On Characteristics and Differences of Component Execution Environments

Kung-Kiu Lau and Vladyslav Ukis

Department of Computer Science
University of Manchester
Preprint Series
CSPP-41

On Characteristics and Differences of Component Execution Environments

Kung-Kiu Lau and Vladyslav Ukis

February 2007

Abstract

Binary components are typically deployed into an execution environment. The execution environment impacts to a great extent how the components will execute at runtime. Current component models, however, neglect this fact. It is assumed that a component can run in any execution environment. In order to defeat this assumption, in this report we investigate current widely proliferated execution environments for components. We show what they are, characteristics they possess and differences they exhibit. The differences lead us to the insights why and how the component execution environment influences component runtime execution.

Keywords: components, deployment environments for components, web environment, desktop environment

Contents

1 Introduction 4

2 Component Execution Environments 4
2.1 Properties of Interest in an Execution Environment 5

3 Desktop Execution Environment 6
3.1 Transient State Management 6
3.2 Concurrency Management 7

4 Web Execution Environment 8
4.1 Transient State Management 8
4.1.1 Transient State Management in Java Server Pages 10
4.1.2 Transient State Management in Active Server Pages 12
4.1.3 Observations on State Retention Issues in Web Environment 13
4.2 Concurrency Management 14
4.2.1 Assembly Instantiation Modes 14
4.2.2 Concurrency Management with "An Assembly Instance Per Request" Mode 16
4.2.3 Concurrency Management with "An Assembly Instance Per User Session" Mode 17
4.2.4 Concurrency Management with "An Assembly Instance For All Requests" Mode 18
4.2.5 Concurrency Management with "A Pool Of Synchronised Assembly Instances For All Requests" Mode 19
4.2.6 Observations on Concurrency Management in Web Environment 20

5 Comparison of Desktop and Web Execution Environment 22

6 Resource Availability in an Execution Environment 23

7 Conclusion 25

List of Figures

1 Components with and without container deployed in an execution environment. . 5
2 Properties of interest in an execution environment. 5
3 Assembly in desktop environment. 6
4 Transient state management in the desktop environment. 7
5 Concurrency management in the desktop environment. 7
6 Assembly in web execution environment. 8
7 User-Assembly interaction using HTTP protocol. 8
8 Request-Response interaction mode using HTTP protocol. 9
9 Request-Response interaction mode and assembly instantiation. 9
10 Servlet Container. 11
11 Servlet Container with Single Thread Model. 12
12 ASP.NET Environment. 13
13 Concurrency management with “an assembly instance per request” mode. 16
14 Concurrency Management with “An Assembly Instance Per User Session” Mode. 17
15 Concurrency Management with “An Assembly Instance For All Requests” Mode. 19
16 Concurrency Management with “A Pool Of Synchronised Assembly Instances For All Requests” Mode. 20
17 Resources in an execution environment. 23
18 Components requiring resources in an execution environment. 24
19 Resources in an execution environment. 24

List of Tables

1 Transient state management in the web environment. 14
2 Assembly instantiation modes in the web environment with corresponding technologies for building web applications. 15
3 Assembly transient state and concurrency management depending on assembly instantiation mode in the web environment. 21
4 Comparing the properties of the desktop and web execution environment. 22

1 Introduction

A component goes through different phases during its lifecycle [13]. These phases are: design, deployment and runtime, in this order. In design phase, a component is designed and implemented in source code. Following this, in deployment phase, the component in binary form is deployed into an execution environment. Finally, at runtime, component instance is created and executed in the execution environment.

The execution environment the component is deployed into has great impact on how the component executes at runtime. This, however, is neglected by current component models. It is assumed that the component can run in any execution environment. In this report, we set out to show that this assumption is unpractical. We show that the execution environment influences how the component executes at runtime.

We construct the argument as follows. In section 2, we define current widespread component execution environments. They are desktop and web (server) environment. Following this, in section 3, we consider properties of the desktop environment, and in section 4 properties of the web environment. With the knowledge of the properties of these execution environments, we undertake a comparison of them in section 5. The comparison gives us clues as to how the execution environment influences components at runtime. Finally, we conclude the argument in section 7.

2 Component Execution Environments

In this section, our goal is to find out about execution environments for components and to explore their properties. In order to approach the topic, we draw our attention to current comprehensive platforms for software development: J2EE [22, 24, 17] and .NET platform [5, 30, 15, 18, 21]. These platforms allow for building of two different kinds of software systems: desktop and web applications. Desktop applications run on desktops, whereas web applications run on web servers. Following this, there are two environments for software systems: desktop and web (server) environment. These environments are recognised to be widespread in the current software industry [14].

The desktop and web execution environments fundamentally differ from containers for components [12], like e.g. EJB or CCM containers in EJB and CCM component models respectively. A component may run in a container. If so, then the container runs in the desktop or web execution environment. If not, then the component itself runs in either of the environments. In any case, the execution environment exists regardless of the existence of component container. This is illustrated in Figure 1.

On the left, components A and B are deployed into a container using so-called deployment descriptors [8]. The container, in turn, is deployed into an execution environment. On the right, components A and B are deployed in the execution environment without container. That is, it is the execution environment, desktop or web, and not the container that serves the lowest layer of component execution.

In the following sections, we explore properties of the desktop and web execution environments based on an analysis of J2EE and .NET platforms. We begin the endeavour in section 2.1 by defining properties of an execution environment that are of interest to achieve our goals.

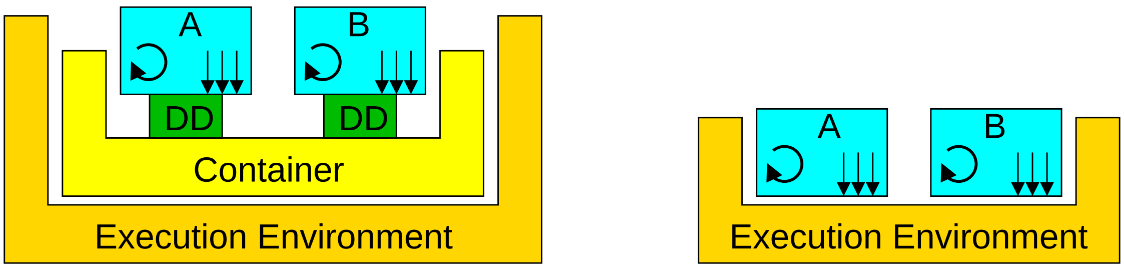

Figure 1: Components with and without container deployed in an execution environment.

2.1 Properties of Interest in an Execution Environment

Which properties of an execution environment are we interested in? Let us consider a binary component deployed into an execution environment. Such a component is shown in Figure 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 [12]. Furthermore, it may have some specific threading model implemented inside [12].

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

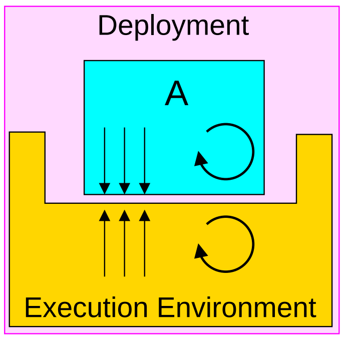

Figure 2: Properties of interest in an execution environment.

In Figure 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 [25, 10, 16, 26] of the component. The transient state of the component, unlike persistent state, can exist 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 two sections, we explore the first two of these properties in desktop (section 3) and web (section 4) execution environments. The availability of resources is a common topic for execution environments, which we deal with in section 6.

