the RAIC and use it as a single component, as shown in Figure 2, instead of using individual components directly, as illustrated in Figure 1. RAIC applications typically do not have any knowledge of the underlying individual components with RAIC.

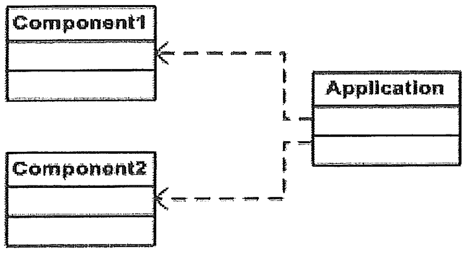


Figure 1. An application uses individual components directly.

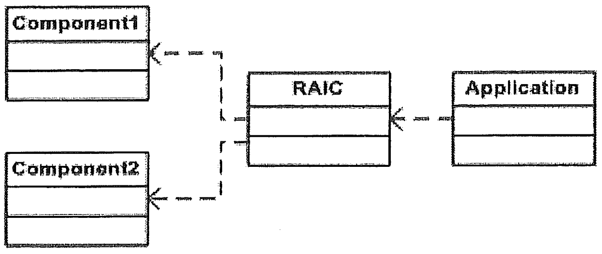


Figure 2. An application interfaces with a RAIC instead of individual components.

2.1 Component Types

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. In ad hoc implementations of component integration, code for similar purposes is sometimes called "glue code". Connectors, as defined in software architecture literatures, usually take the responsibilities of RAIC controllers, among other things. RAIC controllers can be regarded as a special form of connectors. 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.

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 " [ ] ".

A RAIC can be either *static*, denoted by "–", or *dynamic*, denoted by "~". As an example, expression "`RAIC~()`" represents a dynamic array of stateless components.

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 [32]. 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.

2.2 Component Relations

There are many aspects of relationships 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.

2.2.1 *Interface Relations*

Interfaces of two components can have the following relations: identical (≡), equivalent (=), similar (≈), inclusionary (⊆), or incomparable (≠).

Table 2. Interface specifications, in IDL[9].

```
[uuid(54444504-8F80-4C0B-9BAD-7E3EA83E2DD1)]
interface Interface1
{
  HRESULT Function1();
};

[uuid(54444504-8F80-4C0B-9BAD-7E3EA83E2DD2)]
interface Interface2
{
  HRESULT Function1();
};

[uuid(54444504-8F80-4C0B-9BAD-7E3EA83E2DD3)]
interface Interface3
{
  HRESULT Function1(char parameter1);
  HRESULT Function2(float parameter1, char parameter2);
};

[uuid(54444504-8F80-4C0B-9BAD-7E3EA83E2DD4)]
interface Interface4
{
  HRESULT Function3(char parameter1);
  HRESULT Function4(double argument1, char argument2);
  HRESULT Function5();
};

[uuid(54444504-8F80-4C0B-9BAD-7E3EA83E2DD4)]
interface Interface5
{
  HRESULT Function3(char parameter1);
  HRESULT Function4(char argument_A, float argument_B);
};
```

Identical (≡): Two components have identical interfaces if and only if both components implement the exact same interface. When components are compiled separately without reference to the same source definition of the interface, which is a common situation because components are often produced by different developers, a globally unique identifier such as a GUID[1] is usually used to identify the interface definition. In this case, two components have identical interfaces only when both implement an interface

[1] Globally Unique Identifier (GUID) is a 128-bit unique identification string. When printed, it typically looks like this: 12345678-1234-1234-1234-123456789ABC. Also known as a Universally Unique Identifier (UUID).

with the same unique interface identifier. In UDDI, a tModel key[2] is used to identify type specifications [32].

Equivalent (=): Component *A* and component *B* have equivalent interfaces if and only if both *A* and *B* implements same interfaces. These interfaces may be identified by different identifiers. For example, if component *A* implements *Interface1* and component *B* implements *Interface2*, as shown in Table 2, since *Interface1* and *Interface2* are the same except for the different identifiers, *A* and *B* have equivalent interfaces.

Inclusionary ($\subseteq$): An interface of component *A* is a subset of the corresponding interface of component *B* if and only if every possible call to each function in the interface that *A* implements can be converted to a call to a corresponding function in B's interface without any lose of information. In this case, component *A* and component *B* have inclusionary interfaces. For example, if *A* implements *Interface3* and *B* implements *Interface4* in Table 1, since *Function3()* in *Interface4* has the same signature as *Function1()* in *Interface3,* each call to *Function1()* can be mapped *to Function3()* without change. Similarly, because conversion from type *float* to type *double* does not cause any lose of accuracy, all calls to *Function2()* in *Interface3* can be mapped to calls to *Function4()* in *Interface4* with out any lose of accuracy. Therefore, the interface of *A* is a subset of the interface of *B*

Similar ($\approx$): Component *A* and component *B* have similar interfaces if and only if the interfaces of *A* and *B* have mutually inclusionary relations. For example, if *A* implements *Interface3* and *B* implements *Interface5*, *A* and *B* have similar interfaces because all calls to *Function4()* in *Interface5* can be mapped to calls to *Function2()* in *Interface3* by simply exchanging the positions of two parameters, and vice versa.

Incomparable ($\neq$): When the interfaces of two components have none of the above relations, these two have incomparable interfaces.

From these definitions, it is trivial to infer that all identical interfaces also have equivalent, similar, and inclusionary relations. All equivalent interfaces also have similar and inclusionary relations.

When there are more than two components in a RAIC, the relation of all components in terms of interfaces is determined by all binary relations among interfaces of all components using the following *relation combination rules*:

- *Rule 1*: If all binary relations among all components are the same, then that binary relation represents the relation of all components in the RAIC.
- *Rule 2*: Otherwise, the least strict binary relation represents the relation of all components in the RAIC.

The order of strictness of all binary interface relations is listed in Table 3.

Table 3. Order of strictness of binary interface relations.

Strictness	From highest to lowest				
Interfaces	$\equiv$	$=$	$\approx$	$\subseteq$	$\neq$

2.2.2 *Functionality Relations*

Functionalities of two components can have the following relations: identical ($\equiv$), equivalent (=), similar ($\approx$), inclusionary ($\subseteq$), or incomparable ($\neq$).

[2] A tModel key is a GUID-based number that identifies tModels. See the API Specification of the UDDI specification.

Identical (≡): Two components have identical functionalities if and only if they are compiled from the same piece of source code.

Considering that different compilers may cause slightly different behaviors, either due to different features or defects, it may be enticing to define the identical relation as having the exact same binary code. But the fact is that different platforms may cause even the exact same binary code to behave differently. To make the identical relation a practical one that helps in component integration, we decide to treat all components that are implemented by the same source code as identical.

Equivalent (=): Two components have equivalent functionalities when they are implemented according to the same specification. In the real world, different implementations of any realistic component are very likely to behave slightly differently under some conditions even if they are implemented according to the same specification. The essence of the equivalent relation here is to capture the intention behind component implementations. If two components are designed to be interchangeable, we should treat them as such in component selection.

Similar (≈): Two components have similar functionalities when they are implemented to perform the same tasks but with different requirements. For example, suppose there are two *TerraService* components that provide a map when given a U.S. zip code. The former component provides maps in 400 pixels by 300 pixels size, while the latter provides maps in 800 pixels by 600 pixels size. These two components are not equivalent or identical since the return values are different intentionally. They are similar because they accomplish the same tasks.

Inclusionary (⊆): Two components A and B have inclusionary relations, i.e. the functionalities of component *A* is a subset of the functionalities of component *B*, if every possible task that *A* performs can be done by *B*, maybe with different accuracy, but nonetheless accomplishable by *B*.

Incomparable (≠): When two components' functionalities have none of the above relations, these two have incomparable functionalities.

It can be inferred that all components with identical functionalities also have equivalent, similar, and inclusionary functionalities. All components with equivalent functionalities also have similar and inclusionary functionalities.

When there are more than two components in a RAIC, the relation of all components in terms of functionalities is determined by all binary relations among the functionalities of all components in the RAIC using the relation combination rules. The order of strictness of all binary functionality relations is listed in Table 4. The less strict a binary relation is, the more representative.

Table 4. Order of strictness of binary functionality relations.

Strictness	From highest to lowest				
Functionalities	≡	=	≈	⊆	≠

2.2.3 Domain Relations

Domains of two components can have the following relations: identical (≡), inclusionary (⊆), exclusionary (‖), or incomparable (≠).

For components with similar interfaces, it may be possible to further compare their input domains. Domains of two components can have the following relations:

Identical (≡): Two components have identical domains if and only if they respond meaningfully to any possible inputs that are the same, i.e. both return results are useful to caller applications, not an error message or an exception.

Inclusionary (⊆): The domain of component *A* is a subset of the domain of component *B* when each input in *A*'s valid input domain is also in *B*'s valid input domain. In this case, *A* and *B* are said to have inclusionary domains. For example, suppose two *StockQuote* components both provide real-time quote information. One component only provides quote of NASDAQ stocks. The other component provides quote of both NASDAQ and NYSE stocks[3]. In this case, the first component's domain is a subset of that of the second component.

Exclusionary (‖): The domain of component A is exclusionary to the domain of component B when none of the valid input in A's domain is in B's domain, nor vice versa. For example, suppose there is a third *StockQuote* component that provides quote information only to NYSE stocks. This component has a parallel domain if compared to the first *StockQuote* component above that only provides NASDAQ stock quotes.

Incomparable ($\nsim$): When two components' domains have none of the above relations, these two have incomparable domains.

Note that even components with incomparable functionalities may possibly have comparable domains.

