A Study of Execution Environments for Software Components

Kung-Kiu Lau and Vladyslav Ukis

School of Computer Science, The University of Manchester
Manchester M13 9PL, United Kingdom
{kung-kiu,vukis}@cs.man.ac.uk

Abstract. Software components are deployed into an execution environment before runtime. The execution environment influences the runtime execution of a component. Therefore, it is important to study existing execution environments for components and learn how they influence components' runtime execution. In this paper, we undertake such a study. We show that deploying components into different execution environments may incur runtime conflicts, which, however, can be detected before runtime.

1 Introduction

The execution environment for a software component controls the component's lifecycle, beginning with component instantiation through runtime management to shutdown. Currently, there are two widely used execution environments for components [13]: desktop and web. The problem we set out to investigate in this paper is: Given a component, how can we determine which execution environments it can be deployed into?

This problem becomes more acute when components are developed by component developers but used by system developers independently [2]. In such a situation, on the one hand, the component developers develop components without the knowledge of the execution environments they will be deployed into; and, on the other hand, the system developers deploy the components into specific execution environments not knowing if the chosen components are suitable for these environments. In this situation, it is important for the system developers to be able to check if a component is suitable for running in a particular execution environment. Current component models do not allow for this kind of checking. In this paper, we make a study of execution environments to enable that.

The work in this paper builds on the work we presented in [9]. In that paper we considered how deployment contracts for components can be defined, and used to check mutual compatibility between components. In this paper, we consider how the deployment contracts for components can be used to check a component's compatibility with its execution environment. Preliminary checking of this kind was introduced in [9]. In this paper we undertake a thorough study of what is involved when checking a component's compatibility with its execution environment. The results of the study enable us to extend our Deployment Contracts Analyser Tool [9] to automatically check a component's compatibility with its execution environment.

H.W. Schmidt et al. (Eds.): CBSE 2007, LNCS 4608, pp. 107–123, 2007.

2 Execution Environments

In this section we study widely used execution environments for components. In general, when we talk about the execution environment in this paper we do not mean component containers as they are not necessarily present in current component models. In fact, despite widely used component models such as EJB [6] and JavaBeans [3] employing containers for component execution, the majority of current component models do not use containers [12].

The distinction between a container and an underlying component execution environment can be seen clearly in Fig. 1.

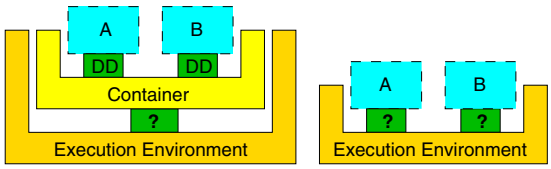

Fig. 1. Component deployment in an execution environment with and without container

On the left hand side in Fig. 1, components A and B are deployed into a container. This is typically done using so-called Deployment Descriptors (DD) [1] that describe the relationship between a component and the container. The container itself, however, is deployed into an execution environment in our sense. The question mark between the container and the execution environment denotes the question of compatibility of the container with the environment.

On the right hand side in Fig. 1, components A and B are deployed directly into the execution environment (without the container). That is, the execution environment is there regardless of the existence of container and impacts component runtime execution. The question marks between the components and the execution environment denotes the uncertainty of compatibility of the components with the environment.

In order to investigate execution environments that are widely used, we turned our attention to platforms widely used for software development in practice [5]. These are the J2EE [14] and .NET [16] platforms. The platforms allow for the development of software that can be deployed into two different execution environments: desktop and web (server). These execution environments are ubiquitous today [13]. In this paper, we want to undertake an analysis of them to find out their characteristics and differences that should lead us to the answer to the question we posed in section 1: Given a component, how can we determine which execution environments it can be deployed into?

The analysis we want to undertake in this paper will be implemented in a tool that will automatically check a component's suitability to run in a particular execution environment. In order to implement the tool we need to know the properties of interest of current component execution environments first.

2.1 Properties of Interest

Let us consider a binary component deployed into an execution environment. Such a component is shown in Fig. 2. The component is ready made and ready for execution. It may access some resources from the execution environment in order to be able to run [9]. Furthermore, it may have some specific threading model implemented inside [9].

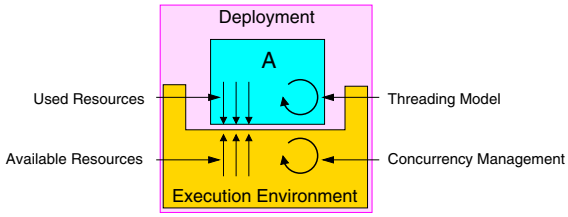

Fig. 2. Properties of interest in an execution environment

Therefore, in order to find out how a component execution environment can influence the runtime execution of the component, we have to find out which resources can be available as well as what the concurrency management is in the execution environment, in general.

For instance, in Fig. 2 the component A at deployment time has some environmental dependencies and a threading model, and is deployed into an execution environment. In order to find out how the execution environment influences runtime execution of the component A, we have to know the resources offered by the environment and the concurrency management of the environment.

