# Apartment threading in a client-server enviroment

**Benjamin Muskalla (EclipseSource)**


Version: **1.0.2**

## Abstract

This work covers the aspects of an event loop in a client-server enviroment. The event loop itself is a well known concept of UI toolkits [Pik89] in order to react to the users input. In this paper we will first declare a context for which such an implemenation is needed. Furthermore there will be a discussion of several problems arising from this kind of pattern in order to determinate a working solution. The result of this paper is now part of the described product in the first sections.

# Contents

# 1 Introduction

## 1.1 Context

Eclipse evolved from a ground-breaking IDE to a new platform for application developers. The Eclipse foundation[1] presented with the 3.0 release of Eclipse a platform concept called Rich Client Platform (RCP)[2]. RCP was designed to have an open architecture in order to build new (client) application without the need to have any IDE-specific parts in it. In opposite to other frameworks RCP is very flexible in the ways a developer can compose new applications. By the use of Equinox - a reference implementation of the OSGi specification [All03] - you have a powerful and highly dynamic plugin-oriented application model. In addition Equinox helps developers with additional services on top of OSGi like the extension point mechanism (see Figure 1[3]) The base technology for all UI-related activites is the Standard Widget Toolkit (SWT)[4]. SWT acts as a widget toolkit by providing an abstraction layer above the operation system resouces and reuses the native widgets in opposite to other UI frameworks like Swing. Besides Eclipse itself there are many prominent projects using RCP as their application framework like Bioclipse or IBM Lotus Symphony.

## 1.2 Single sourcing and beyond

*Write once, run everywhere* is the main objective of the Java™programming language [AHL+05]. This holds true most of the time for running the same program on different operation systems. But as technology evolves the Eclipse Foundations wants to bring this slogan to a new level. Two new technology projects found their new home under the Eclipse.org umbrella. Those two - namely RAP and eRCP - provide an alternative runtime environment for RCP applications. This means that the application - normally

[1] http://www.eclipse.org
[2] http://www.eclipse.org/rcp
[3] Diagram is taken from http://en.wikipedia.org/wiki/OSGi
[4] http://www.eclipse.org/swt

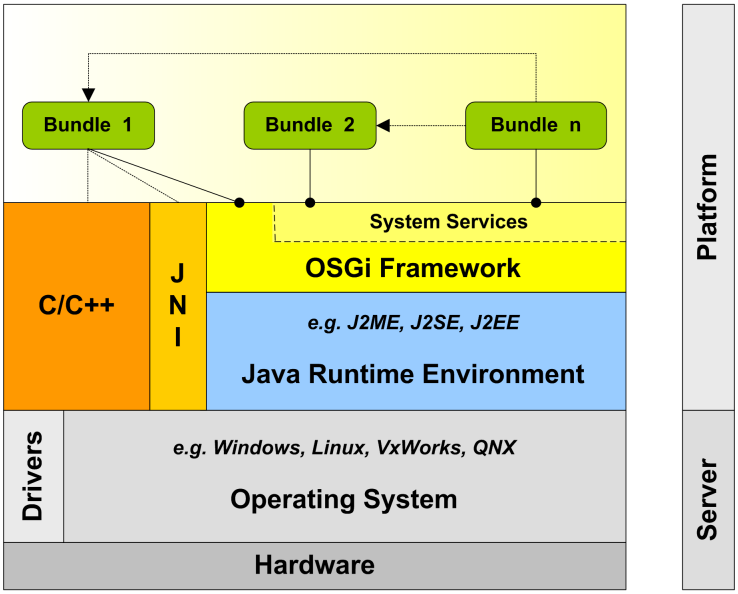


Figure 1: OSGi system layers

running as a desktop application on a personal computer - can now be deployed to another runtime environment. To have a concrete example in this case let us assume to have a ready to launch application based on RCP. By switching the runtime from RCP to RAP it is now possible to launch the application on an application server so clients can use the same application with a Web 2.0 centric interface within their browsers. There is no need to install any further add-ons or plugins for the user. In order to achieve a full compatibility between the platforms many concepts which are already implemented in SWT need to be adapted to other runtimes and hidden behind the public API which is synchronous across all runtime projects. One of the most interesting items is the event loop of SWT being ported to the multi-user environment RAP provides.

