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

For The ICSE 2002 Workshop on Architecting Dependable Systems
Orlando, Florida, USA

Chang Liu, Debra J. Richardson
Information And Computer Science
University of California, Irvine
{liu,djr}@ics.uci.edu
May 25, 2002
Outline

- RAIC overview.
- An example of dependable application in the RAIC architectural style.
- Conclusions and related work.
RAIC Overview

- Redundant Arrays of Independent Components
- A *redundant software component array* is a group of independent software component that provide identical, similar, or related functions.

[Liu, Richardson (2002)] UCI-ICS-TR-02-09.
The RAIC Architectural Style

- Goal: to reduce component integration cost.

[Liu, Richardson (2002)] Submitted to FSE-10.
RAIC Details

- Component Array Types
  - Component Types
  - Component Relations
- RAIC Controllers
- RAIC Levels
- RAIC Invocation Models
Array Types

- Static
- Dynamic
  - UDDI
  - Jini lookup service
Component Types

- Stateless
- Stateful
  - State-preserving function calls
  - State-changing function calls
  - State-defining function calls
  - State-independent return values
  - State-dependent return values
Component Relations

- Interface relations
- Functionality relations
- Domain relations
- Snapshot relations
- Relations on security, invocation prices, or other aspects
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 Invocation Models

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

[Liu, Richardson (2002)] Preparing for Foclasa (Czech).
using System;
using System.Net;
using System.Threading;
using System.Runtime.Remoting;

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

public class Light : MarshalByRefObject, ILight
{
    [MethodImpl(MethodImplOptions.Property).enumStateDefining]
    public int TurnOn()
    {
        // ...
    }
    [MethodImpl(MethodImplOptions.Property).enumStateDefining]
    public int SetIntensity(int intensity)
    {
        // ...
    }
    [MethodImpl(MethodImplOptions.Property).enumStateDefining]
    public int TurnOff()
    {
        // ...
    }
    // ...
}
using System;
using System.Threading;

public class LightApp
{
    public static void Main(string[] args)
    {
        int pause_in_seconds = 3;
        int number_of_passes = 50;

        Light light = new Light();

        for (int i = 1; i <= number_of_passes; i++)
        {
            light.TurnOn();
            Thread.Sleep(pause_in_seconds * 1000);
            light.SetIntensity(50);
            Thread.Sleep(pause_in_seconds * 1000);
            light.TurnOff();
        }
    }
}
```csharp
using System;
using System.Threading;

public class LightApp
{
    public static void Main(string[] args)
    {
        int pause_in_seconds = 3;
        int number_of_passes = 50;

        Light light = new Light();

        for (int i = 1; i <= number_of_passes; 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 Light Components With Versions

- Light version 1.0.0.1
  - Allows arbitrary calls to all three methods
- Light version 1.0.0.2
  - Must call `TurnOn()` before calling `SetIntensity()` or `TurnOff()`
  - Cannot call `TurnOff()` if the light is already off.
LightApp1, LightApp2, the Light Components, and LightRAIC
LightApp1 enjoys Light:1.0.0.2

The Light component is upgraded online.
LightApp2 reverts to Light:1.0.0.1

The Light component is upgraded online.
Just-In-Time Component Testing

- JIT component testing versus traditional software testing.
- JIT component testing versus perpetual testing.
- JIT component testing versus self-checking software or components.
  [Yau, S. S. and R. C. Cheung (1975)]
  [Liu, C. and D. J. Richardson (1998)]
JIT Testing VS Traditional Testing

- Happens even after application deployment.
- Uses heuristics and other means in place of traditional test oracles.
- Uses mostly live input data.
- Efficiency extremely important for predetermined test inputs.
- Should not change component states.
JIT Testing VS Perpetual Testing

- Perpetual testing is optional and removable.
- JIT testing in conjunction with the RAIC controller is an integral part of the final product.
JIT Testing VS Self-Checking Components

- JIT testing mechanisms are part of the RAIC controller.
- Self-checking mechanisms are part of the component.
Component State Recovery

- Snapshot-based
- Invocation-history-based
  - Method properties
  - Invocation history trimming

- Could be enhanced with component dependency information

[Liu, Richardson (2002)] ICSE02/CBSE5.
Related Work

- Barga and Lomet: Phoenix
- Cook and Dage: Hercules
- S. S. Yau and R. C. Cheung: Self-checking software
Conclusions

- Dependability-through-redundancy can be achieved by adopting the RAIC architecture style.
- Just-in-time component testing and component state recovery techniques support RAIC to achieve the above goal.