It can be inferred that all components with identical domains also have inclusionary domains. Also, obviously, inclusionary domains are not exclusionary. In addition, mutually inclusionary domains are identical domains.

When there are more than two components in a RAIC, the relation of all components in terms of domains is determined by all binary relations among domains of all components using the relation combination rules introduced above and one additional rule. The third rule is added because in the case of domain binary relations, exclusionary (‖) cannot be compared with identical (≡) or inclusionary (⊆) in terms of strictness.

❖ *Rule 3*: If two binary relations that cannot be compared for strictness exist among binary relations of all components in the RAIC, the relation of all components in the RAIC is incomparable (≠).

The order of strictness of all binary domain relations is listed in Table 5.

Table 5. Order of strictness of binary domain relations.

Strictness	From highest to lowest		
Domains	≡	⊆	≠
	‖		

For example, suppose a RAIC has three components A, B, and C. In terms of domains, these binary relations exist:

A≡B, B≡C, C‖A.

The domain relation of this RAIC is incomparable (≠).

2.2.4 *Discussions on Interface, Functionality, and Domain Relations*

The relations of two interfaces can be determined solely from interface specifications. Since formal interface specifications usually exist as parts of the source code or separate pieces of formal documents, this process may be fully automated. Functionality specification, however, are not usually formally

[3] NYSE and NASDAQ are two U.S. stock exchange markets.

specified, even though formal specification technologies do exist. Existing code understanding technologies are not mature enough to be used to compare two pieces of arbitrary code. Therefore, either application programmers or component developers have to determine functionality relations manually. The same is true for domain relations. Testing may provide assistance in functionality and domain relation analysis.

Interface relations alone do not shed much light on the relations of two components, except for the identical interface relation. Only when combined with the functionality relation and the domain relation can interface relations give meaningful insights about the relations of two components.

Not all combinations of different binary relations are useful. For example, it is obvious that RAIC-[$\equiv_i$, $\neq_f$] is not an interesting type of RAIC. By using the same interface to provide unrelated functionalities, the sole purpose of interface identifier is violated. The result is not meaning at all. On the other hand, RAICs such as RAIC-[$\equiv_i$, $\approx_f$] and RAIC-[$\approx_i$, $\approx_f$] are common in the real world.

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.

2.2.5 *Snapshot Relations*

A *snapshot* is a collection of the values of all data members of a component at a particular moment. It could be created through runtime mechanisms provided by an OS or a platform, serialization mechanisms provided by languages, or component-customized mechanisms. A component restored from its snapshot should be exactly the same in memory as when the snapshot was taken. This, however, does not necessarily mean that the component is in the same state from component users' perspective before a component may store part of its state information in some external storage.

Snapshots of two components can have these two relations: Identical ($\equiv$) and Incomparable ($\neq$).

Identical ($\equiv$): Two components have identical snapshots if and only if one component can use a snapshot of the other component and vice versa. Identical snapshots mean every all fields in the snapshot have the same order, same size, same structure, and same semantics. Typically, two components with identical snapshot relation have the exact same data members and only differ in program logic.

Incomparable ($\neq$): When two components' snapshots are not identical, they are incomparable.

The identical relation is stricter than the incomparable one.

2.2.6 *Notations*

The following notation, known as a *RAIC expression*, is used hereafter to refer to a RAIC with identical interfaces, similar functions, exclusionary domains, and incomparable snapshots.

```
RAIC-[≡i, ≈f, ‖d, ≠s]
```

This RAIC expression also specifies that the RAIC is a static array containing stateful components.

2.2.7 *Security, Invocation Price, and Other Aspects*

Other sometimes-comparable aspects of components may also be relevant in component integration and selection. One example is security. For example, in C#, some components can be signed; some can be delay-signed; while others can be unsigned at all [5, 20]. They offer different security options. When using several components with same functionalities but different security strengths, an application may choose not to use the less secure ones unless the more secure ones become unavailable. In this case, there is a need to compare the security aspect. Similarly, when component invocations are not free, an application may choose to invoke cheaper ones whenever possible and keep the more expensive ones as backups. Performance, communication bandwidth, and other aspects may also be relevant. Some of these are domain-specific; some are application-specific. There may not be a universal comparison model that fits well with all of them. We feel that it is better to leave the detailed categorization of relations of these aspects until the need arises in a specific context. In general, the following relations usually apply.

Identical ($\equiv$): Two components are identical in term of a certain aspect if these two components are indistinguishable as far as this particular aspect is concerned.

Equivalent ($=$): Two components are equivalent in term of a certain aspect if these two components are exchangeable as far as this particular aspect is concerned. The difference between the identical relation and the equivalent relation is that, from an application's point of view, components with the latter relation may have tolerable differences while components with the former relation have no perceivable difference at all.

Comparable ($\sim$): A component is comparable with another component in terms of a certain aspect if it is always possible to determine which component is more desirable to applications as far as this particular aspect is concerned.

The comparable relation can be either static or dynamic. If a component is statically comparable with another one in terms of a certain aspect, one of the components is better at all time. If a component is dynamically comparable with another one, either component could be a better choice depending on occasions.

Incomparable ($\neq$): Two components are incomparable in terms of a certain aspect if no other relation applies to these two components with respect to this particular aspect. Note that the so-called "other relations" here may include aspect-specific relations other than the general "identical", "equivalent", and "better" relations described above.

When there are more than two components in a RAIC, the relation of all components in terms of a certain aspect is determined by all binary relations among all components of that particular aspect using the relation combination rules. The strictness of four general binary relations is listed in Table 6. Note that for a specific aspect, additional binary relations may be defined and their strictness as compared to other relations may be added to this table.

Table 6. Order of strictness of four general binary relations.

Strictness	From highest to lowest			
Other Aspects	$\equiv$	$=$	$\sim$	$\neq$

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.

2.3 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 a redundant array:

- 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-0: No redundancy. There is no redundancy among components. Components in the array have such relations: RAIC-[$\neq_f$]. In this case, it is just plain component integration. There is no redundancy at all.

Traditional in-house component-based software development approaches usually cannot afford developing duplicate components for the same purpose. Therefore, traditional component integration can usually be regarded as RAIC-0.

RAIC-1: Exact Mirror redundancy. Components in the array typically have such relations: RAIC-[$\equiv_i$, $\equiv_f$, $\equiv_d$, $\equiv_s$]. In exact mirror redundancy, since all components are exactly the same, there is no gain in accuracy or flexibility. The goal is to improve reliability. Required service can be delivered, as long as at least one component in the array does not fail. Thus, reliability of the RAIC is better than any of the individual components in the array, assuming the RAIC controller itself does not introduce failures.

RAIC-2: Approximate Mirror redundancy. Components in the array typically have such relations: RAIC-[$\equiv_i$, $=_f$, $\equiv_d$]. In approximate mirror redundancy, since none of components is any better than any other components, the intention is not to gain in accuracy or flexibility. The main goal is to improve reliability, the same as the goal of RAIC-1.

RAIC-3: Shifting lopsided redundancy. Components in the array typically have such relations: RAIC-[$\equiv_i$, $\approx_f$, $\equiv_d$]. One component might provide more desirable result than another. However, which one is better is unknown before results are received and compared. In addition, one component might be more desirable for some inputs or under some situations; others might be more desirable for other inputs or under other situations. One such example is two *StockQuote* components that both provide real-time quote for stocks listed on NASDAQ. How up-to-date the quote information is depends on the network conditions between the component provider and NASDAQ and between the component user and the component provider. Since network conditions vary from time to time, each component may provide more up-to-date quote information than the other one at some time, but not always.

RAIC-4: Fixed lopsided redundancy. Components in the array typically have such relations: RAIC-[$\equiv_i$, $\approx_f$, $\equiv_d$]. It is known ahead of time that one component is more desirable than another. One such example is the *TerraService* components mentioned above. RAIC-4 can be used to provided "graceful downgrades" when more desirable components fail.

RAIC-5: Reciprocal redundancy. Components in the array typically have such relations: RAIC-[$\approx_i$, $\equiv_d$]. The sum of all results is usually better than any single result. For example, suppose there are several search engine components that provide search results according to a list of key words. Assuming search results from each search engine are incomplete, but in different ways. In this case, simple mergence of all search results is better than any individual results.

RAIC-6: Reciprocal domain redundancy. Components in the array typically have such relations: RAIC-[$\approx_i$, $\approx_f$, $\|_d$]. All components provide satisfactory results. But not all components provide results for all inputs. For example, suppose several wireless phone service providers each provide a text-messaging component that can deliver a short text message to a subscriber's wireless phone. Each provider's service delivers messages only to its own subscribers' phone, not others'. In this case, the combination of all components provides a more complete service than any of those individual components.

Table 7 shows the appropriate RAIC levels for various types of RAICs.

Table 7. Typical RAIC Types for different RAIC Levels.

RAIC Levels	Typical RAIC Types
RAIC-0	RAIC-[$\neq_f$]
RAIC-1	RAIC-[$\equiv_i$, $\equiv_f$, $\equiv_d$, $\equiv_s$]
RAIC-2	RAIC-[$\equiv_i$, $=\ _f$, $\equiv_d$]
RAIC-3	RAIC-[$\equiv_i$, $\approx_f$, $\equiv_d$]
RAIC-4	RAIC-[$\equiv_i$, $\approx_f$, $\equiv_d$]
RAIC-5	RAIC-[$\approx_I$, $\equiv_d$]
RAIC-6	RAIC-[$\approx_i$, $\approx_f$, $\|_d$]

2.4 RAIC Invocation Models

RAIC controllers can also use different invocation models, including:

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

RAIC-a uses components in a one-by-one fashion. At any given time, at most one component is invoked.