3 Desktop Execution Environment

The desktop environment is an execution environment for systems deployed on a desktop. We consider a system to be an assembly of components and refer to the "system" as "assembly" in this report.

A typical user interaction with assemblies deployed into the desktop environment is shown in Figure 3.

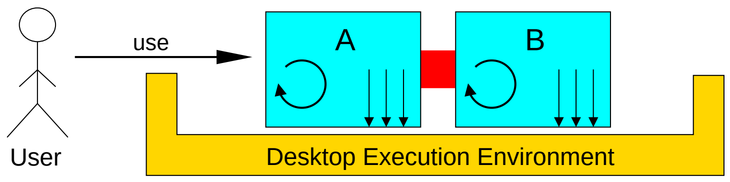

Figure 3: Assembly in desktop environment.

A component assembly AB is deployed into the desktop environment. There is a *single user* interacting with the assembly. It is a distinguishing characteristic of the desktop environment that it enables a single user to interact with an assembly instance.

Known examples of desktop applications include: MATLAB [9], Adobe Acrobat Reader [20], Ghost Viewer [4] etc. They provide a Graphical User Interface (GUI) [19] to enable the user to interact with them in a convenient way. Moreover, small programs like UNIX commands, e.g. ls or ps, also represent examples of systems deployed into the desktop environment. Unlike previous example systems, they do not provide a GUI interface to interact with them but are launched using a command shell.

3.1 Transient State Management

If a component assembly is deployed into the desktop environment, it is instantiated by the environment 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 assembly, a component accumulates state [1], the desktop environment does not interfere. This is shown in Figure 4.

The component A is deployed into 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.

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

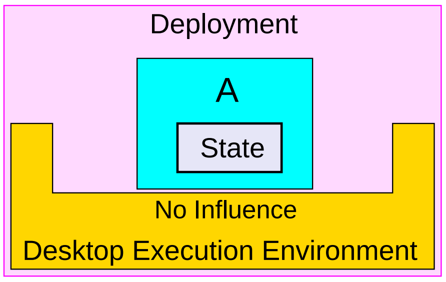

Figure 4: Transient state management in the desktop environment.

3.2 Concurrency Management

In general, it is possible to deploy both single-threaded and multithreaded component assemblies into the desktop environment. If the assembly is single-threaded, it uses the main thread, provided by the desktop environment, to process requests. If the assembly is multithreaded, 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 an assembly instance during its lifetime. This is illustrated in Figure 5.

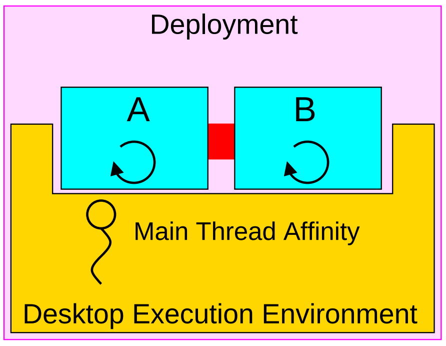

Figure 5: Concurrency management in the desktop environment.

The assembly 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 assembly may be single- or multithreaded. In any case, the assembly makes use of the main thread provided by the desktop environment. It is ensured by the desktop environment for the assembly AB that the main thread remains the same for the lifetime of an assembly 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 assembly.

4 Web Execution Environment

The web execution environment is an environment for component assemblies deployed on a web server. A typical user interaction with such assemblies is shown in Figure 6

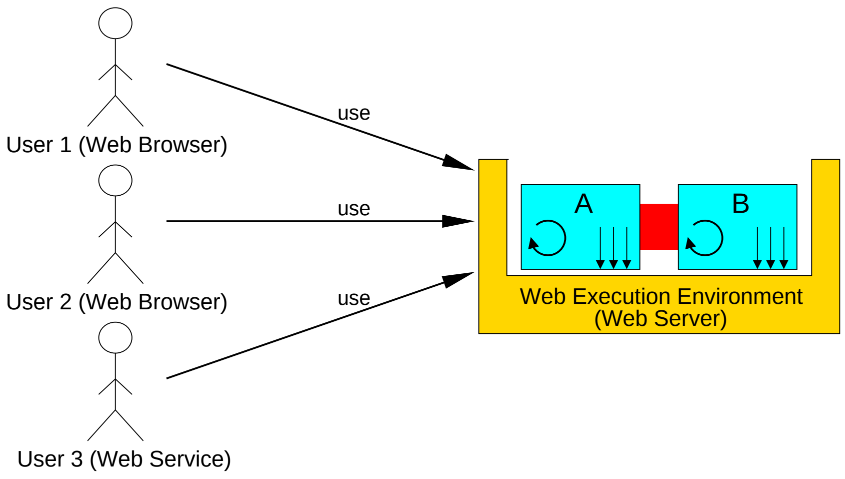

Figure 6: Assembly in web execution environment.

An assembly AB is deployed into the web environment. There are *many users*, possibly simultaneously, interacting with the assembly. It is a distinguishing characteristic of the web environment that it enables many users to interact with an assembly instance. A user typically uses a web browser to interact with an assembly deployed into the web environment.

Known examples of web applications include: web portals like Amazon [23], Ebay [11] or search engines like Google [2].

4.1 Transient State Management

The distinguishing characteristic of the web environment is that the user interacts with the assembly on the web server using Hyper Text Transfer Protocol (HTTP) [7], as shown in Figure 7.

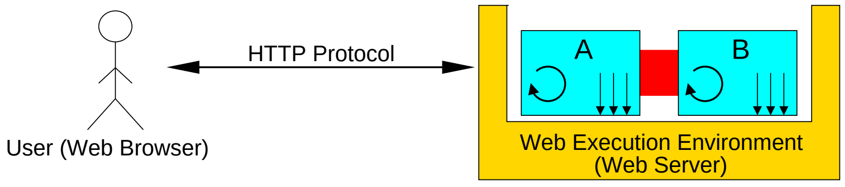

Figure 7: User-Assembly interaction using HTTP protocol.

The HTTP Protocol implements an interaction mode referred to as Request-Response mode [14]. This is shown in Figure 8.

The user places an HTTP Request 1 to the assembly and receives a result. Subsequently, the user places another HTTP Request 2 to the assembly and receives a result to it. An important characteristic of the HTTP protocol is that it does not maintain any transient state between

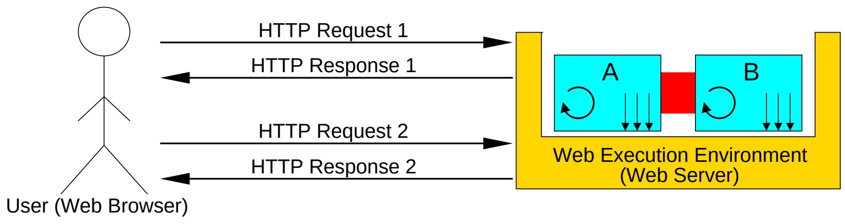

Figure 8: Request-Response interaction mode using HTTP protocol.

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 assembly on the web server. For instance, in the example from Figure 8, at the beginning of the interaction, the HTTP Request 1 is sent to the web server, processed by the assembly 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 assembly, HTTP Request 2. This request does not maintain any relationship to the Request 1 and is processed by the assembly without any transient state related to the Request 1.

