UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science

Alar Kvell

Aranea Ajax

Bachelor thesis (4 AP)

Supervisor: Jevgeni Kabanov

Autor: .... "...." mai 2007
Juhendaja: .... "...." mai 2007

TARTU 2007

Contents

Introduction

This chapter introduces the work, its goal, its scope and target audience.

Motivation

In the last few years many web applications have started becoming more and more similar to desktop applications in terms of features and qualities. Using new technologies and techniques like Ajax has allowed web applications to create a much more responsive user experience, that was previously only specific to desktop applications.

The goals of this work are obtaining an overview of how Aranea web framework makes use of Ajax technologies and improving the framework by introducing several new Ajax-related features.

Prerequisites

This work requires the user to have a basic understanding of web development and programming in Java. Familiarity with Aranea is recommended.

Contributions and Outline

The first chapter introduces the concept of Ajax. Parts of it are based on a book [ZM06]. This is needed to understand the partial rendering and action features of Aranea described in later chapters.

The second chapter gives a quick introduction to the core principles of Aranea web framework (which is based on Aranea technical paper [MK06]) and then describes how actions can be used with Ajax. It also describes the enhancements that were made during the course of this work: adding the support of action calls to Aranea JavaScript API and introducing the support of unsynchronized actions to the framework.

The third chapter introduces Aranea's feature of updating web page regions via Ajax requests and the problems associated with its current implementation. It then describes the implementation of rendering encapsulation concept and partial rendering feature that were carried out during the course of this work.

The main contribution of this work is the implementation of partial rendering in Aranea 1.1. Smaller contributions include enhancements to Aranea's JavaScript API and synchronization filter service. The contributions are available as part of Aranea framework 1.1-M1 release, which is included on the accompanying CD and is also available on the website http://www.araneaframework.org/.

1 Ajax

This chapter introduces the concept of Ajax. Parts of it are based on the book „Professional Ajax“ [ZM06].

1.1 Introduction

Ajax stands for „Asynchronous JavaScript + XML“. It is an approach to building web applications that involves transmitting only a small amount of information to and from the server in order to give the user the most responsive experience possible.

Ajax applications can use many technologies, but only the following three are required:

- HTML/XHTML [EH02] – Primary content representation languages, needed for the display of information.
- DOM [DO] – Dynamic updating of the loaded page, changes portions of an XHTML page without reloading it.
- JavaScript [EL99] – Scripting language used to initiate client-server communication and manipulate the DOM to update the web page. (Any other scripting language that the web browser supports could be used as well.)

The client-server communication is asynchronous – it is performed in the background, while the web page remains usable to the user. The traditional HTTP [Fi99] requests that are performed by the web browser when the user clicks on a link or submits a form are executed synchronously, which means that user interaction with the web page is not possible during the request and after the completion of the request the web page content is replaced.

1.2 History

The way asynchronous client-server communication has been achieved, has changed over time. It first became possible with the introduction of frames to HTML 4.0 standard. Using a (hidden) frame with JavaScript created the possibility to perform asynchronous client-server communication.

The manipulation of document structure became possible with the introduction of Dynamic HTML (DHTML) [HD] in Internet Explorer 4.0 (later superseded by Document Object Model (DOM) standard). Combining DHTML with the hidden frame technique allows to refresh any part of the web page with server information at any time.

The most recent method of asynchronous client-server communication is using an ActiveX object called XMLHttp. Introduced in 2001 in Internet Explorer browser the object allows to initiate and control *ad hoc* HTTP requests from JavaScript. It provides access to HTTP status codes and headers as well as any data returned from the server.

Similar approach was soon adopted by Mozilla, Opera and Safari browsers. Instead of allowing access to ActiveX they replicated the object's principal methods and properties in a native browser object XMLHttpRequest, which Internet Explorer came to support starting from version 7.x [AN].