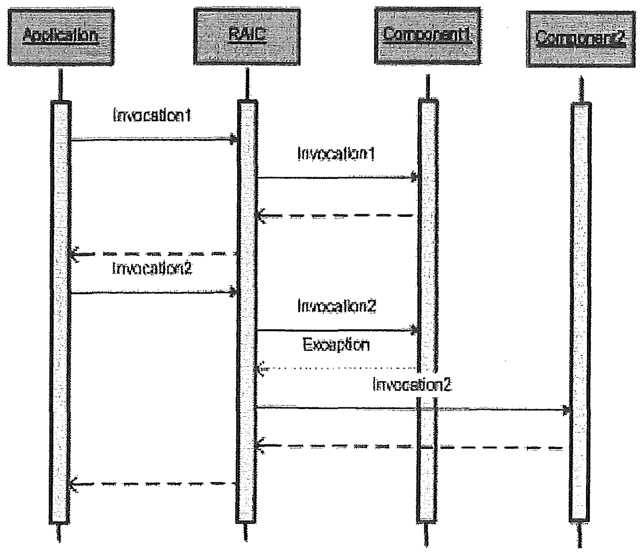


Figure 3. A UML sequence diagram of a sequential invocation scenario (RAIC-a) [6, 25].

Figure 3 shows a scenario of the sequential invocation model, where two similar components *Component1* and *Component2* are used sequentially by the RAIC controller. The RAIC controller only invokes *Component2* after detecting that *Component1* throws an exception during the second invocation. The exception is masked from the application by the RAIC controller. A good result for the second invocation from *Component2* is returned to the application. Note that in this diagram, possible component state recovery, which is discussed later, was omitted.

Both RAIC-b and RAIC-c invoke components simultaneously. The difference is that RAIC-b waits until all calls return or time-out, whereas RAIC-c goes ahead as long as satisfactory results are returned.

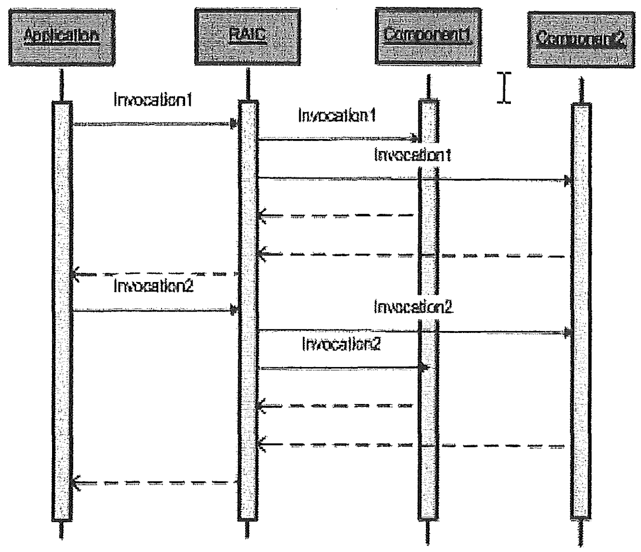


Figure 4. A UML sequence diagram of a synchronous parallel invocation scenario (RAIC-b).

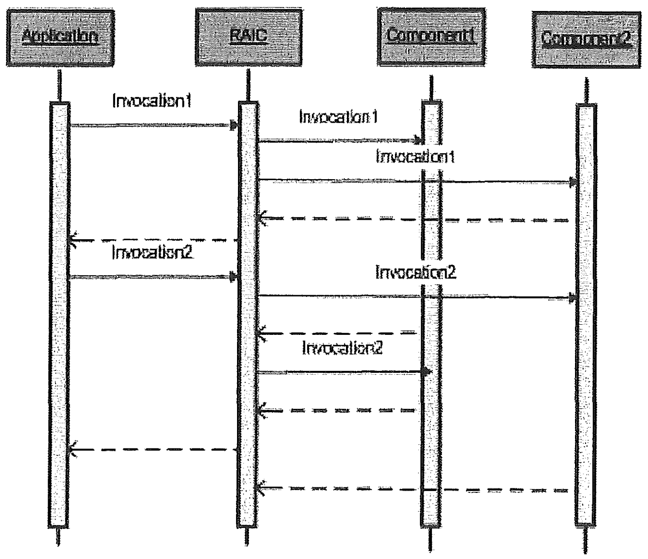


Figure 5. A UML sequence diagram of an asynchronous invocation scenario (RAIC-c).

Figure 4 shows a scenario of the synchronous parallel invocation model. Note that the RAIC controller is free to invoke the components in any order. In this case, the RAIC controller calls *Component1* first for the first invocation. for the second invocation, it calls *Component2* first instead. Since the RAIC controller can use multiple threads[4] to place invocations without having to wait for anything, the delay between invocations can be considered insignificant. In RAIC-b, all parallel

[4] Other technologies such as built-in support for asynchronous calls in the underlying component model make it possible to do this in a single thread.

invocations have to either complete or time-out before the RAIC controller returns values or an exception to the application.

Figure 5 shows a scenario of the asynchronous parallel invocation model. Notice that in this scenario, *Component2* is still working on the first invocation when *Component1* is already working on the second one. In fact, in our proof-of-concept implementation of a RAIC controller, the RAIC controller maintains an invocation queue for every single component in the array. It is possible for one component to work on the Nth call while another is already working on the (N+M)th call, where N and M can be any arbitrary natural numbers. In addition, when combined with component method properties [14], it is possible to trim items in the queues of slower components so that they don't have to go through all of them and can catch up sooner. To the application, the RAIC always behaves as if it were the fastest component.

RAIC levels have direct effects on invocation models. Some RAIC levels are best implemented by certain invocation model but not others. Some RAIC levels cannot be implemented by certain invocation model. Table 8 shows the correspondence between invocation models and RAIC levels. Table 9 shows examples mentioned above. More examples can be found in the RAIC cheat sheet on [11].

Table 8. Invocation Models and RAIC levels.

RAIC	0	1	2	3	4	5	6
a	✗	✓	✓	⊖	✓	✗	✗
b	✓	✓	✓	✓	⊖	✓	⊖
c	✗	✓	✓	⊖	⊖	✗	✓

Legends: ✓ Works best. ⊖ Works. ✗ Does not work.

Table 9. RAIC examples.

RAIC	Examples
RAIC-1c($\equiv_i$, $\equiv_f$, $\equiv_d$)	StockQuote, with identical components running on different computers as backups.
RAIC-2c($\equiv_i$, $\equiv_f$, $\equiv_d$)	StockQuote, with identical racing components running on different computers.
RAIC-3c($\equiv_i$, $=_f$, $\subseteq_d$)	StockQuote, with similar components from different providers.
RAIC-6c($\equiv_i$, $=_f$, $\|_d$)	StockQuote, with one NASDAQ component and one NYSE component.
RAIC-6c($\equiv_i$, $=_f$, $\|_d$)	Cell phone text messaging.
RAIC-4a($\equiv_i$, $\approx_f$, $\equiv_d$)	TerraService.

3. JUST-IN-TIME COMPONENT TESTING

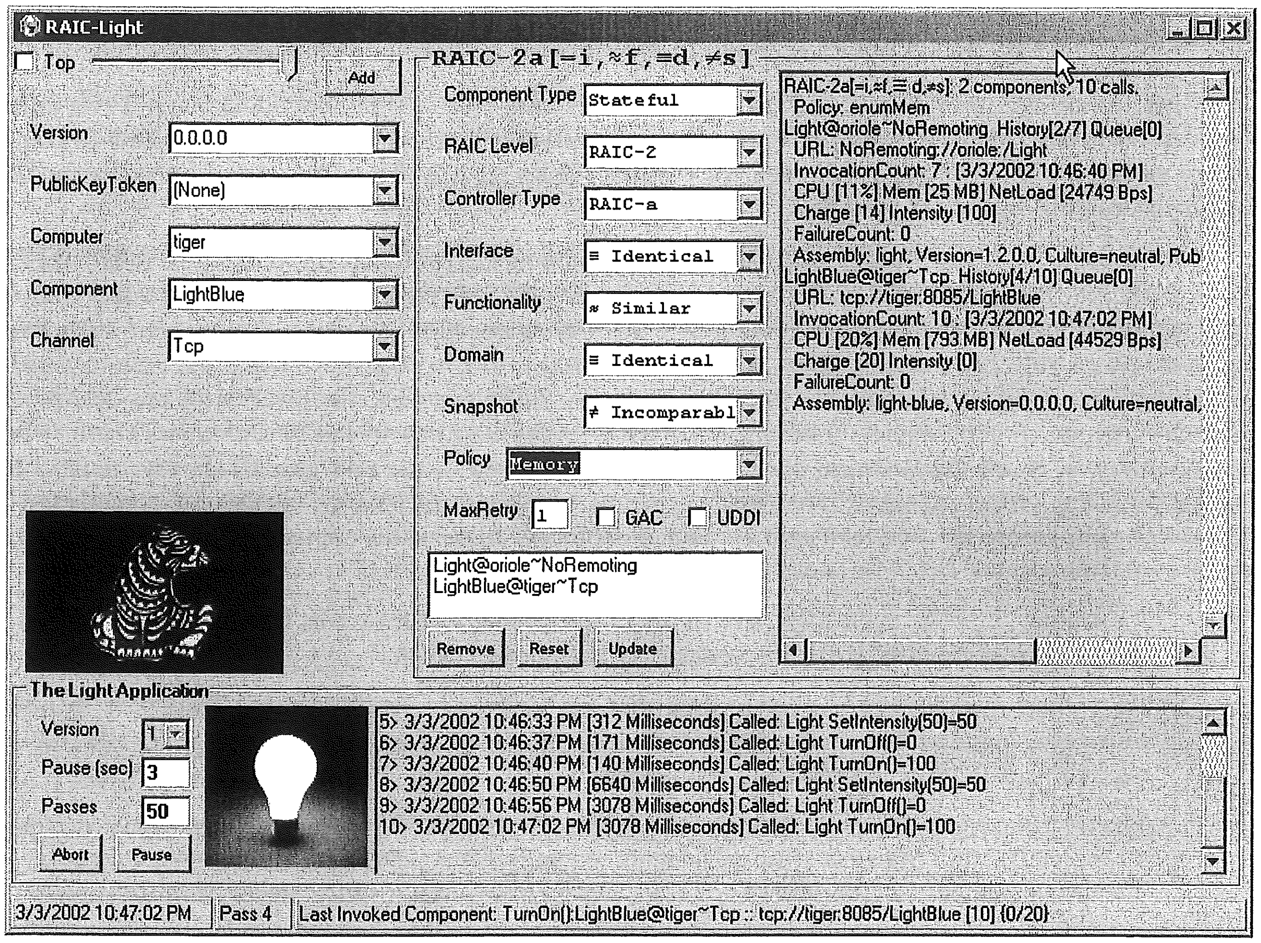
RAIC controllers need to make judgment about the return values from individual components in the array to determine whether or not to invoke another component, which result to pick, or how to merge return values. To do that, the RAIC needs to evaluate return values at run-time. *Just-in-time component testing* is designed for this purpose [12].