Additionally, we need to know how the execution environment manages transient state [15,7] of the component. The transient state of the component, unlike persistent state, can exist only for the lifetime of a component instance. It is shared by requests to the instance, and disappears when the instance vanishes. Transient state is inherently connected to concurrency because it is the state which is shared by multiple threads operating concurrently in the component and has to be protected from corruption by thread synchronisation primitives.

In summary, we need to know the following properties of an execution environment to be able to assess its impact on the component executing in it: *Transient State Management*, *Concurrency Management*, *Availability of Resources*. In the following section we investigate these properties for the desktop environment based on an analysis of the J2EE and .NET platforms.

3 Desktop Execution Environment

The desktop environment is an execution environment for systems deployed on a desktop. Known examples of desktop applications are Adobe Acrobat Reader, Ghost Viewer etc. They provide a Graphical User Interface (GUI) to enable the user to interact with the system. Moreover, small programs like UNIX commands, e.g. ls or ps, also

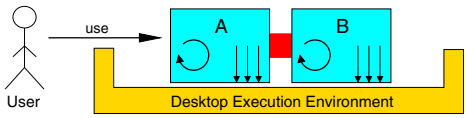

Fig. 3. A system deployed into a desktop environment

represent examples of systems deployed into the desktop environment. These systems do not provide a GUI interface but are launched using a command shell.

A typical user interaction with systems deployed into the desktop environment is shown in Fig. 3.

A system consisting of the components A and B assembled together is deployed into the desktop environment. There is a *single user* interacting with the system. It is a distinguishing characteristic of the desktop environment that it enables a single user to interact with a system instance. (We consider a system to reside on one machine and not to be distributed over several computers)

In the following, we consider Transient State Management, Concurrency Management and Availability of Resources, the properties we identified to be important in Section 2.1, in the desktop environment.

3.1 Transient State Management

If a system is deployed into the desktop environment, it is instantiated by it on the system startup and destroyed on the system shutdown. If in the meantime, i.e. in the time when requests are placed to the system, a component accumulates state,[1] the desktop environment does not interfere.

This is shown in Fig. 4. The component A is deployed into the desktop environment.

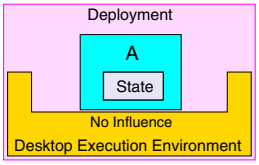

Fig. 4. Transient state management in the desktop environment

It holds transient state, and therefore can be referred to as stateful component. The desktop execution environment does not have an influence on the component A's state.

3.2 Concurrency Management

In general, it is possible to deploy both single-threaded and multithreaded systems into the desktop environment. If the system is single-threaded, it uses the main thread, provided by the desktop environment, to process requests. If the system is multithreaded,

[1] We mean "transient state" when referring to "state" in this paper.

it spawns other threads in addition to the main thread for request processing. The distinguishing characteristic of the concurrency management in the desktop environment is that *the main thread is guaranteed to be the same* for every request placed to a system instance during its lifetime.

This is illustrated in Fig. 5.

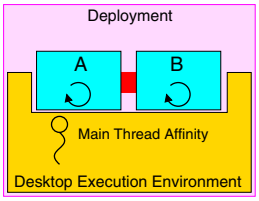

Fig. 5. Concurrency management in the desktop environment

The system AB consists of two components A and B. Each of the components has its own threading model. Depending on the threading model of either component, the system may be single- or multithreaded. In any case, the system makes use of the main thread provided by the desktop environment. It is ensured by the desktop environment for the system AB that the main thread remains the same for the lifetime of a system instance.

This has implications on the elements in components, which require thread affine access. In the desktop environment, such elements can be safely used from the main thread since it is guaranteed to be the same for all requests placed to the system.

3.3 Resource Availability

An execution environment can contain a set of resources that can be used by components deployed in it. In order to find out which resources can be found in the set, we studied the APIs of the J2EE and .NET frameworks that provide access to them. The results of our investigation are shown in Fig. 6. The resource set is as applicable to the desktop environment as it is to the web environment (see later).

The categories of resources in Fig. 6 are self-explanatory. Detailed descriptions of the categories can be found in [10]. Each resource from the categories may or may not be available in a particular execution environment. Therefore, when deploying a component into an execution environment, it is necessary to check if the resources required by the component are available in the execution environment.

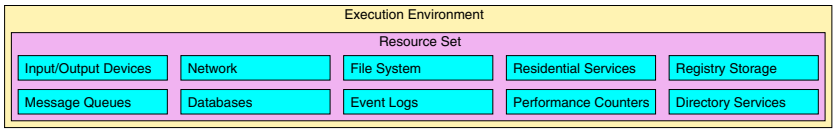

Fig. 6. Resources in an execution environment

The resources we discovered are restricted to and complete with respect to the APIs of the J2EE and .NET frameworks. However, the comprehensiveness and wide applicability of the investigated frameworks should imply the same for the derived results.