## 1.3 Problem description

The overall goal was to provide the same event loop mechanism as SWT does. In SWT they implemented the so-called *apartment threading* model in order to provide implementations for all currently known platforms. In order to reuse the same source code using this approach on a runtime like RAP there was a need to emulate the same principals while keeping the multi-user enviroment of RAP in mind. Another big difference is that RAP lives in an application server thus interacts trough the stateless HTTP [BLFF96] protocol with its clients. Due to the request-response cycle it is not trivial to implement the event loop as is as the request to the server would never come back to the client. This results in a more complex scenario to handle incoming events and dispatch them to the correct server-side widget implementation.

# 2 Event loop

## 2.1 Singled-threaded UI

Nearly all GUI toolkits currently on the market are implemented in a singled-threaded manner. This is based on the experience with frameworks (like AWT) trying to support a multi-threaded UI toolkit. As it is really hard to correctly lock and synchronize everything most frameworks achieve thread safety via thread confinement (see [Hyd99]); all GUI objects are accessed exclusively from the event thread [GPB+06]. This does not mean that the applications using SWT are restricted to use a single thread but every interaction with GUI toolkit needs to be done on a distinct thread. This thread is normally called the *event dispatch thread (EDT))* or *UI Thread*. Every interaction between and application and the ouside world is handled by the UI thread. All events fired due to user interaction with the operation system are put into a queue. Applications using SWT now have the chance to let SWT get those events out of the operation system queue and dispatching them to the corresponding SWT widgets. This is called the *readAndDispatch-loop*. The event loop repeatedly reads and dispatches the next user interface event from the operating system. When an event is dispatched, it is delivered to a widget for processing. For example, when the user moves the mouse, a mouse event is delivered to the control that is under the cursor. The *UI Thread* in our case is always the one which initially created the SWT Display [NW04].

## 2.2 SWT readAndDispatch mechanism

In order to have a more detailed look at the readAndDispatch mechanism we start first with a little example how applications have to use the SWT API for event dispatching (see Listing 1).

Listing 1: Hello World SWT Snippet

```
1 public static void main( String [] args ) {
```

```
2     Display display = new Display();
3     Shell shell = new Shell( display );
4     shell.open ();
5     while( !shell.isDisposed () ) {
6       if ( !display.readAndDispatch () ) {
7         display.sleep ();
8       }
9     }
10    display.dispose ();
11  }
```

As we saw in Listing 1 is that as long as there is no event to dispatch to the corresponding widgets the method *sleep* is called. It will wake up on any event recieved from the operation system and delegates it to the readAndDispatch mechanism. This allows a sequential event processing without busily waiting for the next event.

## 2.3 Distributed readAndDispatch mechanism

Adapting the apartment threading model to RWT has several fundamental problems. The first and most redoubtable issue is the distributed nature of RAP. As RWT relies on the Servlet specification [CY03] it has to handle the request-response cycle. Using the same approach as SWT would fail as spinning the event loop in the request thread would block all following requests. Another aspect the implementation should cover is the multi-user capabilites of RAP. In contradiction to RCP - which runs for one user at time - we need to care about multiple users accessing the server from multiple client machines.

# 3 Rich Ajax Platform Architecture

RAP is very similar to Eclipse RCP, but instead of being executed on a desktop computer RAP is run on a server and clients can access the application with standard browsers. This is mainly achieved by providing a special implementation of SWT (precisely a subset of SWT API). As we can see in *Figure 2* the architecture of RAP is mainly the same as in RCP. Only the fundamental layers needed some architectural restructuring due to the distributed nature of RAP. A rough overview about the structure of RWT in comparison to SWT is given in Section 3.1.