1.3 Different Techniques

Using the hidden frame technique allows to maintain the browser history and thus enable users to still use the Back and Forward buttons in the browser. As the browser doesn't know that a hidden frame is special in any way it keeps track of all requests made through it. Although the main page of an Ajax application doesn't change, the changes in the hidden frame mean that the Back and Forward buttons will move through the history of the frame instead of the main page. This technique is used in both Gmail [Gi] and Google Maps [GM].

The downside of hidden frames is that there is very little information available about the HTTP request that is happening behind the scenes. You are completely reliant on the proper page being returned by the hidden frame. There is no notification to the user that a problem has occured with the request. XMLHttp(Request) improves in that area.

Using XMLHttp for client-server communication instead of hidden frames has many advantages. The code that is written is much cleaner and the intent of the code is much more apparent than using numerous callback functions with hidden frames. The developer has access to request and response headers as well as HTTP status codes, which enables to determine if the request was successful.

Many Ajax applications use a mixture of XMLHttp and hidden frames to make a truly usable interface.

1.4 Ajax Request Work Flow

An Ajax request usually proceeds like this (see figure 1):

1. User interaction in the web page (or a preconfigured timer) creates an event in the browser.
2. JavaScript code is executed that processes the event. An Ajax request is initiated for client-server communication.
3. Browser sends the asynchronous HTTP request in the background.
4. Server processes the request and generates a response.
5. Browser receives the response in the background.
6. When the request is complete, a callback function (that was provided in step 2) is invoked by the browser . It processes the received response and performs necessary actions in the web

page.