3.4 Deploying Components into Desktop Execution Environment

In this section, we show an example of components deployed into the desktop environment.

To show the example, we make use of a tool we have developed and initially presented in [9,8]. The tool is the Deployment Contract Analyser (DCA). One of the purposes of the tool is to enable automated checking of compatibility of a component with the execution environment it is deployed into. The checking is done statically on binary components when they are at deployment time (as opposed to instantiated components at runtime).

The DCA implements the properties of the desktop and web execution environments we present in this paper. It takes as input binary components that are augmented with deployment contracts [9,10]. A deployment contract of a component indicates the component's threading model and usage of resources in the execution environment. It is manually defined by component developer. Knowing a component's deployment contract and its execution environment (specified using a graphical tool that can be seen in [8]), the DCA can detect conflicts of the component with the execution environment and present them to the user. Consider the example shown in Fig. 7.

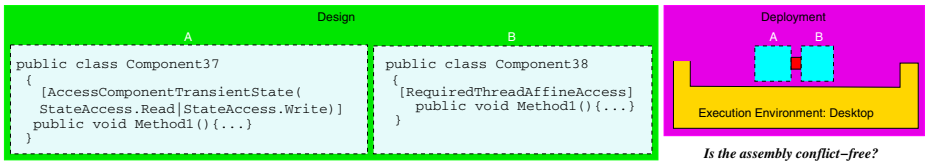

Fig. 7. Example 1

Component A is designed in a way that its method "A.Method1" accesses component's transient state in Read/Write mode. Component B is designed in a way that its method "B.Method1" requires thread affine access. This can be seen from their deployment contracts in Fig. 7.

Suppose at deployment time, a system AB is created. In the system, components' methods are connected so that there is one connection: *Connection 1: method "A.Method1" is invoked prior to the method "B.Method1"*. The system AB is deployed into the desktop environment. Resources available in the execution environment are irrelevant in this case.

Deployment contracts analysis performed by the DCA for the system AB is shown in Fig. 8.

Neither for the component A (Component37 in Fig. 8), nor for the component B (Component38 in Fig. 8) has the DCA found any problem. Therefore, the system AB is conflict-free and can execute safely at runtime.

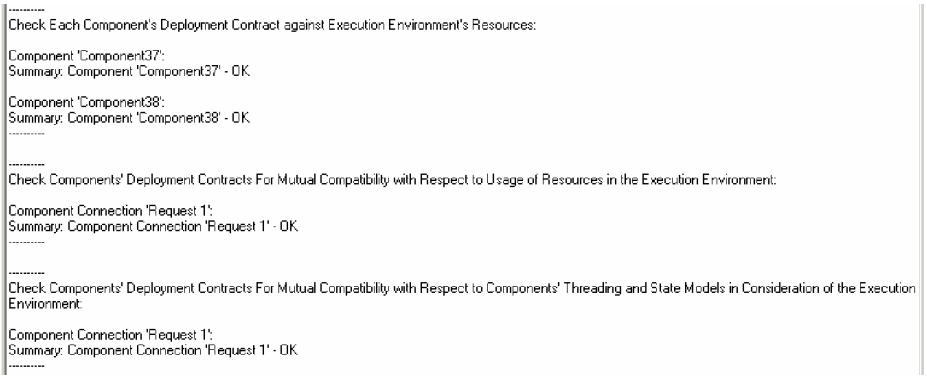

Fig. 8. Deployment contracts analysis for the Example 3

In the following section we consider properties of the web execution environment and try deploying the assembly AB in it.

4 Web Execution Environment

The web execution environment is an environment for systems deployed on a web server. Known examples of such systems are web portals like Amazon or search engines like Google.

A typical user interaction with such systems is shown in Fig. 9.

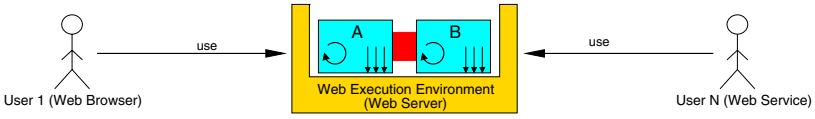

Fig. 9. System in web execution environment

A system AB consisting of two components A and B is deployed into the web environment. There are *many users*, possibly simultaneously, interacting with the system. It is a distinguishing characteristic of the web environment that it enables many users to interact with a system instance. A user typically uses a web browser to interact with a system deployed into the web environment, but it can also be accessed by web services (Fig. 9).

In the following, we consider Transient State Management and Concurrency Management, the properties we identified to be important in Section 2.1, in the web environment.