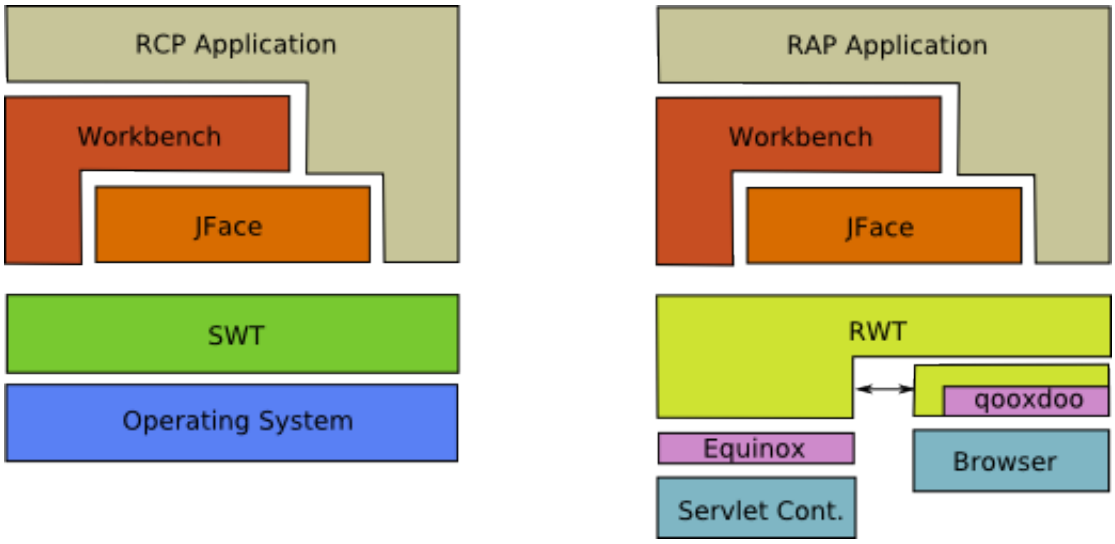


Figure 2: RAP Architecture

## 3.1 RWT vs. SWT

Without going into too much detail there is one difference between SWT and RWT which is worth to mention at this point. As we are in a multi-user environment with RAP it is necessary to have a common *entrypoint* to the application itself. While this is the well known *main* [AHL+05] method in pure Java programs we need a more abstract way to define such an entrypoint. RAP provides an interface called *IEntryPoint* which

contains the application startup code. The only three minor differences when looking at the code compared to the SWT example (see Listing 1):

- Class needs to implement *IEntrypoint*
- Code is in *createUI()*
- Needs to return a status of type *int*

Listing 2: Hello World RWT Snippet

```
1  public class HelloEntrypoint implements IEntrypoint
2    public int createUI() {
3      Display display = new Display();
4      Shell shell = new Shell( display );
5      shell.open ();
6      while( !shell.isDisposed () ) {
7        if ( !display.readAndDispatch () ) {
8          display.sleep ();
9        }
10     }
11     display.dispose ();
12     return 0;
13   }
14 }
```

## 3.2 Lifecycle

Due to the distributed nature of RAP there needs to be a mechanism to synchronize the state of the widgets between the server and the client. With each request from the client the life cycle on the server side will be executed. It is responsible to process all

changes and events from the client and in the end it will send back a response to the client how to update the UI (eg. hide/show widgets, update data, etc). The life cycle itself is splitted up in several phases which are executed in a specific order (see Figure 3).

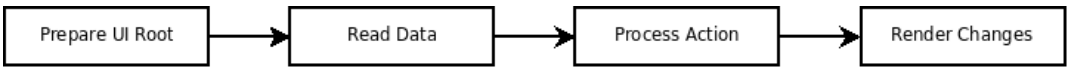


Figure 3: RAP Lifecycle

The RAP lifecycle itself is managed and executed by an instance of *ILifeCycleRunner* which in turn is triggered by the *LifeCycleServiceHandler*. The connection between the outside world and the life cycle exists between the Servlet API and the *LifeCycleServiceHandler*. In the next section the different phases are explained in a little more detail. We will only cover the aspects which are interesting later for explaining the implementation of the apartment threading model in RAP.