However, why did the web server “forget” about the 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 assembly instance. The newly created assembly instance processes the request and generates a result. The web server *destroys* the assembly 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 Figure 9 for the two HTTP requests we considered before.

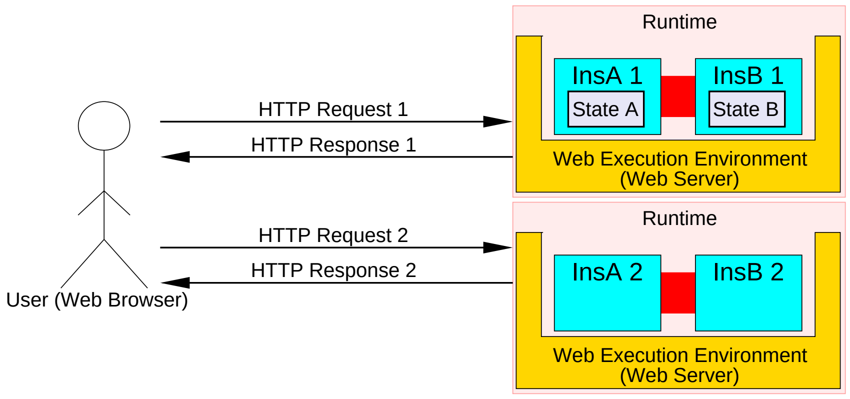

Figure 9: Request-Response interaction mode and assembly instantiation.

At the time when the user places the Request 1 to the assembly AB, an assembly instance actually does not exist. It is only created by the web server when the request arrives there. The newly created assembly instance processes the request and generates a result. Subsequently, the web server destroys the assembly instance and sends the result back to the user. The user is now not only disconnected from the assembly instance, which processed the Request 1, but the assembly instance actually does not physically exist any more. The web server does not maintain any information about the Request 1 either. Now that, the user places another request, Request 2, to the assembly AB. Again, no assembly instance is existent till the request arrives at the web server, which creates a new assembly instance. The assembly instance processes the Request 2 and generates a result. After that the web server destroys the instance and returns the result. Again, no assembly instance exists till another request to the assembly AB hits the web server.

In such environment, any component transient state cannot be retained between requests to the assembly. For instance, assuming components in Figure 9 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 an assembly 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 among requests to component assembly.

It is certainly possible to avoid transient state and to put all the component state information in a persistent storage, like e.g. a database, thus replacing transient by persistent state. However, this approach entails performance problems since all the state information has to be retrieved from the persistent storage at the beginning of each request processing. Furthermore, it has to be written back to the persistent storage at the end of each request processing, which is all time consuming.

In order to deal with the statelessness of the web environment, different technologies for web application development have been put forward. For instance, 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) scripts [27] follow the Request-Response model of the HTTP protocol and do not retain state on the web server side. In the following, we consider state management in JSP and ASP.NET since these technologies are representative and widely used in practice.

4.1.1 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. We investigate the technology here to gain an understanding of how it deals with the issue of statelessness of the web environment.

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 assembly to process all requests from all users. This is illustrated in Figure 10.

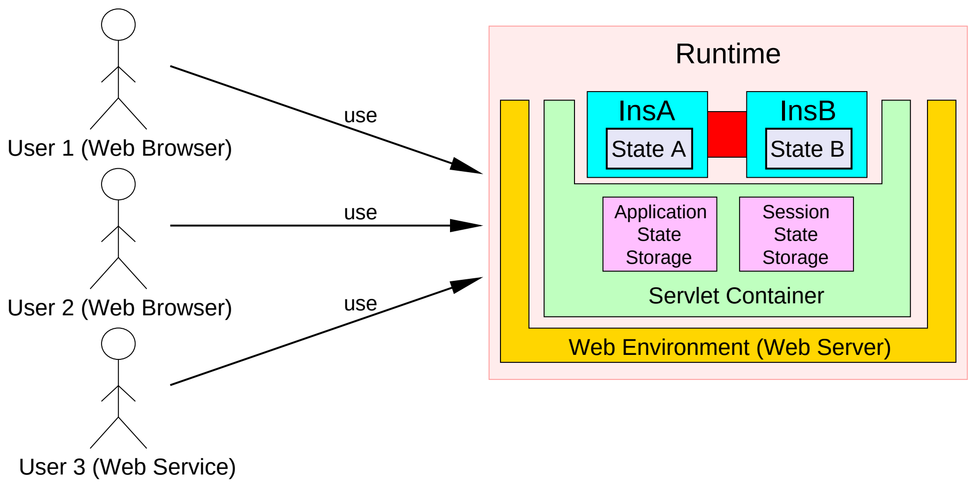

Figure 10: Servlet Container.

Users place, possibly simultaneous, requests to the assembly AB. An assembly 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 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 an assembly 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 assembly on the web server. The number of requests embraced by a browser session may vary and is tunable. Both state storages represent collections that can be used by component developers to store key/value pairs managed by the servlet container.

If a component assembly in the web environment is going to have a large number of concurrent users, it is inefficient to have a single assembly instance process all user requests. Therefore, the Java Servlet Technology provides another mode of assembly 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 assembly instances that process user requests. If the user requests are concurrent, the load is distributed among available assembly instances. This is shown in Figure 11.

Three users place simultaneously a request to the assembly AB. The requests are accepted by the web server and forwarded to the servlet container. The servlet container has created 3 instances of the assembly AB. It forwards the first request to an instance. The second one to another one, and lets the third request to be processed by the third assembly instance. If another request comes at the time of processing of the three requests, the servlet container will hold it until one of the assembly instances finishes a request processing. It will then assign the request to the free assembly instance.

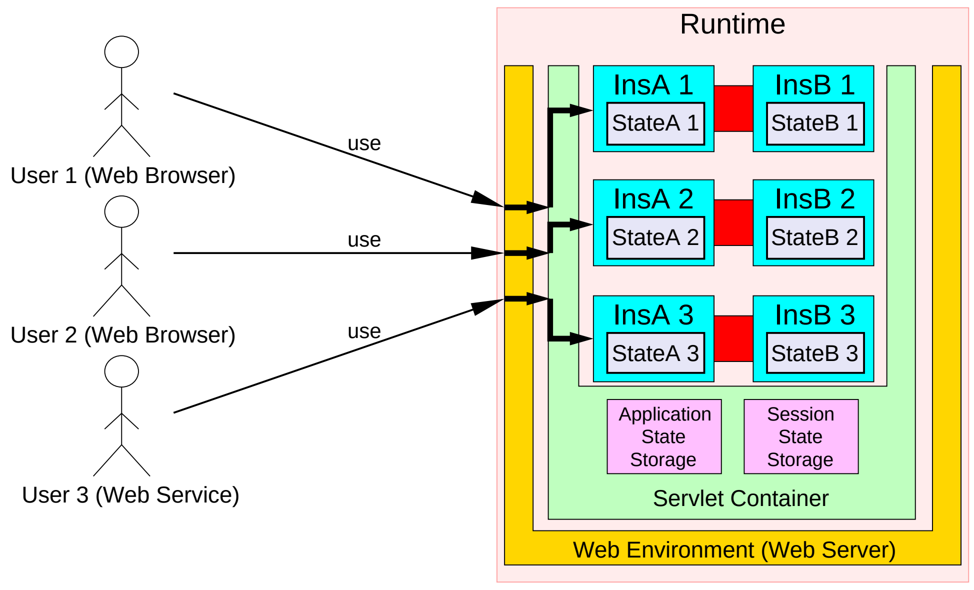

Figure 11: Servlet Container with Single Thread Model.