4.1 Transient State Management

The distinguishing characteristic of the web environment is that the user interacts with the system on the web server using Hyper Text Transfer Protocol (HTTP) [4]. The

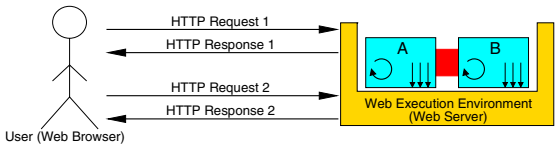

Fig. 10. Request-Response interaction mode using HTTP protocol

HTTP Protocol implements an interaction mode referred to as Request-Response mode [13]. This is shown in Fig. 10.

The user places an HTTP Request 1 to the system and receives a result. Subsequently, the user places another HTTP Request 2 to the system and receives a result to it. An important characteristic of the HTTP protocol is that it does not maintain any transient state between HTTP requests. It is therefore referred to as a stateless protocol. In fact, an HTTP request does not maintain any relationship to previous requests issued to the system on the web server. For instance, in the example from Fig. 10, at the beginning of the interaction, the HTTP Request 1 is sent to the web server, processed by the system and a result is returned to the user. The user is now completely disconnected from the web server. The web server, in turn, does not maintain any transient state about the Request 1 placed by a user. It, indeed, has "forgotten" about it. Now, the user places another request to the system, HTTP Request 2. This request does not maintain any relationship to Request 1 and is processed by the system without any transient state related to Request 1.

However, why did the web server "forget" about Request 1? The truth is that the HTTP protocol's Request-Response interaction mode operates in a way that for each HTTP request the client establishes a new connection to the web server. The web server, in turn, creates a new system instance. The newly created system instance processes the request and generates a result. The web server *destroys* the system instance and sends the result back to the client. On the next request, the chain of the events is repeated etc.

This is exemplified in Fig. 11 for the two HTTP requests we considered before.

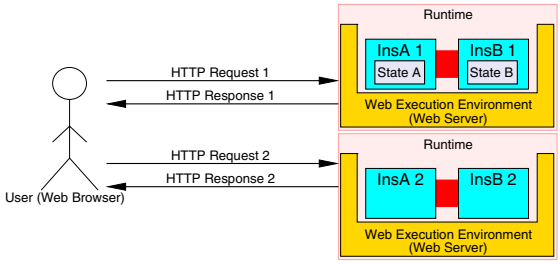

Fig. 11. Request-Response interaction mode and system instantiation

At the time when the user places Request 1 to the system AB, a system instance actually does not exist. It is only created by the web server when the request arrives there. The newly created system instance processes the request and generates a result. Subsequently, the web server destroys the system instance and sends the result back

to the user. The user is now not only disconnected from the system, which processed Request 1, but the system instance actually does not physically exist any more. The web server does not maintain any information about Request 1 either. Now, the user places another request, Request 2, to the system AB. Again, no system instance exists till the request arrives at the web server, which creates a new system instance. The system instance processes the Request 2 and generates a result. After that the web server destroys the instance and returns the result. Again, no system instance exists till another request to the system AB hits the web server.

In such an environment, any component transient state cannot be retained between requests to the system. For instance, assuming components in Fig. 11 hold transient state. The component A holds 'State A', whereas the component B hold 'State B'. These transient states exist only for the lifetime of a system instance. Since the web server destroys the instance at the end of a request processing and creates another one at the beginning of the processing of the next request, the states 'State A' and 'State B' exist only during processing of Request 1 and do not exist during processing of Request 2.

Components that do not hold state, i.e. *stateless components*, can be deployed into and smoothly run in the web environment. They are immune to instance creation and destruction by the web server since they process each request individually without reliance on state information from previous requests. They, therefore, represent ideal candidates for the web environment. By contrast, components that do hold state, *stateful components*, pose a problem in the web environment since it, unlike the desktop environment, does not retain component state in between requests to the system.

In order to deal with the statelessness of the web environment, different technologies for web application development have been put forward. For instance, the J2EE Platform contains Java Server Pages (JSP) technology for web application development. Furthermore, .NET platform has Active Server Pages (ASP.NET) technology for the same purpose. These technologies allow for state retention in components on the web server. More traditional techniques for web application development such as Common Gateway Interface (CGI) scriptsfollow the Request-Response model of the HTTP protocol explained above and do not retain state on the web server side. JSP and ASP.NET are technologies that are representative and widely used in practice. We undertook a thorough analysis of them in [11]. For lack of space, we only present the essential findings of the analysis here.

4.2 Transient State Management in Java Server Pages

JSP from J2EE platform is a technology for building web applications. It is based on Java Servlet Technology, which is also part of the J2EE platform.

Java Servlet Technology provides a special container running on the web server. The container hosts and manages components referred to as "Servlets". The servlet container prevents servlets from being created and destroyed by the web server on each request processing cycle. It ensures that there is always one instance of component system to process all requests from all users. This is illustrated in Fig. 12.