## 3.3 Phases

although there are four distinct phases implemented in RAP (see Table 1) we need to take care that not all phases are executed with every request. The regular behavior of RAP only uses the *ProcessUIRoot* and *Render* for the initial request of the application. This is done because with the initial request there is no need to execute the remaining phases as there no events triggered yet from the client. All subsequent requests use the full chain of phases to respond to the client. Each phase is represented as a full object by implementing the State design pattern [GHJV95, State].

Table 1: Lifecycle phases

| Phase | Description |
|---|---|
| Prepare UI Root | To initially start the application there needs to be a bootstrap rendered to the client. In the case of using the qooxdoo client side library this means to generate an html page which includes references to all dependent javascript libraries. This phase is only interesting for the first request. |
| ReadData | This is responsible to read the values sended from the client like occurred events. At the end of this phase, all widget attributes are in sync again with the values on the client. The attributes are preserved for later use. |
| ProcessAction | ProcessAction is responsible for dispatching the events to the widgets. Attributes may change in this phase as a response for the events. |
| Render | At the end of the lifecycle every change will be rendered to the client. Be aware that only values which are different than there preserved ones are send to the client (means: only the delta). |

# 4 Implementation

There are several problems we needed to address while implementing a real readAndDispatch mechanism in RAP. One of the hand side we need to cover all differences behind the real SWT API. Another big problem is that we cannot disturb the request-response cycle while spinning the event loop.

## 4.1 Seperating the event loop

In order to prevent blocking the request thread we need to move the facilities of the event loop to a separate thread. As you can see in Section 4.3 this is a dedicated thread which is responsible for spinning the event loop. This helps to have a non-blocking request thread in order to send the response back to the client at the end of the lifecycle (see Section 3.2). The implementation requires that the lifecycle now runs in the UI thread. This semantic change was needed to support the strict apartment threading of SWT as widgets can only be accessed by the UI thread. But this does not affect client code at all unless they relay on being handled in the request thread. As the lifecycle needs to touch the widgets during the **readData** and **processAction** phases (see Section 3.3) the lifecycle needs to run in the UI thread. A quick look at the thread switching (see Figure 4) makes clear how the mechanism behaves at runtime.

## 4.2 Initial lifecycle bootstrap

The initial call to the lifecycle by the *LifeCycleRunner* is handled by the *execute* method of the current life cycle implementation. We assume here that the current lifecycle always refers to the **RWTLifeCycle** implementation. One thing which will definitely happen when the lifecycle is called is that it tries to acquire an instance of a new *UIThread*. With the instance of the session-scoped *UIThread* we can now initiate the thread switching to run the lifecycle within the designated thread. The real thread