With the Single Thread Model, the container guarantees that an assembly 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 assembly instance. This makes state management in the assembly complicated. Indeed, a user request may be processed by an assembly instance. The assembly instance may accumulate some transient state during the request. Then, another request from the same user may be processed by another assembly 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 that let us consider how state is managed when using another technology – ASP.NET.

4.1.2 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 Figure 12.

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 container, with respect to assembly instantiation is that it follows the Request-Response model of HTTP protocol. That is, the ASP.NET environment creates and destroys a component assembly 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.

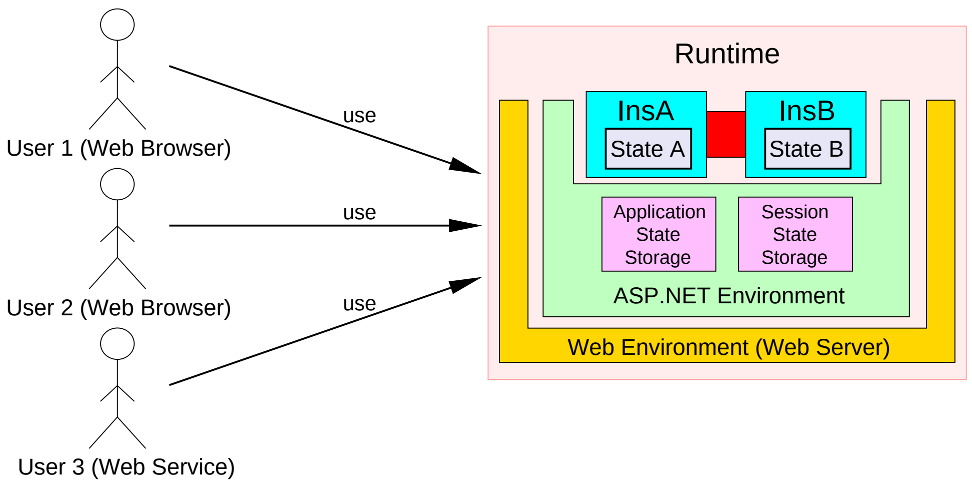

Figure 12: ASP.NET Environment.

4.1.3 Observations on State Retention Issues in Web Environment

The web environment is characterised by potentially very large number of concurrent users of a component assembly. This can be achieved by simply using many web browsers pointing simultaneously to the same URL. If the component assembly on a web server is stateful, it will potentially maintain state for each concurrent user of the assembly on the server side. A magnitude of thousands of concurrent users is a realistic number in today's web applications and there may be more at peak times. Such a heavy interactivity with the web server places a heavy burden on it. It may run out of resources (e.g. main memory) with increasing number of users and become overloaded. Additionally, a web server can run several web applications in parallel, which multiplies the burden placed on it. An increased load on the web server results in longer response times and deterioration of the availability of the web application. This, in turn, gives rise to dissatisfaction of the application users.

On the other hand, if the assembly on the web server is stateless, it does not store user-specific state information. As a corollary, the assembly is very scalable and can be exposed to a large number of users without placing an additional burden on the web server.

However, even with stateless assemblies, a web server can hardly cope with thousands of concurrent users. To improve the scalability of web applications, so-called web server farms [3] are used. A web server farm is made up of several web servers running the same web application. With a web server farm a request to an application is processed by the web server with the least load at the time of request. However, the next request may be processed by another web server than the one that processed the first request. In this context, if either application or session state storage is used on one of the web servers, the data may not be available on subsequent requests, if they are processed by other web servers of the server farm. That is, in the context of a web server farm the usage of transient state of any kind is not appropriate. Web applications designed for high scalability and deployment on a web server farm are designed stateless and maintain only persistent state, usually in databases. However, only very high scalable web applications need to be deployed to a web server farm. If a web application is not expected to have thousands of concurrent users, it can be hosted by a single web server and make use of state storages offered.

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

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

Table 1: Transient state management in the web environment.

With plain Request-Response mode implied by the HTTP protocol and used by CGI scripts, the transient state of the assembly is not retained among requests to the assembly. With ASP.NET there are several options of state management. By default, assembly state is not retained among requests. However, using application and session state storage, it is possible to retain state either for the lifetime of the assembly instance or for the duration of a user session respectively. Furthermore, with JSP, by default, assembly state is retained, as in the desktop environment. In addition, it can be stored in the application and session state storage. Moreover, with Single Thread Model, state is not retained among requests to the assembly but can be retained using state retention storages.

4.2 Concurrency Management

An assembly deployed on a web server is exposed to, theoretically, unlimited number of concurrent users. Therefore, concurrency issues are inherent in such a deployment. 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 an assembly instance during its lifetime[2]. 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 assembly instantiation mode. Below, we undertake a categorisation of assembly instantiation modes in the web environment.

4.2.1 Assembly Instantiation Modes

With CGI, Request-Response mode imposed by the HTTP protocol is used. That is, an assembly 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 *an assembly 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 *an assembly instance per user session* by using the session state storage. That is, an assembly instance is created on the first request

[2]In case of the web server farm, it is even not guaranteed that every request placed to an assembly instance is processed by the same computer. However, web server farm deployment is a rather special case we do not consider here.

from a specific user and is put into the session state storage. On all subsequent requests from the same user, the assembly is retrieved from the session state storage and used for request processing.

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

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

In summary, we can identify the following four assembly instantiation modes in the web environment:

- "An Assembly Instance Per Request",
- "An Assembly Instance Per User Session",
- "An Assembly Instance For All Requests" and
- "A Pool Of Synchronised Assembly Instances For All Requests"

They correspond to the individual technologies for building web applications in the way shown in Table 2.

	Assembly instance per request	Assembly instance per user session	Assembly instance for all requests	Pool of synchronised assembly 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

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

The assembly instantiation mode with which an assembly instance per request is created is available by default in CGI and ASP.NET. However, it is not available in JSP.

Furthermore, the assembly instantiation mode with which an assembly instance per user session is created is available in ASP.NET with the use of the session state storage. This mode is not available in CGI. Nor is it available in JSP.

Moreover, the assembly instantiation mode with which an assembly instance processes all requests is the default behaviour in JSP. This mode is also available in ASP.NET with the use of the application state storage. However, it is not available in CGI.

Finally, the assembly instantiation model with which a pool of synchronised assembly instances process all requests is available in JSP with the use of Single Thread Model. Such a mode is available neither in CGI nor ASP.NET.

Now that, let us explore the concurrency management with each of these assembly instantiation modes in the web environment.

4.2.2 Concurrency Management with "An Assembly Instance Per Request" Mode

In this mode an assembly instance is created and destroyed on each request processing cycle. In terms of threading, the web server allocates a thread for each request processing. To process a request, a new assembly instance is created and accessed by the thread allocated by the web server, which is the main thread for the instance. On the next request, the web server may allocate another thread for request processing. Again, an assembly instance is created and accessed by the allocated thread. In this situation, although the web server allocates two different threads for the processing of two requests to the assembly, there are no threading issues since each request is processed by a distinct assembly instance. In addition, as we learnt above, there is no state retention on the web server side between requests in this assembly instantiation mode. Therefore, there is no data which would have to be protected from concurrent access by multiple threads. Figure 13 shows such a scenario.