Users place, possibly simultaneous, requests to the system AB. A system instance is running in the servlet container, which, in turn, is running on the web server. The container makes sure that the web server does not destroy the instance at the end of

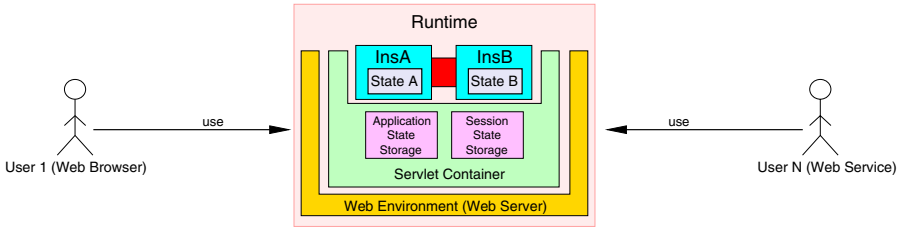

Fig. 12. Servlet Container

each request processing. Therefore, the components A and B can safely hold state and make use of it for request processing.

In addition, the servlet container offers two types of storage to component developers. That is, *application and session state storage*. On the one hand, application state storage can store information for the lifetime of a system instance. Moreover, it is shared among all users of a web application. On the other hand, session state storage stores information for the lifetime of a user or browser session. The session state storage is therefore user-specific. A user, or browser, session embraces a specific number of requests from a web browser instance to the system on the web server.

If a system in the web environment is going to have a large number of concurrent users, it is inefficient to have a single system instance process all user requests. Therefore, the Java Servlet Technology provides another mode of system instantiation referred to as Single Thread Model. With this model, the servlet container instantiates not only one but a fixed, configurable, number of more than one system instances that process user requests. If the user requests are concurrent, the load is distributed among available system instances.

With the Single Thread Model, the container guarantees that a system instance is accessed by one and only one thread per request, and not concurrently. However, the container does not guarantee that all requests from a user will be processed by the same system instance. This makes state management in the system complicated. Indeed, a user request may be processed by a system instance. The system instance may accumulate some transient state during the request. Then, another request from the same user may be processed by another system instance, whose component instances do not hold the data created on the previous request. It becomes even more complicated to hold some global data in the system. However, for all these cases, the usage of application and session state storage provides a solution to cope with state retention issues.

Now we consider how state is managed with another technology – ASP.NET.

4.3 Transient State Management in Active Server Pages

Active Server Pages (ASP.NET) from .NET platform is another technology for developing web applications. It provides a special environment referred to as ASP.NET environment. The ASP.NET environment runs on a web server and hosts .NET components. This is shown in Fig. 13.

Components A and B are running inside the ASP.NET environment, which in turn runs on a web server. The default behaviour of the ASP.NET environment, unlike servlet

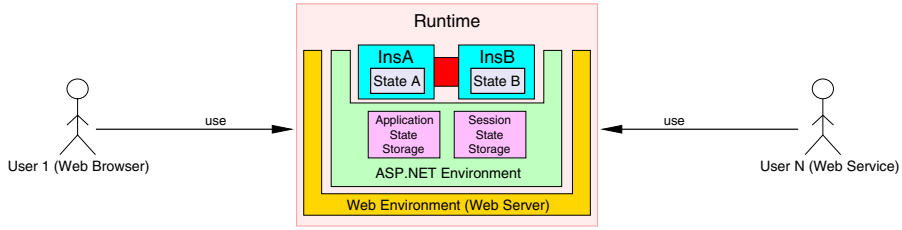

Fig. 13. ASP.NET Environment

container, with respect to system instantiation is that it follows the Request-Response model of HTTP protocol. That is, the ASP.NET environment creates and destroys a system on each request processing cycle. However, like the servlet container, it offers application and session state storage to component developers to deal with state retention issues.

Overall, the web environment, unlike the desktop environment we learnt in Section 3, has great influence on state inside components of a system. The influence depends on the technology used as summarised in Table 1.

Table 1. Transient state management in the web environment

	Request–Response Mode (CGI)	ASP.NET	JSP
System transient state management	System state is not retained among requests to the system.	1) Default: System state is not retained. 2) Application and Session state storage available.	1) Default: state is retained 2) Application and Session state storage available. 3) Single Thread Model: state is not retained.

4.4 Concurrency Management

A system deployed on a web server is exposed to, theoretically, an unlimited number of concurrent users. Therefore, concurrency issues are inherent in such an environment. A web server spawns a thread for every request it receives. Following this, we can conclude that in the web environment *the main thread is not guaranteed to be the same* for every request placed to a system instance during its lifetime. The handling of a request depends on the technology used for web application development, i.e. CGI, JSP or ASP.NET. Moreover, it depends on the way a technology is used. In particular, concurrency management in the web environment depends on the chosen system instantiation mode. Below, we undertake a brief categorisation of system instantiation modes in the web environment. Full details can be found in [11].

4.5 System Instantiation Modes

With CGI, Request-Response mode imposed by the HTTP protocol is used. That is, a system instance is created at the beginning of a request processing and destroyed

after the request has been processed by the instance. In other words, in this mode *a system instance per request* is created. This is also the default mode of operation with ASP.NET.