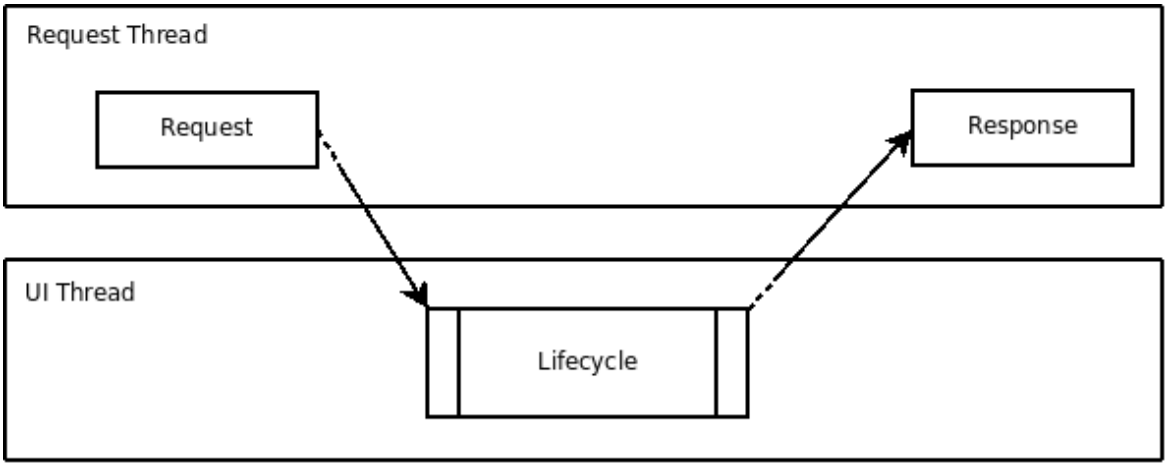


Figure 4: Transition between UI and request thread

switching happens in the *UIThread#switchThread* method. The implementation just lets the current active thread (the request thread) wait and notifies all other waiting threads. Depending on the calling context - whether it is the initial startup of the application or a subsequent request - the call to *notify* will start/resume the execution in the now active UI thread.

## 4.3 Dedicated UI Thread on startup

As mentioned before the real event loop spins inside the UI thread. As we need to combine the event loop and the existing life cycle phases we need to start processing the life cycle. One of the most important methods in this case is *RWTLifeCycle#continueLifeCycle*. The method name already divines that it will continue the life cycle from its current phase on. On the first request the application needs to be started. This is done by calling the application startup implementation residing in the applications entrypoint (see Listing 2). Running this ends up with a call to *Display#readAndDispatch* which internally just delegates to *RWTLifeCycle#readAndDispatch*. All queued events and runnables are now executed one by one in order to start the application. By surrounding the readAndDispatch call by a while-loop the application code itself spins as long as all queues are non-empty. If all queues

are empty the readAndDispatch call will return *false*. If everything went fine the control flow now reaches the *sleep* call which tells the UI that there are no events to process and the UI thread can fall into a non-blocking sleep. As we need to finish the lifecycle - remember that we are in the initial request - and we only consumed the **PrepareUIRoot** phase yet. To render the current server-side state of the application there is one phase missing: the Render phase. Internally the call to *sleep* has to handle this by calling *RWTLifeCycle#continueLifeCycle* again. The life cycle is now complete we need to remember that we're still in the UI thread without coming back. It is the time now to complete the current request that is waiting for us in *RWTLifeCycle#execute* (at that point it called *switchThread* and slept). The implementation of *sleep* now does the same thing as we did in the beginning. It calls *UIThread#switchThread* but this time from the UI thread. Remember that we initially switched from the request thread to the UI thread. This time the switch is in the other direction and we continue to run the *execute* of the life cycle (see Figure 5). As the UI thread is still interesting for all subsequent requests it is stored in the current session of the user.

## 4.4 Continuing the UI thread

The initial application is now rendered in the client. On the first event occurring on the client it triggers a round-trip to the server with a new request. The initial processing is happening as before (see Section 3.2). As the session already has a dedicated UI thread stored in the session we will use this one to do the thread switch. The thread switch is needed to run the lifecycle inside the UI thread as before. And this is one of the most interesting parts of the implementation. As we saw in the section before the last statement of the initial request inside the UI thread was the call to *switchThread* inside the sleep. The control flow then went down the road in the request thread. Now - in a subsequent request - when we switch to the UI thread again it is still inside the *sleep* method where it abdictated its responsibility. If we take a look at a (simplified) version of the sleep implementation (see Listing 3) we may imagine what is happening.