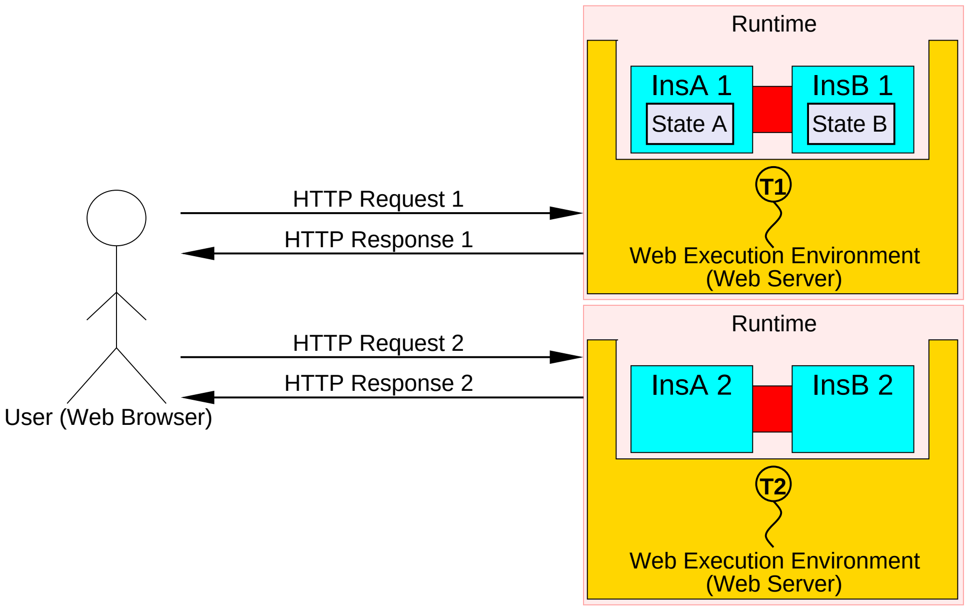

Figure 13: Concurrency management with "an assembly instance per request" mode.

The user places Request 1 to the assembly AB. To process the request, the web server allocates a thread T1. Furthermore, it creates an assembly instance. The instance processes the request on the thread T1 and generates a result, which is sent back to the user by the web server. Finally, the web server destroys the assembly instance. Subsequently, the user places another Request 2 to the assembly AB. To process the request, the web server allocates another thread T2. In addition, it creates a new assembly instance, which processes the request and generates a result. To complete the request processing, the web server sends the result is back to the user and destroys the assembly instance. In this situation, although the web server allocates two different threads, T1 and T2, to process different requests, Request 1 and Request 2, to the same assembly, there are no threading issues sine the requests are processed by two distinct assembly instances. Furthermore, transient state accumulated by the components A and B during the processing of Request 1 is not retained until the Request 2. This means that there is no data available on the processing of both requests, which would be accessed by two different threads and would need to be protected using thread synchronisation primitives.

With this assembly instantiation mode, concurrent requests from different users are processed by distinct assembly instances on different threads. However, all the assembly instances reside inside a single web server process. Therefore, singletons and static variables will be shared between these assembly instances and accessed concurrently by different threads. Therefore, singletons and statics have to be made thread-safe in such environment.

4.2.3 Concurrency Management with “An Assembly Instance Per User Session” Mode

With this assembly instantiation mode, an assembly instance is created for the duration of a user session. A user session comprises a set of requests from a single user. In terms of threading, for each request a thread is allocated by the web server. Therefore, an assembly instance in this mode can be accessed by multiple threads during a user session. However, since in a user session requests from a single user cannot be placed concurrently but only sequentially, multiple threads access the assembly instance sequentially. This is illustrated in Figure 14.

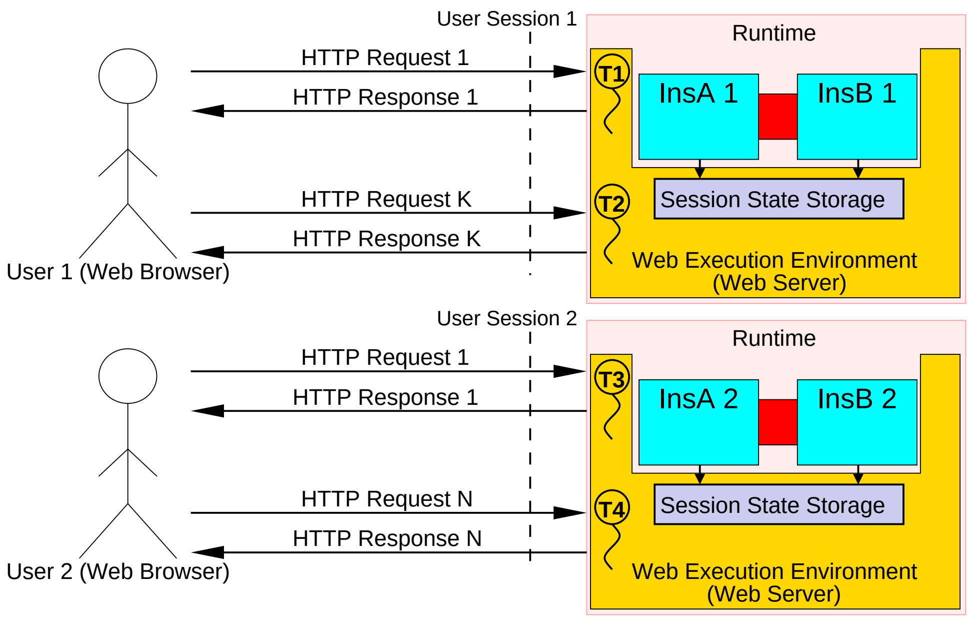

Figure 14: Concurrency Management with “An Assembly Instance Per User Session” Mode.

The User 1 places a Request 1 to the assembly AB. To process the request, the web server creates a User Session 1 and a thread T1. On this thread, an assembly instance is created, which processes the request and generates the result. The assembly is placed into the session state storage and the result is sent back to the user. Some time later, the same user places another request, Request K, to the assembly AB. The request falls into the User Session 1, i.e. the session has not yet expired. In order to process the request, the web server allocates another thread T2. On this thread, no new assembly instance is created but the assembly instance created at the beginning of processing of Request 1 is retrieved from the session state storage and used to process the Request 2. The retrieved assembly instance processes the request and generates a result, which is sent back to the User 1 by the web server.

Threads T1 and T2, the main threads for the assembly instance, accessed it sequentially, and not concurrently. That is, there is no thread affinity of the main thread in this assembly instantiation mode. In such situation, although no state corruption [29, 28] may occur since there is effectively no concurrent execution of the assembly instance, problems will arise with component elements requiring thread affinity.

Later on in Figure 14, another user places a request to the assembly AB. The web server creates another user session, User Session 2, and allocates a thread T3 to process the request. On the thread T3, an assembly instance is created, which processes the request and generates a result. Subsequently, the assembly is put into the session state storage and the result is sent back to the user. On the next request, the web server allocates another thread T4 for request processing. The request falls into the User Session 2. Therefore, to process it the previously created assembly instance is retrieved from the session state storage. The instance processes the request on the thread T4 and generates a result, which is sent back to the user.