Moreover, with ASP.NET, it is possible to create *a system instance per user session* by using the session state storage. That is, a system instance is created on the first request from a specific user and is put into the session state storage. On all subsequent requests from the same user, the system is retrieved from the session state storage and used for request processing.

Furthermore, with JSP, by default, the servlet container creates a system instance which processes all requests to the system. Therefore, in this mode there is *a system instance for all requests*. The same behaviour can be achieved with ASP.NET by using the application state storage. That is, a system instance is created on the first request and is put into the application state storage. On all subsequent requests, the system is retrieved from the application state storage and used.

Finally, with the servlet container using Single Thread Model, a pool of system instances is created. The container guarantees that a system instance is accessed by only one thread during a request processing. In other words, in this mode *a pool of synchronised system instances for all requests* is created.

In summary, we can identify the following four system instantiation modes in the web environment: *A System Instance Per Request*, *A System Instance Per User Session*, *A System Instance For All Requests*, *A Pool Of Synchronised System Instances For All Requests*. They correspond to the individual technologies for building web applications in the way shown in Table 2.

Table 2. System instantiation modes in the web environment with corresponding technologies for building web applications

	System instance per request	System instance per user session	System instance for all requests	Pool of synchronised system instances for all requests
CGI	default	not available	not available	not available
JSP	not available	not available	default	Single Thread Model
ASP.NET	default	use of session state storage required	use of application state storage required	not available

Observations on Concurrency Management in Web Environment. An inherent property of the web environment is that it enables a system to be accessed by, potentially concurrent, multiple users. As a corollary, the web environment itself may impose threading issues on a system instance by exposing it to multiple threads. Moreover, the concurrency management in the web environment depends heavily on the system instantiation mode used. During the analysis of the concurrency management in the web environment [11], we encountered the following three problems:

State Corruption Problem – This problem occurs when multiple threads concurrently access state of a component, which is not protected by a thread synchronisation primitive.

Lack of Thread Affinity Problem – This problem occurs when a component containing thread affine elements is not always accessed by one and the same thread.

Shared Statics and Singletons Problem – For this problem to occur, the following conditions must be met: (i) A component in a system must contain static variables (statics) or singletons, (ii) The system must be instantiated more than once in an operating system process, (iii) Created system instances must be executed concurrently to each other by the execution environment. This means that a system instance itself may not be executed concurrently but process requests sequentially. However, other system instances are also executed at the same time, (iv) The statics or singletons are not used by a thread-safe component part.

In this case, the statics or singletons are shared by component instances that are executed concurrently. Since they are unprotected from concurrent access by multiple threads, state corruption in them will occur. This problem, indeed, boils down to the state corruption problem described above. However, since it has far more conditions to emerge we can treat it as a problem in its own right.

With these problems defined, in the Table 3 we show occurrences of them depending on the system instantiation mode used.

Table 3. System transient state and concurrency management depending on system instantiation mode in the web environment

	System instance per request	System instance per user session	System instance for all requests	Pool of synchronised system instances for all req.
Concurrency management	A system instance is accessed by one thread	A system instance can be accessed by mult. threads sequentially	Concurrent access of a system instance by multiple thr.	A system instance can be accessed by multiple threads sequentially
System transient state managment	System state is not retained among requests	System state is retained during a user session	System state is retained among all requests	System state is not retained among requests
State Corr. Problem	No	No	Yes	No
Lack of Thread Affinity Problem	No	Yes	Yes	Yes
Shared Statics and Singletons Problem	Yes	Yes	No	Yes

With the system instantiation mode “system instance per request”, a system instance is accessed by only one thread provided by the web environment. Therefore, the lack of thread affinity problem cannot occur here. Furthermore, the system state is not retained among requests to the system. Therefore, in this case the state corruption problem cannot occur. However, the shared statics and singletons problem may occur here if different users place their requests to different system instances concurrently.

Furthermore, with the system instantiation mode “system instance per user session”, a system instance can be accessed by multiple threads but only sequentially. System state is retained during a user session. However, since no threads operate concurrently

in a system instance, no state corruption problem will occur. On the other hand, due to the access of the system by multiple threads, lack of thread affinity problem may occur. Moreover, since in this mode a system instance is created for each user and they all reside in a single operating system process, shared statics and singletons problem may occur.

Additionally, with the system instantiation mode "system instance for all requests", a system can be accessed concurrently by multiple threads. System state is retained among requests. Therefore, the state corruption problem may occur here. Furthermore, since the system is not always accessed by one and the same thread, lack of thread affinity problem may occur. As to the shared statics and singletons problem, it cannot occur here since there is only one system with this system instantiation mode.

Finally, with the system instantiation mode "pool of synchronised system instances for all requests", a system instance can be accessed by multiple threads but only sequentially. System state is not retained among requests. Following this, there is also no state corruption problem. Moreover, there is no guarantee of the thread affinity of the main thread for a system instance. Therefore, the lack of thread affinity problem may occur. Since there are several system instances all residing in the web server process, the shared statics and singletons problem may occur here as well.