Just-in-time component testing uses heuristics or component specifications to evaluate return values. It 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 mostly uses 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 efficiently, 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.

JIT component testing happens in run-time. This is very similar to another type of testing - perpetual testing [26]. Perpetual testing is a class of software testing techniques that seeks seamless, perpetual analysis and testing of software products through development, deployment, and evolution. 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.

The RAIC controllers need to know certain status information in addition to the component status information that is monitored by just-in-time testing. For example, during component selection, it may be desirable to know CPU workload, available memory, or free network bandwidth of the computers on which components reside. RAIC feelers are designed for this purpose. A *RAIC feeler* is an entity inside a RAIC controller that retrieves desirable status information. In our proof-of-concept implementation, we use performance counters provided by the .NET platform to implement RAIC feelers.



Version 0.0.0.0
PublicKeyToken (None)
Computer tiger
Component LightBlue
Channel Tcp

RAIC-2a[=i,≈f,=d,≠s]
Component Type Stateful
RAIC Level RAIC-2
Controller Type RAIC-a
Interface ≡ Identical
Functionality ≈ Similar
Domain ≡ Identical
Snapshot ≠ Incomparabl
Policy Memory
MaxRetry 1

```
Light@oriole~NoRemoting
LightBlue@tiger~Tcp
```

```
RAIC-2a[=i,≈f,≡ d,≠s]: 2 components, 10 calls.
 Policy: enumMem
Light@oriole~NoRemoting  History[2/7] Queue[0]
 URL: NoRemoting://oriole:/Light
 InvocationCount: 7 : [3/3/2002 10:46:40 PM]
 CPU [11%] Mem [25 MB] NetLoad [24749 Bps]
 Charge [14] Intensity [100]
 FailureCount: 0
 Assembly: light, Version=1.2.0.0, Culture=neutral, Pub
LightBlue@tiger~Tcp  History[4/10] Queue[0]
 URL: tcp://tiger:8085/LightBlue
 InvocationCount: 10 : [3/3/2002 10:47:02 PM]
 CPU [20%] Mem [793 MB] NetLoad [44529 Bps]
 Charge [20] Intensity [0]
 FailureCount: 0
 Assembly: light-blue, Version=0.0.0.0, Culture=neutral,
```

The Light Application
Version 1
Pause (sec) 3
Passes 50

```
5> 3/3/2002 10:46:33 PM [312 Milliseconds] Called: Light SetIntensity(50)=50
6> 3/3/2002 10:46:37 PM [171 Milliseconds] Called: Light TurnOff()=0
7> 3/3/2002 10:46:40 PM [140 Milliseconds] Called: Light TurnOn()=100
8> 3/3/2002 10:46:50 PM [6640 Milliseconds] Called: Light SetIntensity(50)=50
9> 3/3/2002 10:46:56 PM [3078 Milliseconds] Called: Light TurnOff()=0
10> 3/3/2002 10:47:02 PM [3078 Milliseconds] Called: Light TurnOn()=100
```

```
3/3/2002 10:47:02 PM | Pass 4 | Last Invoked Component: TurnOn():LightBlue@tiger~Tcp :: tcp://tiger:8085/LightBlue [10] {0/20}
```

Figure 6. Status information reported by RAIC feelers.

Figure 6 shows that in the *Light* example, a RAIC feeler detects that there are 793MB free memory on computer *Tiger* and only 25MB on computer *Oriole*. At the moment, the policy of the RAIC controller is to select the component on the computer with the most memory. Therefore, the RAIC controller switched to a *LightBlue* component on *Tiger*.

4. COMPONENT STATE RECOVERY

When JIT component testing detects a failure in a component and asks for a replacement component, or when a newly added component needs to be invoked, component state recovery is needed to bring the new component up-to-date. Component types 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 *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. Component snapshot relations help determine whether a snapshot can be used. The invocation-history-based approach assumes that placing an exact same invocation sequence to equivalent components results in the same component state.

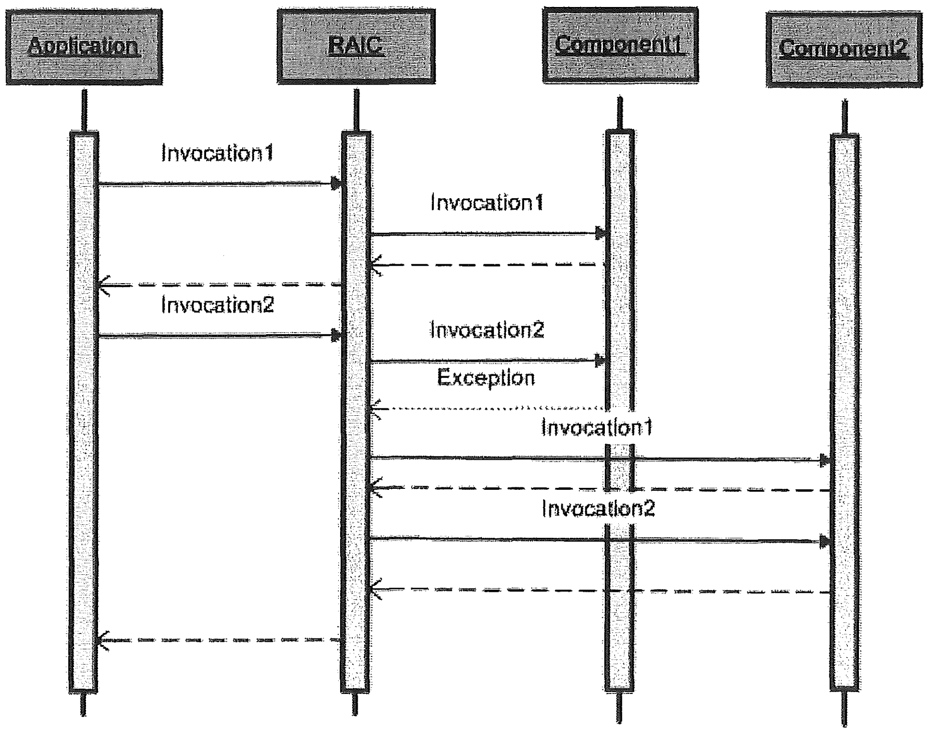


Figure 7. A UML sequence diagram of a scenario that involves component state recovery.

Figure 7 shows an invocation-history-based component state recovery scenario of a RAIC-a. Note that component creation, initialization, and all other component lifetime management issues are omitted in this diagram.

Both approaches have their own advantages and limitations. The advantage of the snapshot-based approach is that it requires limited storage, which is about the size of a component's in-memory foot print, and limited recovery time, which is about the time to copy a memory block of that size, possibly over network in the case of distributed systems. The disadvantage, however, is that this approach does not always work. If a component connects to a piece of external persistent storage such as a database and stores part or all of its state there, the snapshot would fail to preserve a component's full state. For example, consider the following code in C#, where the state of the component is saved in a system performance counter:

```
public class Light: MarshalByRefObject, ILight
{
  private PerformanceCounter IntensityCounter;

  public Light()
  {
    IntensityCounter = new PerformanceCounter();
    IntensityCounter.CategoryName = "Light";
    IntensityCounter.CounterName = "Intensity";
    IntensityCounter.ReadOnly = false;
  }

  public int SetIntensity(int intensity)
  {
    IntensityCounter.RawValue = intensity;
    return IntensityCounter.RawValue;
  }

  // ...
}
```

Using a snapshot of this component, the correct reference to the performance counter can be recovered, but not the value itself because the value is stored in system registry, not in the component.

In addition, if the replacement component is not identical to the failed component and does not have a comparable snapshot, it is not possible to recover component state at all.

The invocation-history-based recovery approach overcomes these shortcoming by storing past invocations instead of component snapshots. During component recovery, the replacement component is initialized and invoked with all the calls in the invocation history. After this, under most circumstances, the state of the new component should be the same as the failed component right before its failure. Since the RAIC controller knows how to translate calls between components with slightly different interfaces[5], it is now possible to perform component state recovery on a different component with different interfaces and incomparable snapshots. This undoubtedly broadens the applicability of the component state recovery technique. The disadvantage of this approach is that it takes space to record each invocation and time to re-invoke it. If a component is invoked repeatedly in an application, the invocation history could grow very long. It would take a large storage space to store the ever growing invocation history. And in the event of a component failure, it would take very long to re-invoke all the method calls.