Threads T3 and T4 served the main threads for the assembly instance. This means that the instance did not experience main thread affinity. If a component from the assembly AB has elements requiring thread affinity, the assembly would have failed.

Furthermore, in this situation, requests from different users are processed, concurrently, by different assembly instances. Since all assembly instances reside in the same operating system process, i.e. web server process, static variables or singletons in components are shared by different users. This may lead to several users operating on the same data specific to a particular user. Therefore, components with static variables or singletons may cause problems if deployed with the assembly instantiation mode "an assembly instance per user session".

4.2.4 Concurrency Management with "An Assembly Instance For All Requests" Mode

With this assembly instantiation mode, an assembly instance is created to process every request from every user. Since requests from different users may be placed concurrently and the web server allocates a thread for each request processing, the assembly instance is exposed to concurrent access by multiple threads. This is illustrated in Figure 15.

The User 1 places a Request 1 to the assembly AB. To process the request, the web server allocates a thread T1. On the thread T1, an assembly instance is created, which is used for processing of every request from every user later on. Now, the instance is processing the Request 1. However, in the middle of the processing, another Request 2 from the User 2 arrives. The web server allocates another thread T2 to process the request and lets the same assembly instance process it, concurrently to the processing of the Request 1. Now that, the assembly instance is executed concurrently by two threads, T1 and T2. However, in the meantime, another Request 3 from the User 3 arrives. The web server allocates yet another thread T3 to process the request and lets the same assembly instance process it. At this time, three threads T1, T2 and T3 execute concurrently in the same assembly instance.

In this situation, component transient state has to be protected by thread synchronisation primitives to avoid state corruption. That is, a situation where a thread reads some state while another one writes it. Furthermore, in such environment several threads may operate in a single assembly instance. This means that there is no thread affinity here and components requiring

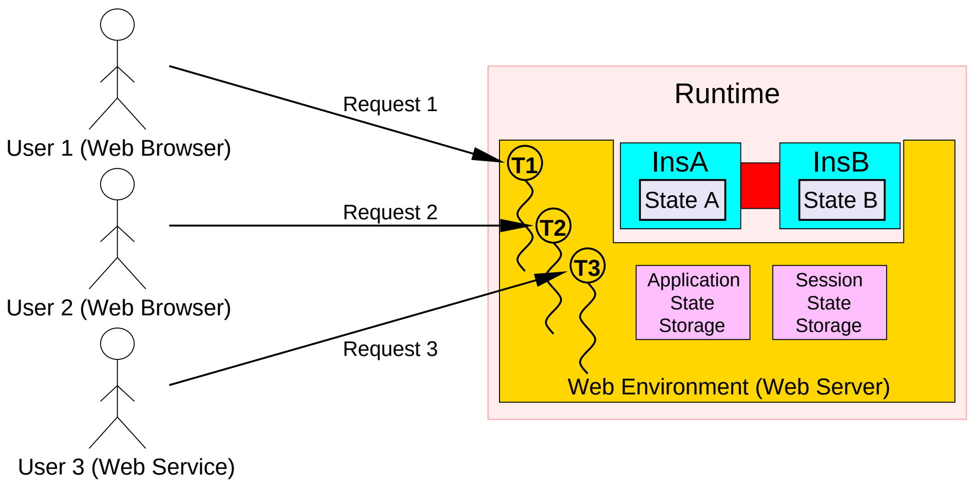

Figure 15: Concurrency Management with “An Assembly Instance For All Requests” Mode.

this will fail.

4.2.5 Concurrency Management with “A Pool Of Synchronised Assembly Instances For All Requests” Mode

With this assembly instantiation mode, a pool of assembly instances is created to process every request from every user. Additionally, an assembly instance from the pool is synchronised, i.e. an assembly never executes several requests concurrently but only one at a time. The web server allocates a thread for each request processing. As a corollary, an assembly instance can be accessed by multiple threads, although, since it is synchronised, only sequentially and not concurrently. This is shown in Figure 16.

On the right hand side of the Figure 16, the web server holds three assembly instances for request processing. On the left hand side, three users place their requests to the assembly AB concurrently. For each of the requests, the web server creates a thread, i.e. threads T1, T2 and T3, and lets each instance process a request. If during processing of the requests by the assembly instances another request arrived, the web server would queue it, wait till one assembly instance finishes its request processing and assign the processing of the newly arrived request to that instance. Moreover, for the processing of the new request the web server may allocate a new thread, which means that the assembly instance would be accessed, although sequentially, by more than one thread. That is, there is no thread affinity with this assembly instantiation mode. Therefore, components containing elements requiring thread affinity will fail here.

Furthermore, since all created assembly instances reside in a single web server process, singletons and statics will be shared among them. Since the assembly instances can be executed concurrently, the singletons and statics may be accessed concurrently by multiple threads. In order to sustain in such environment, the singletons and statics have to be made thread-safe.

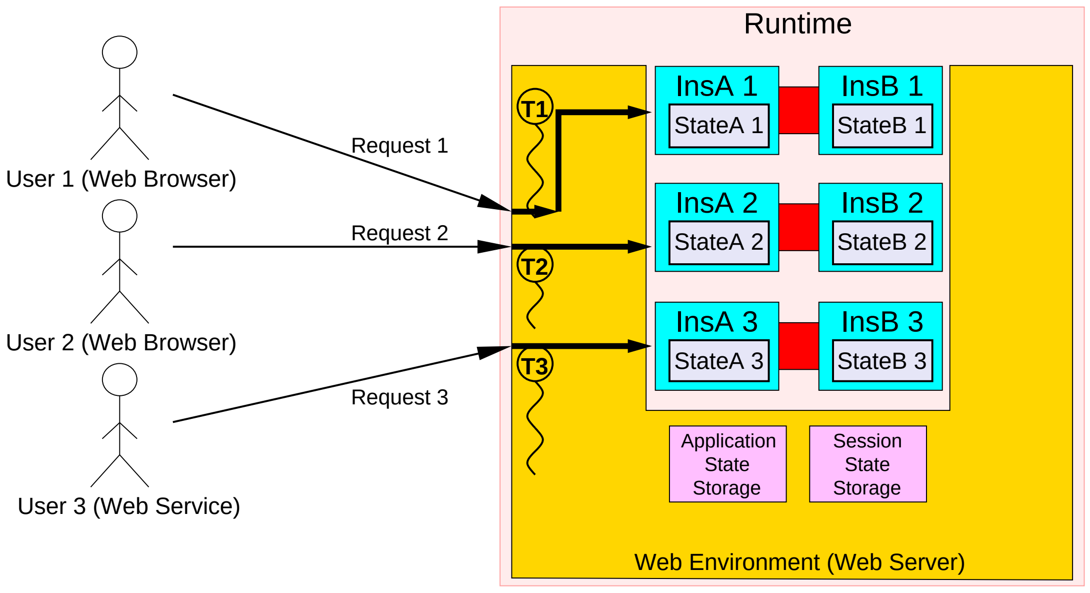

Figure 16: Concurrency Management with "A Pool Of Synchronised Assembly Instances For All Requests" Mode.