4.6 Deploying Components into Web Execution Environment

In this section we show how the system AB from the Section 3.4 can be deployed into the web execution environment.

Example 1. Consider the system AB from the Section 3.4 deployed into the web environment with the system instantiation mode 'system instance per user session' (Fig. 14).

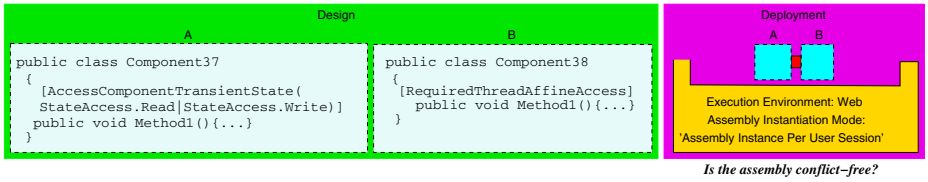

Fig. 14. Example 1

Deployment contracts analysis performed by the DCA for the system AB is shown in Fig. 15.

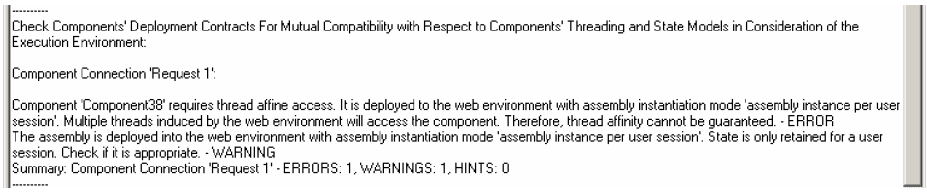

Fig. 15. Deployment contracts analysis for the Example 1

For component B (Component38 in Fig. 15), the DCA finds out that the requirement of the method “A.Method1” cannot be satisfied due to the concurrency management of the environment the system AB is deployed to, namely absence of thread affinity of the main thread.

Moreover, in this environment, state is only retained for the duration of a user session. This is relevant for the component A (Component37 in Fig. 15). Assume that this is acceptable for the system the system developer is building.

Deployment contracts analysis of the system AB has shown 1 error. Therefore, the system AB is not conflict-free and cannot execute safely at runtime. Component B has to be replaced by another one in the system AB.

Example 2. Consider the system from Section 3.4 deployed into the web environment with the system instantiation mode ‘system instance for all requests’. Resource available in the execution environment are irrelevant in this case (Fig. 16).

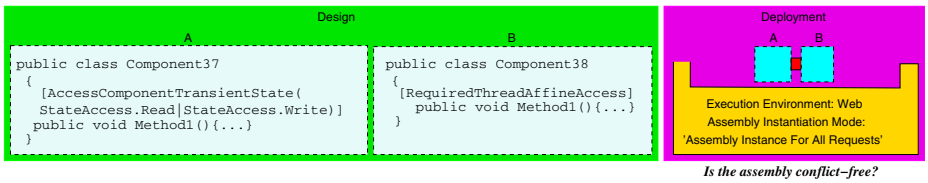

Fig. 16. Example 2

Deployment contracts analysis performed by the DCA for the system AB is shown in Fig. 17.

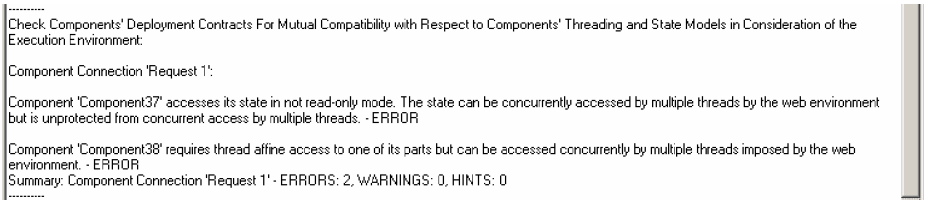

Fig. 17. Deployment contracts analysis for the Example 2

For component A (Component37 in Fig. 17), the DCA finds out that component’s state will be accessed concurrently in the system’s execution environment. Since the state is unprotected from concurrent access by multiple threads, state corruption problem will occur.

For component B (Component38 in Fig. 17), the DCA finds out that the requirement of the method “A.Method1” cannot be satisfied due to the concurrency management of the environment the system AB is deployed to, namely concurrent access of the system by multiple threads.

Deployment contracts analysis of the system AB has shown 2 errors. Therefore, the system AB is not conflict-free and cannot execute safely at runtime.

5 Discussion and Concluding Remarks

As we have seen in Sections 3 and 4, desktop and web environments differ substantially with respect to the management of state and concurrency of systems deployed in them. The major differences are: *system instantiation*, *handling of component state*, *allocation of the main thread for a system instance* and *the way a system instance is exposed to the users*.

Table 4 shows how these criteria are handled in the desktop and web execution environment.