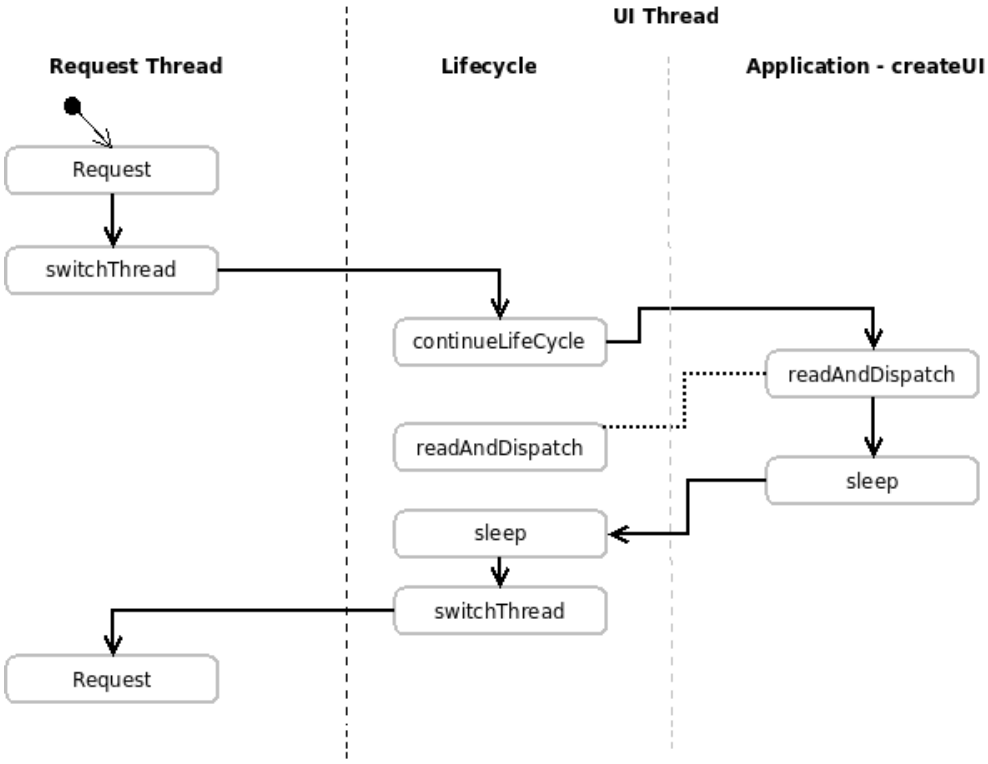


Figure 5: Thread transition at startup

The initial request got into the sleep method, finalized the lifecycle with the first call to *continueLifeCycle* and gave the control flow back to the request thread. Continuing the UI thread now in a subsequent request continues this thread at the exact same point where it again calls *continueLifeCycle* at the end of the method in order to start the lifecycle. This means that the UI thread is waiting inside the sleep method between any further request sent by the client.
As the life cycle is now started again for this subsequent request by the client it processes it first phases like ReadData and ProcessAction(see Section 3.3). After the ProcessAction phase which is responsible for delivering new events to the application the *continueLifeCycle* exits, with it the sleep method is removed from the call stack and the client application now is in has its turn. The return of sleep in the client code calls again *readAndDispatch* (as it is inside the loop) and thus all new events are dispatched to their corresponding widgets (see Figure 6).

Listing 3: sleep Implementation

```
1 public void sleep() {
2    continueLifeCycle();
3    uiThread.switchThread();
4    continueLifeCycle();
5 }
```

From this point the mechanism works as we already saw it in the intial request. When the event and action runnable queues are empty *readAndDispatch* will return *false*, the application calls *sleep* and the thread switch is triggered again to finalize the lifecycle. In the end the changes triggered by the event handlers are sended back to the client.