4.1 Method Properties

For stateless components, since there is no need for component state recovery, it is not necessary to specify method properties.

For stateful components, method properties help reduce the amount of invocation histories that are need for state recovery purposes. Each public method of a component can have two types of properties. The first type specifies the relationship between this method and its effect on component states, which can be either *state-preserving*, *state-changing*, or *state-defining*. The second type specifies the relationship between the return value and its dependency on component states, which can be either *state-dependent* or *state-independent*.

[5] For example, RAIC controllers know how to translate calls to *SetIntensity(int intensity)* into a similar call with different parameter type: *SetIntensity(string intensity)*.

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. An example of this is the *GetIntensity()* method of the Light component.

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. *AdjustIntensity()* is a state-changing method.

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. *TurnOn()*, *TurnOff()*, and *SetIntensity()* are all state-defining methods.

Methods with state-dependent return values may return different values if the previous states of the component-under-invocation are different, even when all the parameters are the same.

Methods with state-independent return values always return the same value as long as the parameters are the same.

Both properties are optional. When property attributes are absent, the RAIC controller assumes the worst case scenario and treats the method as state-changing and its return value as state-dependent.

```
public class Light: MarshalByRefObject, ILight
{
  [MethodProperty(EnumMP.StateDefining)]
  public int TurnOn()
  {
    // ...
  }

  [MethodProperty(EnumMP.StateDefining)]
  [ReturnValue(EnumRV.StateIndependent)]
  public int SetIntensity(int intensity)
  {
    // ...
  }

  [MethodProperty(EnumMP.StatePreserving)]
  [ReturnValue(EnumRV.StateDependent)]
  public int GetIntensity()
  {
    // ...
  }

  [MethodProperty(EnumMP.StateChanging)]
  public int AdjustIntensity(int delta)
  {
    // ...
  }

  [MethodProperty(EnumMP.StateDefining)]
  public int TurnOff()
  {
    // ...
  }
}
```

With method properties, one can significantly trim component invocation history without damaging the ability for component state recovery. For example, a call history of these eleven calls on the left can be trimmed into the three calls on the right:

1 TurnOn()	1 ~~TurnOn()~~
2 SetIntensity(20)	2 ~~SetIntensity(20)~~
3 TurnOff()	3 ~~TurnOff()~~
4 TurnOn()	4 ~~TurnOn()~~
5 SetIntensity(40)	5 ~~SetIntensity(40)~~
6 GetIntensity()	6 ~~GetIntensity()~~
7 SetIntensity(50)	7 SetIntensity(50)
8 AdjustIntensity(10)	8 AdjustIntensity(10)
9 AdjustIntensity(-20)	9 AdjustIntensity(-20)
10 GetIntensity()	10 ~~GetIntensity()~~
11 GetIntensity()	11 ~~GetIntensity()~~

In general, the rules for invocation history trimming are:

1. *All invocations prior to the last call to a state-defining method can be trimmed.*
2. *All state-preserving calls can be trimmed.*

As a result, a trimmed invocation history is always an invocation of a state-defining method followed by a sequence of invocations to state-changing methods.

When component state recovery is performed when a new invocation is made to the RAIC, it is possible to further optimize the recovery process in one special situation:

If the current call is placed to a state-defining method with a state-independent return value, there is no need for any re-invocation of past calls. Simply place the current call.

4.2 The *Light* Example

Now let us use a concrete example to illustrate the process of component state recovery. A *Light* application uses the *Light* component. It simply invokes *TurnOn(), SetIntensity()*, and *TurnOff()* repeatedly. The main logic of the *Light* application is shown below:

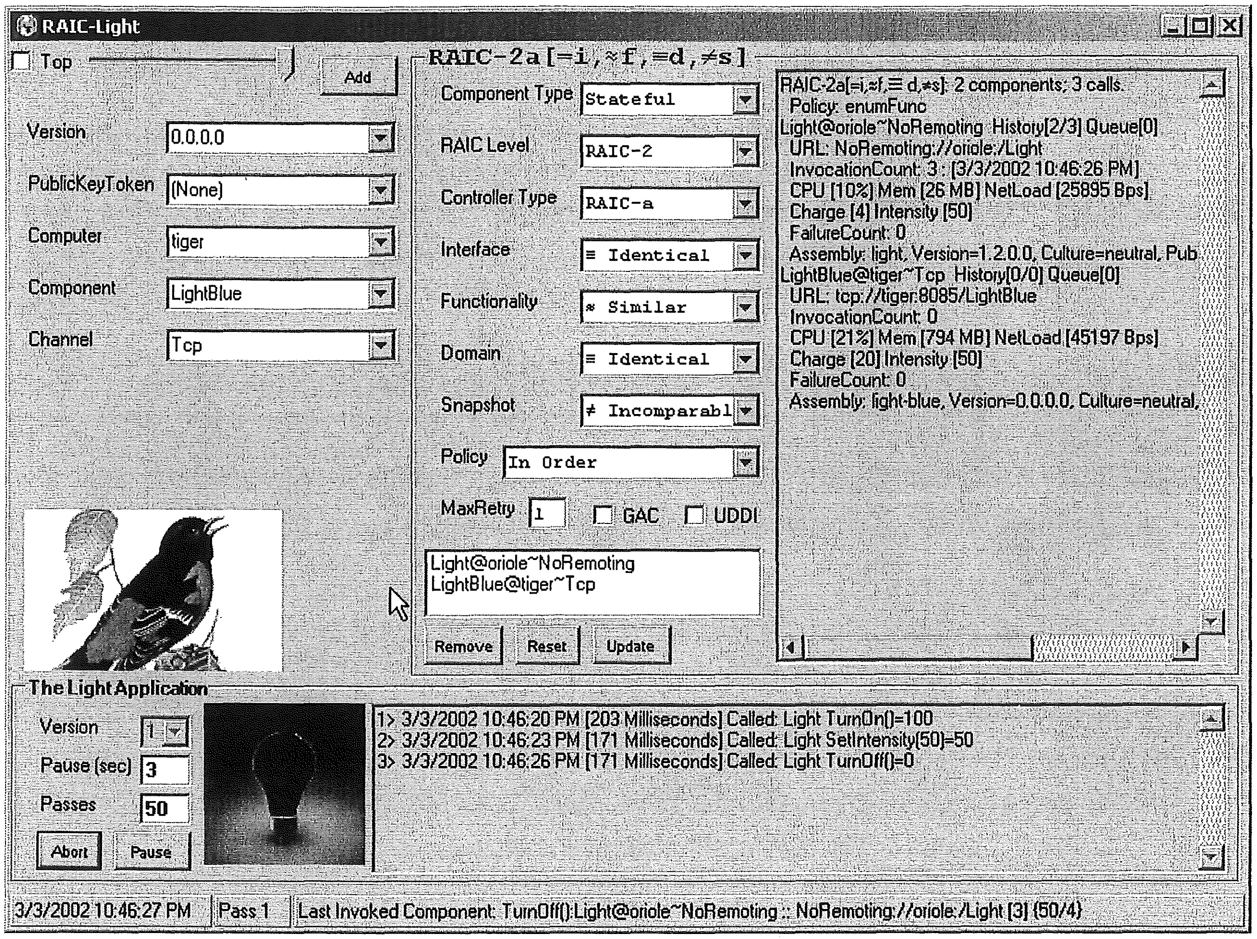
```
public class LightApp
{
  public static void Main(string[] args)
  {
    int pause_in_seconds = 3;
    LightRAIC light = new LightRAIC();

    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 actual code has a few more lines of instrumented code that display various types of status information on the GUI.

Two components are added to the RAIC, one is the *Light* component on computer *Oriole*, the other is the *LightBlue* component on computer *Tiger*. The *LightBlue* component does not have identical snapshot relation with *Light*. Thus, it is not possible to perform snapshot-based recovery in this case. Invocation-history-based recovery is the only choice. Note that while all methods of *Light* have their properties properly defined, no method properties have been defined for *LightBlue*. Therefore, by default, all methods of *LightBlue* are state-changing with state-dependent return values. Figure 8 shows the status of the *Light* application and the RAIC controller after the third method call. On the top-right corner, it shows

"History[2/3]" for the *Light* component on *Oriole*. The second number "3" represents the total number of invocations placed to this component. The first number "2" represents trimmed invocation history plus one (the current call)[6]. So we know that only one invocation is left after trimming.



RAIC-2a[=i,≈f,≡d,≠s]

Version	0.0.0.0
PublicKeyToken	(None)
Computer	tiger
Component	LightBlue
Channel	Tcp

Component Type	Stateful
RAIC Level	RAIC-2
Controller Type	RAIC-a
Interface	≡ Identical
Functionality	≈ Similar
Domain	≡ Identical
Snapshot	≠ Incomparabl
Policy	In Order
MaxRetry	1

☐ GAC ☐ UDDI