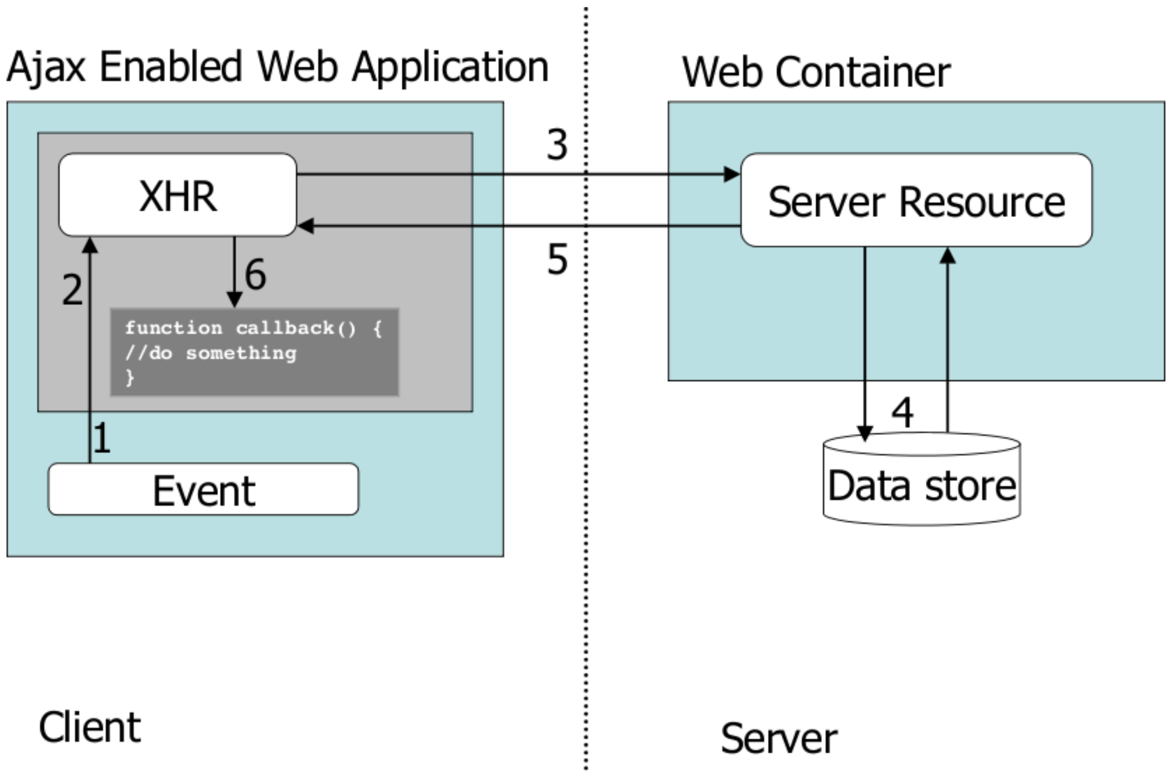

Figure 1: Ajax request work flow

Using Ajax requests changes the synchronous interaction pattern of a traditional web application to more asynchronous (see figure 2), as described by Garrett in 2005 [Ga05]. Every user action that normally would generate an HTTP request takes the form of a JavaScript call to the Ajax engine instead. The engine makes those requests asynchronously, without stalling user interaction with the application. It seems like adding an extra layer to the application would make it less responsive, but the opposite is true.