4.2.6 Observations on Concurrency Management in Web Environment

An inherent property of the web environment is that it enables a component assembly to be accessed by, potentially concurrent, multiple users. As a corollary, the web environment itself may impose threading issues on an assembly instance by exposing it to multiple threads. Moreover, the concurrency management in the web environment depends heavily on the assembly instantiation mode used. During the analysis of the concurrency management in the web environment, we witnessed several issues related to concurrency. In particular, 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:

1. A component in an assembly must contain static variables (statics) or singletons,
2. The assembly must be instantiated more than once in an operating system process,
3. Created assembly instances must be executed concurrently to each other by the execution environment. This means that an assembly instance itself may not be executed concurrently but process requests sequentially. However, other assembly instances are also executed at the same time,
4. 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 assembly instantiation mode used.

	Assembly instance per request	Assembly instance per user session	Assembly instance for all requests	Pool of synchronised assembly instances for all requests
Concurrency management	An assembly instance is accessed by one thread	An assembly instance can be accessed by multiple threads sequentially	Concurrent access of an assembly instance by multiple threads	An assembly instance can be accessed by multiple threads sequentially
Assembly transient state managm.	Assembly state is not retained among requests	Assembly state is retained during a user session	Assembly state is retained among all requests	Assembly state is not retained among requests
State Corruption Problem	No	No	Yes	No
Lack of Thread Affinity Problem	No	Yes	Yes	Yes
Shared Statics and Singletons Problem	Yes	Yes	No	Yes

Table 3: Assembly transient state and concurrency management depending on assembly instantiation mode in the web environment.

With the assembly instantiation mode “assembly instance per request”, an assembly instance is accessed by only one thread provided by the web environment. Therefore, the lack of thread affinity problem cannot occur here. Furthermore, the assembly state is not retained among requests to the assembly. 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 assembly instances concurrently.

Furthermore, with the assembly instantiation mode “assembly instance per user session”, an assembly instance can be accessed by multiple threads but only sequentially. Assembly state is retained during a user session. However, since no threads operate concurrently in an assembly instance, no state corruption problem will occur. On the other hand, due to the access of the assembly by multiple threads, lack of thread affinity problem may occur. Moreover, since in this mode an assembly 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 assembly instantiation mode “assembly instance for all requests”, an assembly can be accessed concurrently by multiple threads. Assembly state is retained among requests. Therefore, the state corruption problem may occur here. Furthermore, since the assembly 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 assembly with this assembly instantiation mode.

Finally, with the assembly instantiation mode "pool of synchronised assembly instances for all requests", an assembly instance can be accessed by multiple threads but only sequentially. Assembly 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 an assembly instance. Therefore, the lack of thread affinity problem may occur. Since there are several assembly instances all residing in the web server process, the shared statics and singletons problem may occur here as well.

Now that, in this section we have investigated state and concurrency management in the web environment. It is, indeed, rather different from the state and concurrency management in the desktop environment we learnt in section 3. To see the differences more clearly, in the next section we briefly compare the two.

5 Comparison of Desktop and Web Execution Environment

As we witnessed in sections 3 and 4, desktop and web environments differ substantially with respect to the management of state and concurrency in component assemblies. In this section, we want to summarise major differences of the environments. They vary in:

- the way assembly instantiation is performed,
- the way component state is handled,
- the way the main thread for an assembly instance is allocated and
- the way an assembly instance is exposed to the users

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

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

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

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

Furthermore, in the desktop environment, there are no state retention issues due to the fact that a single assembly 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 assembly instance processes all requests. Rather, different technologies deal with assembly instantiation differently making it complicated to maintain state among requests.

Moreover, in the desktop environment, the main thread for an assembly 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, an assembly instance is never exposed to multiple threads induced by the environment. Unlike the desktop environment, the web environment makes it possible for an assembly instance to be concurrently accessed by multiple users, thus exposing the instance to the concurrent access of multiple threads.

The properties from Table 4 make component deployment to desktop and web environment quite different. It is due to these properties that a component may be able to run in the desktop but not in the web environment and vice versa.

In the following chapter we investigate resources that can be offered to the component by an execution environment.

6 Resource Availability in an Execution Environment

An execution environment may hold a set of resources that can be used by components from a component model. The type of the execution environment, i.e. desktop or web, is irrelevant in this context since the set of resources may be held by either of them. Figure 19 illustrates an execution environment with a resource set.

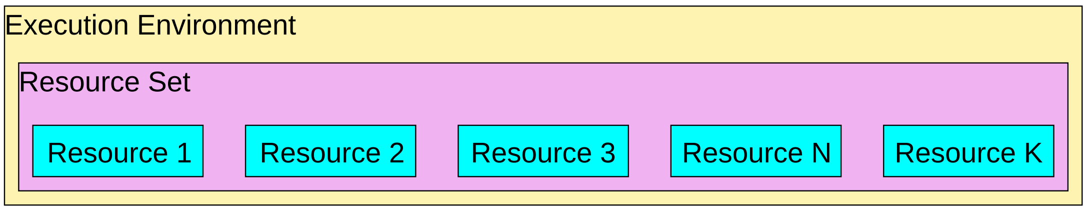

Figure 17: Resources in an execution environment.

The set is made up of five resources: Resource 1, 2, 3, K and N. In general, the number of resources in the resource set held by an execution environment is unrestricted. Our task in this section is to investigate all resources that can be found in a resource set of an execution environment. This knowledge will enable us to know if resource required by a component are available in the execution environment it is deployed into. Such a scenario is shown in Figure 18.

Components A and B are designed independently with the use of resources R1, R2 and R3. At deployment time, they are deployed in an execution environment with a resource set available. The question is now whether the resources R1, R2 and R3 are available in the execution environment. The answer to the question will determine whether the assembly AB is able to run in the execution environment.

To find all resources that can be available in an execution environment, we undertook an analysis of the APIs of J2EE and .NET frameworks. In particular, we searched for APIs that enable access to resources. Resources that can be used by a component fall into 3 categories:

- Operating-system resources, e.g. files etc. (1)

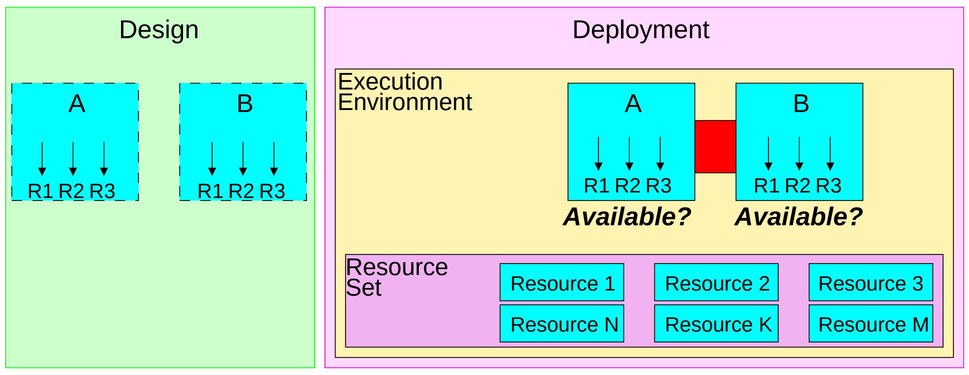

Figure 18: Components requiring resources in an execution environment.

- Resources offered by a framework, e.g. communication channels etc. (2)
- Other resources on a computer, e.g. databases etc. (3)

From these resource categories, the categories (1) and (3) constitute resources that can be found in a resource set of an execution environment. The category (2) represents higher-level abstractions defined by a framework, not found in an execution environment.