```
Light@oriole~NoRemoting
LightBlue@tiger~Tcp
```

```
RAIC-2a[=i,≈f,≡ d,≠s]: 2 components; 3 calls.
 Policy: enumFunc
Light@oriole~NoRemoting  History[2/3] Queue[0]
 URL: NoRemoting://oriole:/Light
 InvocationCount: 3 : [3/3/2002 10:46:26 PM]
 CPU [10%] Mem [26 MB] NetLoad [25895 Bps]
 Charge [4] Intensity [50]
 FailureCount: 0
 Assembly: light, Version=1.2.0.0, Culture=neutral, Pub
LightBlue@tiger~Tcp  History[0/0] Queue[0]
 URL: tcp://tiger:8085/LightBlue
 InvocationCount: 0
 CPU [21%] Mem [794 MB] NetLoad [45197 Bps]
 Charge [20] Intensity [50]
 FailureCount: 0
 Assembly: light-blue, Version=0.0.0.0, Culture=neutral,
```

The Light Application

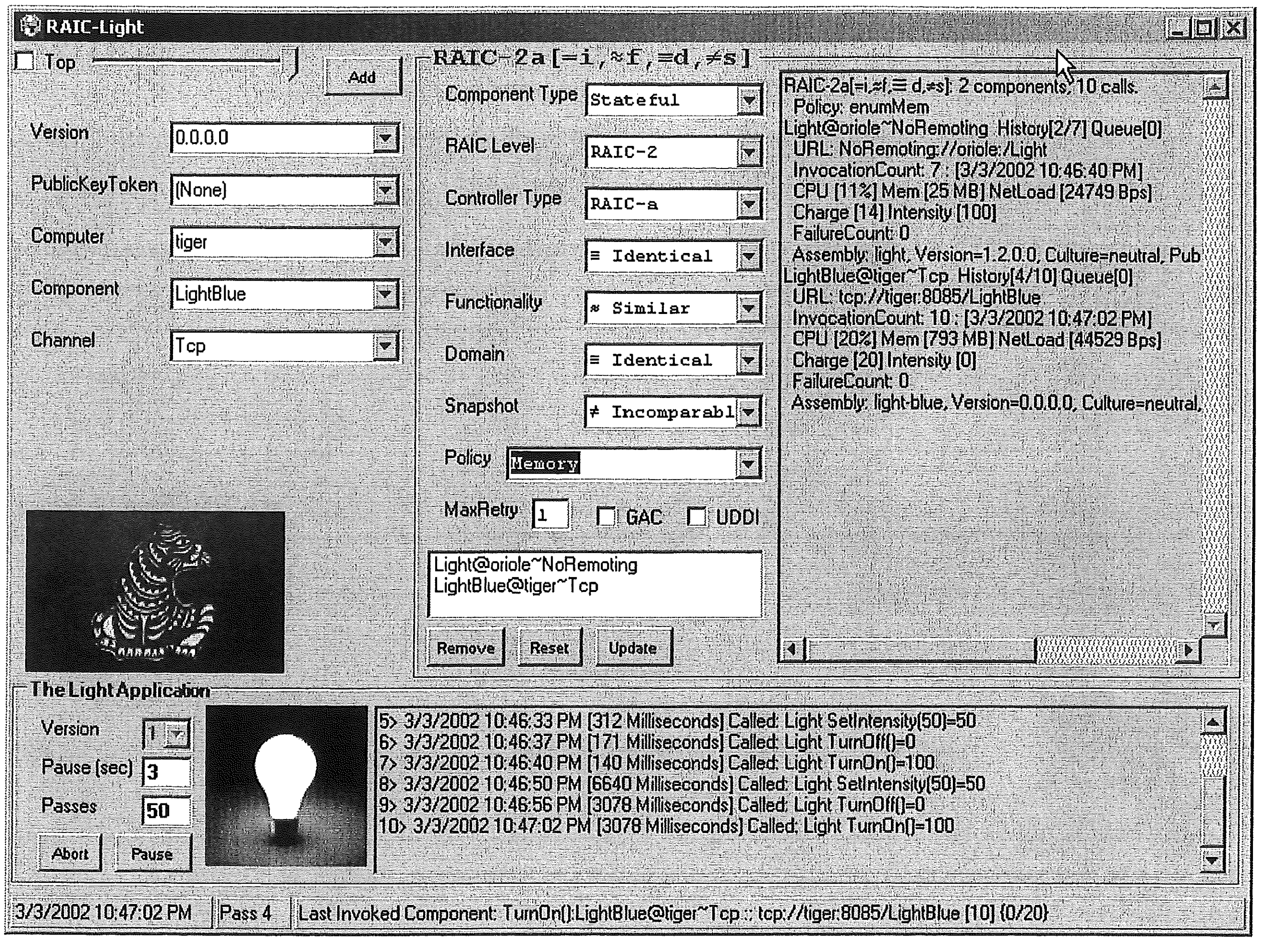
Version	1
Pause (sec)	3
Passes	50

```
1> 3/3/2002 10:46:20 PM [203 Milliseconds] Called: Light TurnOn()=100
2> 3/3/2002 10:46:23 PM [171 Milliseconds] Called: Light SetIntensity(50)=50
3> 3/3/2002 10:46:26 PM [171 Milliseconds] Called: Light TurnOff()=0
```

```
3/3/2002 10:46:27 PM | Pass 1 | Last Invoked Component: TurnOff():Light@oriole~NoRemoting :: NoRemoting://oriole:/Light [3] (50/4)
```

Figure 8. The *Light* application, before switching.

[6] The RAIC controller is implemented as a generic one. It supports asynchronous invocation model, which is not used in this example. To allow asynchronous calls to retrieve return results, the latest invocation, regardless of its method property, is always added to the invocation history array list. This is why the number shown here is the length of trimmed history plus one (the current call).



Top

Version: 0.0.0.0
PublicKeyToken: (None)
Computer: tiger
Component: LightBlue
Channel: Tcp

RAIC-2a[=i,≈f,≡d,≠s]

Component Type: Stateful
RAIC Level: RAIC-2
Controller Type: RAIC-a
Interface: ≡ Identical
Functionality: ≈ Similar
Domain: ≡ Identical
Snapshot: ≠ Incomparabl
Policy: Memory
MaxRetry: 1

```
Light@oriole~NoRemoting
LightBlue@tiger~Tcp
```

```
RAIC-2a[=i,≈f,≡ d,≠s]: 2 components, 10 calls.
  Policy: enumMem
Light@oriole~NoRemoting  History[2/7] Queue[0]
  URL: NoRemoting://oriole:/Light
  InvocationCount: 7 : [3/3/2002 10:46:40 PM]
  CPU [11%] Mem [25 MB] NetLoad [24749 Bps]
  Charge [14] Intensity [100]
  FailureCount: 0
  Assembly: light, Version=1.2.0.0, Culture=neutral, Pub
LightBlue@tiger~Tcp  History[4/10] Queue[0]
  URL: tcp://tiger:8085/LightBlue
  InvocationCount: 10 : [3/3/2002 10:47:02 PM]
  CPU [20%] Mem [793 MB] NetLoad [44529 Bps]
  Charge [20] Intensity [0]
  FailureCount: 0
  Assembly: light-blue, Version=0.0.0.0, Culture=neutral,
```

The Light Application

Version: 1
Pause (sec): 3
Passes: 50

```
5> 3/3/2002 10:46:33 PM [312 Milliseconds] Called: Light SetIntensity(50)=50
6> 3/3/2002 10:46:37 PM [171 Milliseconds] Called: Light TurnOff()=0
7> 3/3/2002 10:46:40 PM [140 Milliseconds] Called: Light TurnOn()=100
8> 3/3/2002 10:46:50 PM [6640 Milliseconds] Called: Light SetIntensity(50)=50
9> 3/3/2002 10:46:56 PM [3078 Milliseconds] Called: Light TurnOff()=0
10> 3/3/2002 10:47:02 PM [3078 Milliseconds] Called: Light TurnOn()=100
```

```
3/3/2002 10:47:02 PM | Pass 4 | Last Invoked Component: TurnOn():LightBlue@tiger~Tcp :: tcp://tiger:8085/LightBlue [10] {0/20}
```

Figure 9. The *Light* application, after component state recovery.

The policy of the RAIC controller is changed to "Memory" after the seventh call. Because the RAIC feelers report that computer *Tiger* has about 793MB free memory (also shown in the top-right RAIC controller status area), more than the approximately 25MB free memory on computer *Oriole*, the RAIC controller decides to place the next call to the *LightBlue* component on *Tiger.* Since both components are stateful and *LightBlue* is a fresh component at this point, component state recovery is forced to happen. This change of policy has the same effect as a failure on the *Light* component.

During component state recovery, the RAIC controller uses the one invocation item in the trimmed invocation history to restore the state of *LightBlue* and place subsequent calls to *LightBlue*. Figure 9 shows the status after the 10th calls. Note that it shows "History[2/7]" for the *Light* component. Apparently no call is placed to *Light* after the seventh call. "History[4/10]" is shown for the *LightBlue* component. It means the total number of invocations is ten. The number of invocations in the trimmed history is three plus the current call. Since all methods of *LightBlue* are treated as state-changing by default, none of the calls placed to *LightBlue* is trimmed. So, the current trimmed invocation history consists of the three most recent calls to *LightBlue* and one call in *Light's* trimmed history that was used to restore *LightBlue's* state.

4.3 Limitations

The effectiveness of invocation history trimming according to method properties largely depends on the frequency of invocations to state-defining methods as these invocations trims the length of invocation history to one. Therefore, it depends on the type of the component and its usage pattern. In the best case scenario, if all invocations are placed to state-defining methods or state-preserving methods; only one method call is needed to fully restore component states. In the worst case scenario, however, all invocations are placed to state-changing methods; no invocation can be trimmed off. The invocation history could grow to an arbitrary length. In practice, developers can either adjust component design to allow more state-defining calls to appear, or use components with identical snapshots and no external state storage to adopt snapshot-based recovery technique.

Currently, all method properties are manually specified. While it may not be possible to programmatically determine all method properties for all sorts of methods, because component states may be partially stored in non-standard locations such as a system performance counter, source code analysis or even machine code or binary code analysis may help determine or verify method properties.

In some situations, neither the snapshot-based nor the invocation-history-based approach can fully restore component state. For example, suppose a component connects to a database table that does not allow duplicate items. This component has a *AddItem()* method that allows an item to be added to the database. If the component fails after *AddItem("something")* is called, it is not possible to recover it state via a new component by re-invoke past invocations. Because when *AddItem("something")* is called for the second time during the recovery, the database would return an exception "Cannot add item: item exists" instead of a insertion success message. The component would go through total different path. Therefore, in this case, even the exact same invocation sequence may bring the same component to a different state. *Phoenix* addresses some of the problems mentioned here [1].

5. APPLYING RAIC IN ON-LINE UPGRADING

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. 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.

Let us use the same *Light* example. Suppose there are two versions of the *Light* component. The first version 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:

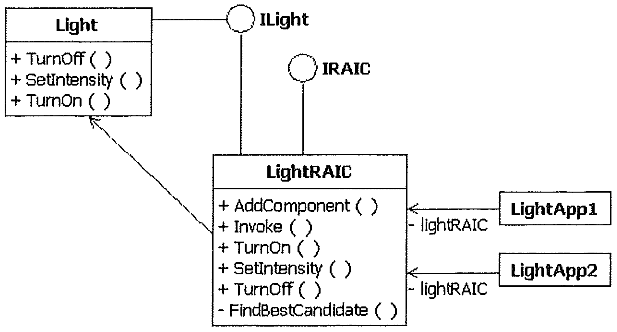


Figure 10. With RAIC, the *Light* applications uses component *LightRAIC* instead of component *Light*.

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 10.