Table 4. Comparing the properties of the desktop and web execution environment

	Desktop Environment	Web Environment
System instantiation	Once for all requests	Depends on technology
State retention issues	No	Yes
Main thread affinity	Yes	No
Exposure to multiple threads	No	Yes

In the desktop environment, system instantiation is done at system start. The created system instance processes all requests to the system. By contrast, in the web environment, the used technology greatly influences the handling of system instantiation. It can range from a system instance per request through a system instance for all requests.

Furthermore, in the desktop environment, there are no state retention issues due to the fact that a single system instance processes all requests to the system. Contrary, in the web environment, it requires an effort to retain state among requests to the system since it is by far not a common case that a single system instance processes all requests. Rather, different technologies deal with system instantiation differently making it complicated to maintain state among requests.

Moreover, in the desktop environment, the main thread for a system instance is guaranteed to be always one and the same. However, this does not hold true for the web environment. Again, the issue of main thread affinity depends on the technology used for building web applications.

Additionally, in the desktop environment, a system instance is never exposed to multiple threads induced by the environment. Unlike the desktop environment, the web environment makes it possible for a system instance to be concurrently accessed by multiple users, thus exposing the instance to the concurrent access of multiple threads.

The main contribution of this paper is the analysis of what is involved in deploying a component into different execution environments. The study of the execution environments undertaken in this paper has enabled us to extend our DCA tool to perform compatibility checks of components with their execution environments.

As a matter of fact, such checks are not performed by any of current component models [12]. However, in sections 3.4 and 4.6 we showed that neglecting these checks

leaves systems go into runtime with conflicts that impair runtime system execution. We also showed how these conflicts can be checked at deployment time, before runtime, using the Deployment Contracts Analyser [8] tool we have developed and extended since its initial presentation in [9].

Our future work will consist in investigating the OSGI framework to extract more expressive deployment contracts for components. Furthermore, we will implement a tool that help the component developer apply the contracts to their components.

Finally, full details of the study of component execution environments and more examples can be found in [8].

References

1. Bachmann, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R., Wallnau, K.: Volume II: Technical Concepts of Component-Based Software Engineering. Technical Report CMU/SEI-2000-TR-008, Carnegie Melon University (2000)
2. Crnkovic, I., Schmidt, H.W., Stafford, J.A., Wallnau, K.C.: Automated Component-Based Software Engineering. Journal of Systems and Software 74(1), 1–3 (2005)
3. Englander, R.: Developing Java Beans. O'Reilly & Associates (1997)
4. Fielding, R., Gettys, J., Mogul, J., Nielsen, H., Berners-Lee, T.: Hypertext transfer protocol HTTP/1.1, 1997. RFC 2068 (1997)
5. Fowler, M., Box, D., Hejlsberg, A., Knight, A., High, R., Crupi, J.: The great J2EE vs. Microsoft .NET shootout. In: OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pp. 143–144. ACM Press, New York (2004)
6. Haefel, R.M.: Enterprise Java Beans, 4th edn. O'Reilly (2004)
7. Heineman, G.T., Councill, W.T. (eds.): Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley, Reading (2001)
8. Lau, K.-K., Ukis, V.: A Reasoning Framework for Deployment Contracts Analysis. Preprint 37, School of Computer Science, The University of Manchester, Manchester, M13 9PL, UK ISSN 1361 - 6161 (June (2006)
9. Lau, K.-K., Ukis, V.: Defining and Checking Deployment Contracts for Software Components. In: Gorton, I., Heineman, G.T., Crnkovic, I., Schmidt, H.W., Stafford, J.A., Szyperski, C.A., Wallnau, K. (eds.) CBSE 2006. LNCS, vol. 4063, pp. 1–16. Springer, Heidelberg (2006)
10. Lau, K.-K., Ukis, V.: Deployment Contracts for Software Components. Preprint 36, School of Computer Science, The University of Manchester, Manchester, M13 9PL, UK, ISSN 1361 - 6161 (February 2006)
11. Lau, K.-K., Ukis, V.: On Characteristics and Differences of Component Execution Environments. Preprint 41, School of Computer Science, The University of Manchester, Manchester, M13 9PL, UK, ISSN 1361 - 6161 (February 2007)
12. Lau, K.-K., Wang, Z.: A taxonomy of software component models. In: Proceedings of the 31st Euromicro Conference, pp. 88–95. IEEE Computer Society Press, Los Alamitos (2005)
13. Lee, D., Baer, J.-L., Bershad, B., Anderson, T.: Reducing startup latency in web and desktop applications. In: 3rd USENIX Windows NT Symposium, Seattle, Washington, July 1999, pp. 165–176 (1999)
14. Sun Microsystems. Java 2 Platform, Enterprise Edition. `http://java.sun.com/j2ee`
15. Szyperski, C., Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley, Reading (2002)
16. Wigley, A., Sutton, M., MacLeod, R., Burbidge, R., Wheelwright, S.: Microsoft.NET Compact Framework (Core Reference). Microsoft Press (January 2003)