The resources we discovered are restricted to and complete with respect to the APIs of J2EE and .NET framework. However, the comprehensiveness and wide applicability of the investigated frameworks should imply the same for the derived results. The resources can be grouped as shown in Figure 19.

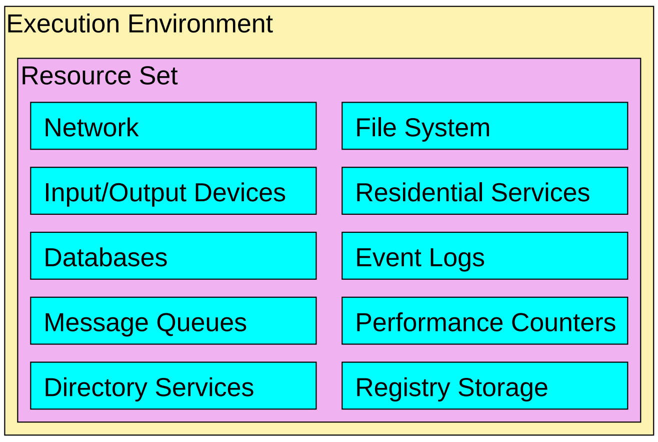

Figure 19: Resources in an execution environment.

Network – Network resources are available in an execution environment in order to enable components to access remote resources like e.g. web services.

File System – File system in an execution environment enables components to access files and file directories.

Input/Output Devices – Input/Output devices in an execution environment enable components to access external devices connected to the computer the component is deployed to. For instance, such devices include: Printers, universal serial bus (USB) [1] devices like memory pens, infrared devices etc.

Residential Services – Residential services in an execution environment enable components to access services that are constantly running in the background. For instance, such services include: Dynamic Host Configuration Protocol (DHCP) [6] client servers, global message queues, security services etc.

Databases – Databases in an execution environment enable components to efficiently access large amounts of data in a structured way.

Event Logs – Event logs in an execution environment enable components to access computer-wide event logs managed by the operating system. Such event logs are widely used as storage for important events during system execution, which can be consulted in case of system failure.

Message Queues – Message queues in an execution environment enable components to send to, receive and pick messages from computer-wide queues used for intra-process communication.

Performance Counters – Performance counters in an execution environment enable components to access component-wide counters managed by the operating system that allow systems to accumulate performance data about themselves.

Directory Services – Directory services in an execution environment enable components to access light-weight databases for storing hierarchical data according to a schema.

Registry Storage – Registry storage in an execution environment enables components to access persistent stores that can efficiently manage data in form of key/value pairs.

Each resource from the above 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.

7 Conclusion

In this report, we set out to investigate current component execution environments. Our goal was to show that the assumption made by current component models that a component can run in any execution environment is unpractical.

We first defined that currently there are two widespread execution environments for components: web and desktop. Following this, we investigated state and concurrency management of these environments. We showed that the two environments fundamentally differ in these regards. This provides motivation for the necessity of compatibility checks between components and execution environments they are deployed into.

Finally, we also investigated resources that can be available in an execution environment. The set of resources we found is valid for desktop and web execution environment.

References

[1] D. Anderson. *Universal Serial Bus system architecture.* Addison Wesley, 2001.

[2] L. A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The Google cluster architecture. *IEEE Xplore*, 23(2):22–28, 2003.

[3] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu. The state of the art in locally distributed Web-server systems. *ACM Comput. Surv.*, 34(2):263–311, 2002.

[4] B. Casselman. *Mathematical Illustrations: A Manual of Geometry and PostScript.* Cambridge University Press, 2005.

[5] Microsoft Corporation. MSDN – .NET Framework Class Library, Version 2.0, 2005.

[6] R. Droms. *Dynamic Host Configuration Protocol (DHCP)*, 1997. RFC 2131.

[7] R. Fielding, J. Gettys, J. Mogul, H. Nielsen, and T. Berners-Lee. *Hypertext transfer protocol HTTP/1.1*, 1997. RFC 2068.

[8] R. M. Haefel. *Enterprise Java Beans.* O'Reilly, 4th edition, 2004.

[9] D. Hanselman and B. C. Littlefield. *Mastering MATLAB 5: A Comprehensive Tutorial and Reference.* Prentice Hall, 1997.

[10] G. T. Heineman and W. T. Councill, editors. *Component-Based Software Engineering: Putting the Pieces Together.* Addison-Wesley, 2001.

[11] D. Houser and J. Wooders. Reputation in Auctions: Theory and Evidence from eBay. *Journal of Economics and Management Strategy*, 2004.

[12] K.-K. Lau and V. Ukis. Defining and Checking Deployment Contracts for Software Components. In *Proceedings of the 9th International Symposium on Component-Based Software Engineering*, volume 4063 of *LNCS*, pages 1–16, Stockholm, Sweden, June 2006. Springer.

[13] K.-K. Lau and Z. Wang. A survey of software component models. Preprint CSPP-30, School of Computer Science, The University of Manchester, April 2005.

[14] D. Lee, J.-L. Baer, B. Bershad, and T. Anderson. Reducing startup latency in web and desktop applications. In *3rd USENIX Windows NT Symposium*, pages 165–176, Seattle, Washington, USA, July 1999.

[15] C Skibo M Young, B Johnson. *Inside Microsoft Visual Studio .NET 2003.* Microsoft Press, 2nd edition, 2003.

[16] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs.* Wiley, 2nd edition, February 2006.

[17] V. Matena and B. Stearns. *Applying Enterprise JavaBeans – Component-based Development for the J2EE Platform.* Addison-Wesley, 2000.

[18] Microsoft .NET web page. `http://www.microsoft.com/net`.

[19] B. A. Myers. *Graphical User Interface Programming.* CRC Press, 2004.

[20] T. Padova. *Adobe Reader 7 Revealed: Working Effectively with Acrobat PDF Files.* Adobe Press, 2005.

[21] D. S. Platt. *Introducing Microsoft .NET.* Microsoft Press, 3rd edition, 2003.

[22] E. Roman. *Mastering Enterprise JavaBeans and the Java 2 Platform.* Wiley, enterprise edition, 1999.

[23] A. Roth and A. Ockenfels. Last Minute Bidding and the Rules for Ending Second-Price Auctions: Evidence from eBay and Amazon Auctions on the Internet. *American Economic Review*, 92:1093–1103, 2002.

[24] Sun Microsystems. *Java 2 Platform, Enterprise Edition.* `http://java.sun.com/j2ee`.

[25] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, second edition, 2002.

[26] R. N. Taylor, N. Medvidovic, K. M. Anderson, Jr. E. J. Whitehead, and J. E. Robbins. A component- and message-based architectural style for GUI software. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 295–304, New York, NY, USA, 1995. ACM Press.

[27] G. Venkitachalam and C. Tzicker. High performance common gateway interface invocation. In *IEEE Workshop on Internet Applications*, pages 4–11, San Jose, CA, USA, August 1999.

[28] J. Voas, G. McGraw, A. Ghosh, and K. Miller. Glueing together Software Components: How good is your glue? In *Proceedings of Pacific Nothwest Software Quality Conference*, 1996.

[29] J. M. Voas. Certifying off-the-shelf software components. *IEEE Computer*, 31(6):53–59, 1998.

[30] A. Wigley, M. Sutton, R. MacLeod, R. Burbidge, and S. Wheelwright. *Microsoft .NET Compact Framework(Core Reference).* Microsoft Press, January 2003.