```
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*.

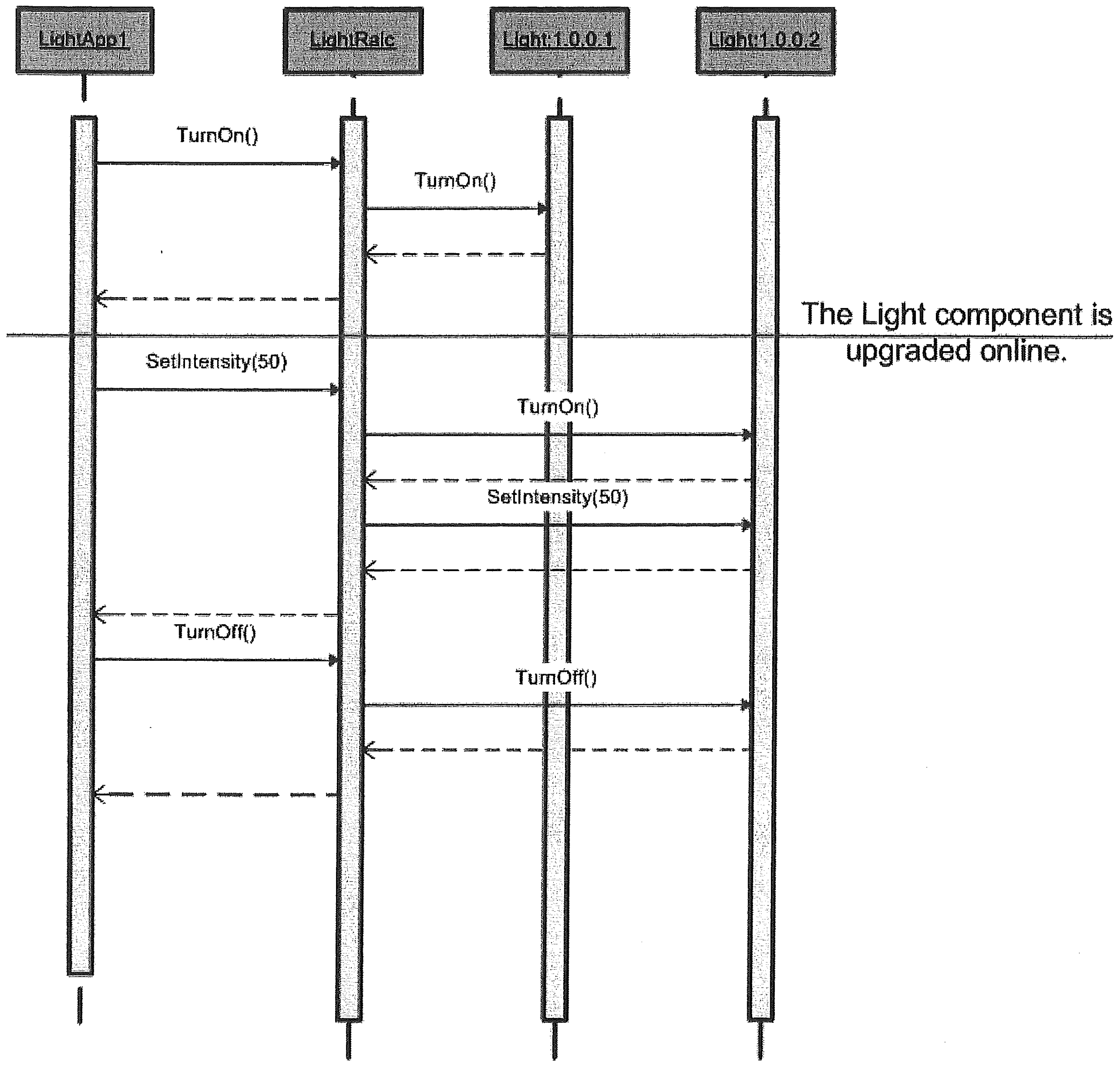


Figure 11. The UML sequence diagram of *LightApp1* during on-line component upgrade.

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, as shown in Figure 11. *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.

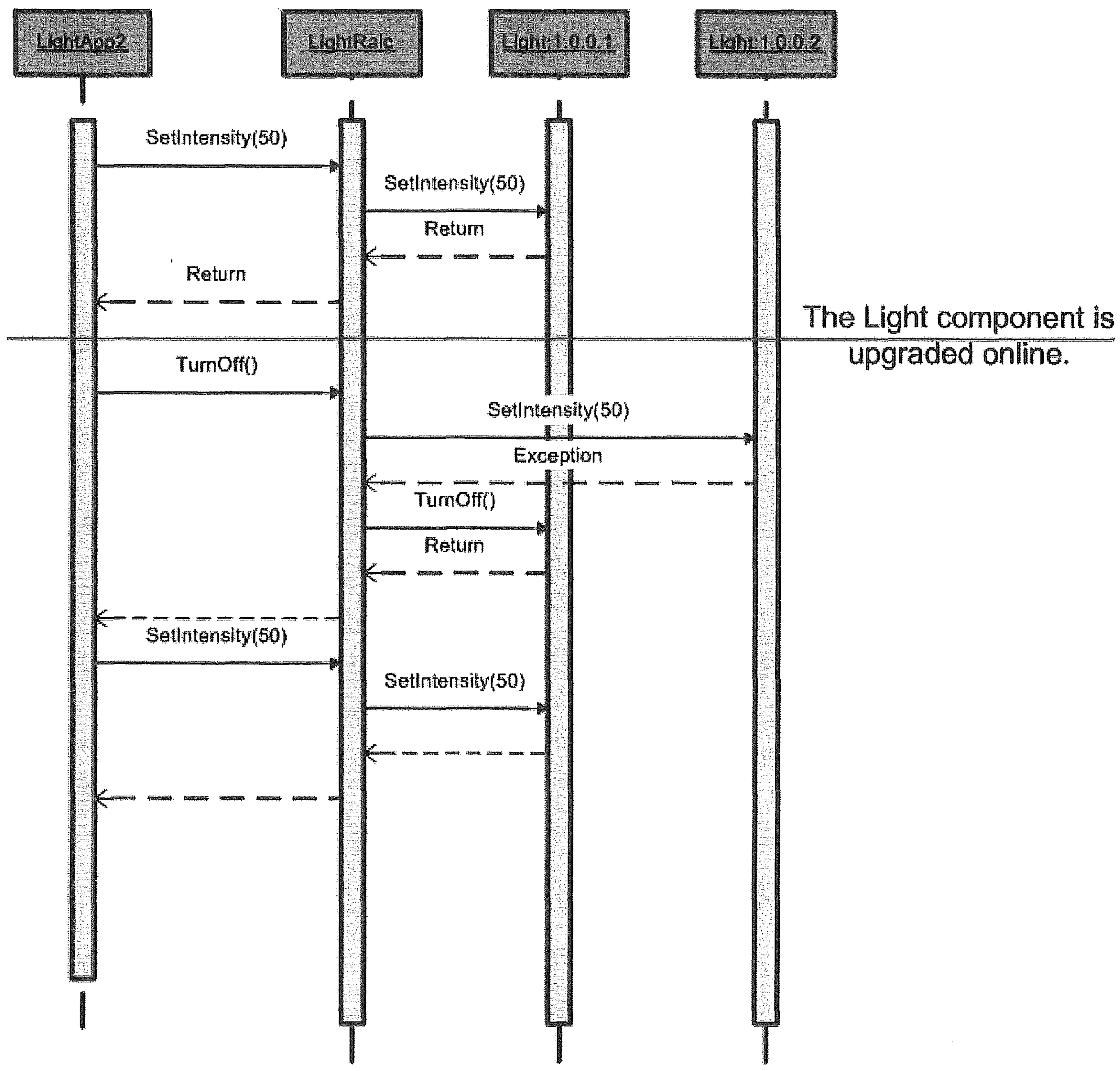


Figure 12. The UML sequence diagram of *LightApp2* during on-line component upgrade.