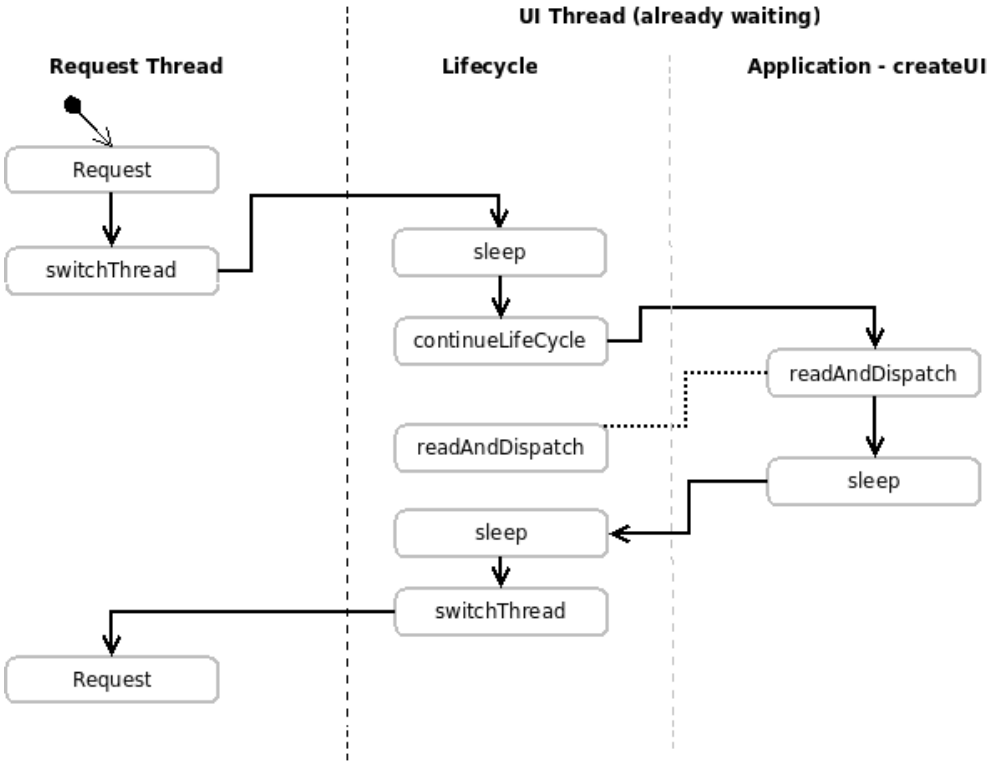


Figure 6: Thread transition at runtime

# 5 Conclusion

In fact we saw that it is possible to have an event-driven UI in a distributed enviroment. One of the key issues was the right handling and timing between the different threads. One of major drawbacks of this approach is that every active user session requires a dedicated thread to be stored within it. Unlike the request thread which is mostly acquired from an internal thread pool for each new request the UI thread needs to stay the same over the whole lifetime of a session. The final result of this work is included in the latest versions of RAP and seems to scale very well in common enviroments.

# Glossary

### (G)UI

(Graphical) User Interface

### API

Application Programming Interface

### eRCP

embedded Rich Client Platform

### HTTP

Hypertext Transfer Protocol

### IDE

Integrated Development Enviroment

### OSGi

Open Service Gateway initiative

### RAP

Rich Ajax Platform

### RCP

Rich Client Platform

### RWT

RAP Widget Toolkit

### SWT

Standard Widget Toolkit

# List of Figures



# List of Tables



# Listings

# References

[AHL+05] Ken Arnold, David Holmes, Tim Lindholm, Frank Yellin, Frank Yellin, The Java Team, Mary Campione, Kathy Walrath, Patrick Chan, Rosanna Lee, Jonni Kanerva, James Gosling, James Gosling, James Gosling, James Gosling, Bill Joy, Bill Joy, Guy Steele, Guy Steele, Gilad Bracha, and Gilad Bracha. Java language specification, third edition, 2005.

[All03] Osgi Alliance. *OSGi Service Platform: The OSGi Alliance*. IOS Press, December 2003.

[BLFF96] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (Informational), May 1996.

[CY03] Danny Coward and Yutaka Yoshida. Java$^{TM}$servlet 2.4 specification (jsr 154). http://jcp.org/aboutJava/communityprocess/final/jsr154/index.html, November 24, 2003.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[GPB+06] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley Professional, 5 2006.

[Hyd99] Paul Hyde. *Java Thread Programming*. Sams, 1 edition, August 1999.

[NW04] S. Northover and M. Wilson. *SWT: the standard widget toolkit. Volume 1*. Boston: Addison-Wesley, 2004.

[Pik89] Rob Pike. A concurrent window system. *Computing Systems*, 2:133–153, 1989.