In *LightApp2*, the RAIC controller also tries to switch to the new component because of the same "latest version first" invocation policy, as shown in Figure 12. 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.

6. RESEARCH DIRECTIONS

6.1 Further Researches in Supporting Technologies

To further enhance RAIC, improvements are needed in the area of just-in-time component testing, component state recovery, component cooperation model, and component relation analysis.

For just-in-time component testing without specification, better heuristics are needed to evaluate test results. For specification-based just-in-time testing, performance needs to be improved.

Both snapshot-based and invocation-history-based component state recovery techniques have drawbacks. More work needed to be done to enable make component state recovery feasible in a broader range of situations. We are currently working on using component dependency information to enhance snapshot-based recovery approach. More works are needed to able to both provide and utilize more accurate and detailed component dependency information [33, 34].

Currently, RAIC assumes that individual components only support the most basic call-and-return invocation model. Some component framework may support more invocation methods, such as queued invocations in queued components, built-in asynchronous invocation [27], or event-based invocation [29]. The coordination model of RAIC needs to be expanded to allow and take advantage of these advanced invocation methods.

In addition, invocation of remote components can take the form of physically retrieving the remote component or its container, such as an assembly [28], to the local machine and then invoking it locally [4, 7, 8]. Migration of code in the form of mobile agents has been possible for a long time. This is now possible using commercially available technology [22, 28]. To incorporate this, the current RAIC invocation model would also need to be expanded.

Component composition languages or architectural description languages enable formalization and analysis of software architectures and architectural styles [19, 24]. For RAIC, instead of a language that describes overall system architecture, a special one focusing on the composition and relations of a redundant component array would be helpful.

Various component relations are the basis for component selection. The key is to obtain accurate and up-to-date component relations with as little human effort as possible. More work needed to be done on component relation analysis to, for example, automate interface relation analysis, or perform computer-aided functionality relation analysis. More relation types may also be needed to provide a more thorough view of component relations.

6.2 Further Researches in RAIC Applications

RAIC can used in many situations. First and foremost, RAIC can be used to group identical software components running on different computer systems to provide higher reliability and availability. But RAIC can also be applied for purposes beyond reliability-through-redundancy. To name just a few, RAIC can be used in performance enhancement, rollbacks in dependable on-line upgrading [15], result refinement, client-side-based distributed load balancing, multi-source file downloading or sharing, or cluster- or grid-based computing, each of which is an interesting and useful area.

To give just one example, if several identical *StockQuote* components are provided by different providers, a `RAIC-1c`$(\equiv_I, \equiv_f, \equiv_d)$ of these component can be used to offer the most up-to-date quote, better than any of the individual components. A preliminary demonstration of stock quotes can be found at [11].

In general, RAIC is a way to manage access to computing and data resources. These resources can be software components. They can also be other things such as hard drives, CPUs, or computer files.

Although the RAIC technology was developed for software components, the underlying principles can be applied to a broader range of components. In fact, RAID, a hardware technology that inspires RAIC, was the application of the same principles on hard drives. When we consider CPUs as components, RAIC can be applied to cluster- or grid-based computing. When we consider computer files as components, RAIC can be used in multi-source file downloading and sharing. Multi-source file downloading and sharing allows the same file to be downloaded from multiple sources and thus increases download efficiency. In peer-to-peer file sharing networks where file availabilities are not guaranteed, RAIC can even potentially help increase file availabilities.

7. CONCLUSIONS

In summary, the RAIC architectural style is a special architectural style designed to take advantage of redundant independent software components to provide reliable services. Component types and relations are analyzed and categorized so that given a particular group of components, developers can follow a systematic approach to determine what the best way is to integrate them. Several RAIC levels and invocation models are defined and explained. Examples are also given on which RAIC level and invocation model are best suited for which component types and relations. In usual cases, developers can simply choose an appropriate pre-defined RAIC level and invocation model to apply to their own RAICs. Computer aided analysis of certain component relations is possible to alleviate developers' workload.

To coordinate components in a RAIC, the RAIC controller needs to know about the status of a component, the status of the computer on which the component is residing, as well as other status information. The just-in-time component testing technique is developed to feed the RAIC controller with component status information. RAIC feelers are designed to feed the RAIC controller with other status information such as CPU workload, available memory, or free network bandwidth. With this information, the RAIC controller can optimize its component selection dynamically. When necessary, a RAIC controller also needs to bring in fresh components to work with applications that have interacted with the RAIC for a while. The component state recovery technique is developed to make this process transparent to applications.

The advent of XML web services and tools that brings XML web services to the masses has made the abundance of free or inexpensive software components an inevitable phenomenon. The goal of RAIC is to make it possible for developers to take advantage of this abundance of software components without full exposure to the complexity of distributed component integration. No matter how inexpensive those third-party remote components will become, more reliable or better performing applications using redundant software components will stay as exceptions instead of norms unless the benefit of adopting them far outweighs the cost of component integration. The RAIC architectural style is a way to lower component integration cost with a clearly defined architectural style, a systematic approach to follow it, and pre-developed techniques or even code templates and modules that supports the style.

Our work on RAIC has merely uncovered a tip of the tremendous potentials of RAIC. As the *Research Directions* chapter points out, more works are needed in both fundamental supporting technologies and application areas.

8. REFERENCES

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

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

[3] T. Berners-Lee, R. Cailliau, A. Luotonen, H. F. Nielsen, and A. Secret, "The World-Wide Web," *Communications of the ACM*, vol. 37, no. 8, pp. 76-82, August, 1994.

[4] L. F. Bic, "Mobile Network Objects," in *Encyclopedia of Electrical and Electronics Engineering*: John Wiley & Sons, Inc., 1998.

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

[6] M. Fowler and K. Scott, *UML Distilled: Applying the Standard Object Modeling Language*, 1999. ISBN: 020165783X.

[7] M. Franz, "Open Standards Beyond Java: On the Future of Mobile Code for the Internet," *Journal of Universal Computer Science*, vol. 4, no. 5, pp. 521-532, May, 1998.

[8] M. Fukuda, L. F. Bic, M. Dillencourt, and F. Merchant, "MESSENGERS: Distributed Programming Using Mobile Autonomous Objects," *Journal of Information Sciences*, 1997.

[9] M. Gudgin, *Essential IDL: Interface Design for COM*: Addison-Wesley Pub Co, 2000. ISBN: 0201615959.

[10] IBM and Microsoft, "Web Services Framework," W3C Workshop on Web Services, San Jose, CA, USA, 2001

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

[12] C. Liu, "Just-In-Time Component Testing and Redundant Arrays of Independent Components," Information and Computer Science, University of California, Irvine, Doctoral Dissertation (in preparation).

[13] 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.

[14] C. Liu and D. J. Richardson, "Specifying Component Method Properties for Component State Recovery in RAIC," Submitted to ICSE2002/CBSE5, March 2002.

[15] 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.

[16] J. Lowy and J. Osborn, *COM and .NET Component Services*: O'Reilly & Associates, 2001. ISBN: 0596001037.

[17] V. Matena and B. Stearns, *Applying Enterprise JavaBeans(TM): Component-Based Development for the J2EE(TM) Platform*: Addison-Wesley Pub Co, 2000. ISBN: 0201702673.

[18] M. D. McIlroy, "'Mass Produced' Software Components," in *Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee*. Garmisch, Germany, 1968, pp. 138-155.

[19] N. Medvidovic and R. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70-93, January, 2000.

[20] Microsoft, *Microsoft Visual C# .NET Language Reference*. Redmond, WA: Microsoft Press, 2002. ISBN: 0-7356-1554-3.

[21] Microsoft, ".NET Platform," 2002, http://www.microsoft.com/net/defined/.

[22] Microsoft, "The Terrarium Distributed Programming Game," 2002, http://www.gotdotnet.com/terrarium/whatis/.

[23] Microsoft, "Visual Studio .NET," 2002, http://msdn.microsoft.com/vstudio/.

[24] E. D. Nitto and D. Rosenblum, "Exploiting ADLs to specify architectural styles induced by middleware infrastructures," Proceedings of the 1999 international conference on Software engineering, Los Angeles, California, United States, 1999

[25] OMG, "OMG Unified Modeling Language Specification, version 1.4," 2001, http://www.omg.org/technology/documents/formal/uml.htm.

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

[27] D. S. Platt, *Understanding COM+*: Microsoft Press, 1999. ISBN: 0735606668.

[28] D. S. Platt and K. Ballinger, *Introducing Microsoft .NET*: Microsoft Press, 2001. ISBN: 073561377X.

[29] D. S. Rosenblum and A. L. Wolf, "A Design Framework for Internet-Scale Event Observation and Notification," Proc. Sixth European Software Engineering Conf./ACM SIGSOFT Fifth Symposium on the Foundations of Software Engineering, 1997, pp. 344-360.

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

[31] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*: Addison-Wesley Pub Co, 1998. ISBN: 0201178885.

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

[33] M. Vieira, M. Dias, and D. J. Richardson, "Describing Dependencies in Component Access Points," Proceedings of The 23rd International Conference on Software Engineering (ICSE'01), Toronto, Canada, 2001, pp. 115-118.

[34] M. Vieira and D. J. Richardson, "Issues in Describing and Analyzing Component Dependencies," Information and Computer Science, University of California at Irvine, Technical Report 01-39, 2001.

[35] C. H. Wittenberg, "Testing Component-Based Software," Presented at International Symposium on Software Testing and Analysis (ISSTA'2000), Portland, Oregon, 2000

[36] M. Wutka, *Using Java 2 Enterprise Edition (J2EE)*: Que, 2001. ISBN: 0789725037.