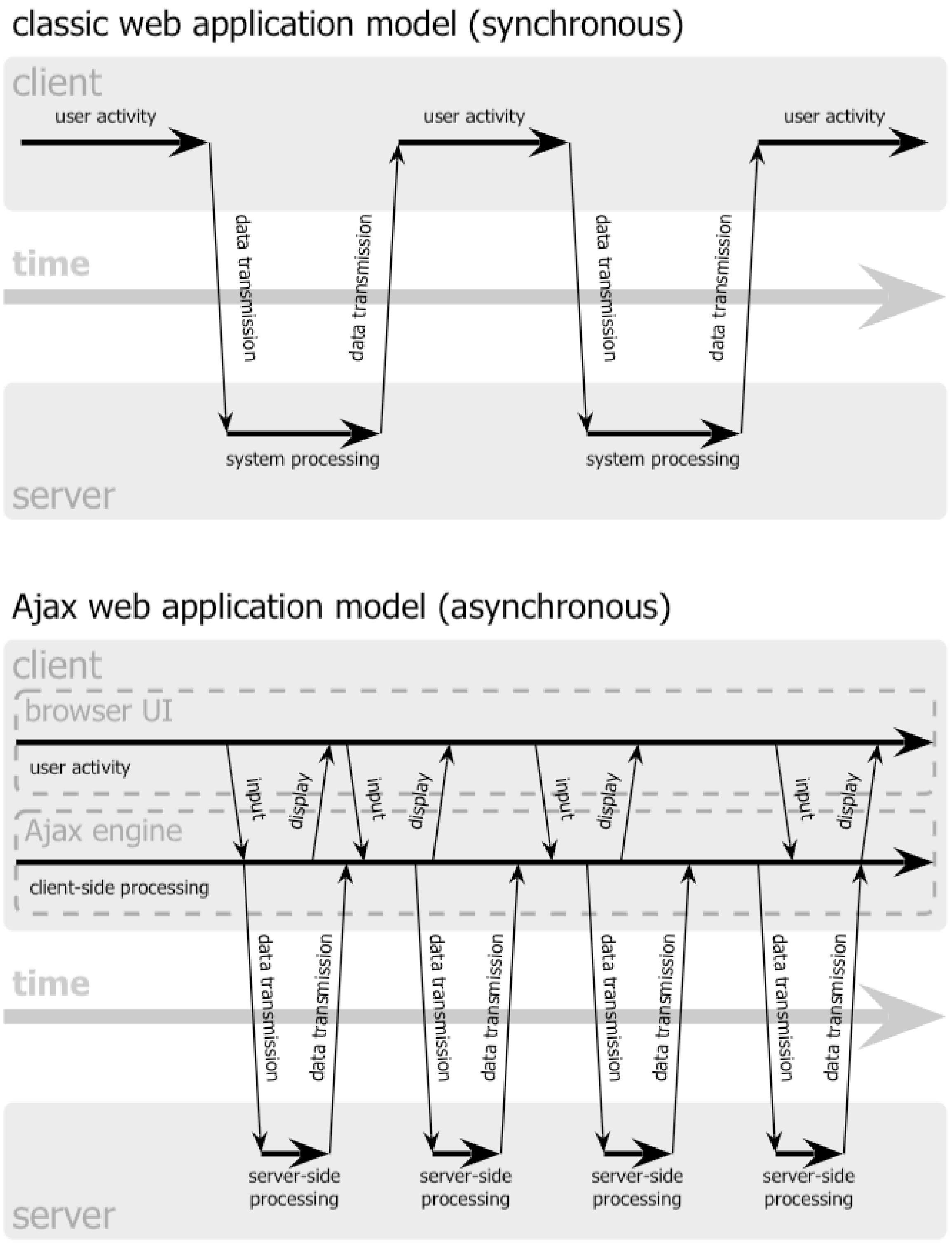

Figure 2. Ajax and classic web application models [Ga05]

1.5 Frameworks

Several frameworks have been created to make the task of developing Ajax-enabled web sites easier and quicker. They all try to abstract the details concerning the underlying communication between the server and the client, leaving the developer free to concentrate on the more interesting aspects, such as implementing the classes that actually take care of the application business logic.

Some frameworks enable the developer to create dynamic, AJAX-driven web user interfaces using only Java [Li06]. In GWT [GW] and Echo2 [EW], user interfaces are developed in a fashion similar to Swing [ST] or SWT [SS]: by assembling hierarchies of components and registering event handlers. Neither project requires the developer to work with HTML, JavaScript, or XML. The most obvious difference between GWT and Echo2 is that all of your GWT code is executed on the client, whereas your Echo2 code is executed on the server.

DWR [DW] allows code in a browser to use Java functions running on a web server just as if it was in the browser.

Some libraries provide various functions for developing JavaScript applications, ranging from programming shortcuts to major functions for dealing with XMLHttpRequest. Usually one of the goals is to provide abstractions of common browser features (event handling, DOM traversal and modification, XMLHttpRequest), while hiding cross-browser differences.

- Prototype [PJ]
- jQuery [JJ]

Some libraries provide effects such as animations and fades.

- script.aculo.us [SJ]
- jQuery

Some libraries provide user interface components, some of which make use of Ajax:

- Yahoo! User Interface Library

1.6 Conclusion

This chapter introduced the basic concepts of Ajax. Ajax is a new interaction model for web applications in which full page loads are no longer necessary. Although Ajax can be used to accomplish many things, it is best used to enhance the user experience rather than providing cool effects.

2 Actions

2.1 Introduction to Aranea

This chapter gives a quick introduction to core components of Aranea framework. It is a premise for the rest of the work, which is heavily connected to Aranea. It is based on Aranea technical paper [MK06].

Aranea is an object-oriented web controller framework. The core of Aranea is the idea that components are arranged into a dynamic hierarchy.

A component is a very simple entity that has a life-cycle that begins with an `init()` call and ends with a `destroy()` call. Component can be disabled and enabled again. Component has an `Environment` that is passed to it by its parent or creator during initialization. Environment can be used by children to discover services (named *contexts*) provided by their parents without actually knowing, which parent has provided it.

```
interface Component {
  void init(Environment env);
  void enable();
  void disable();
  void propagate(Message message);
  void destroy();
}
```

While the environment allows communicating with the component parents, messages allow communicating with the component descendants (indirect children).

```
interface Environment {
  Object getEntry(Object key);
}
```

```
interface Message {
  void send(Object id, Component component);
}
```

Service is an abstraction of a reentrant controller.

```
interface Service extends Component {
  void action(Path path, InputData input, OutputData output);
}
```

Action is a method for routing the request to the one service it is intended for. Path is an abstract representation of the full path to the service from the root.

The `InputData` and `OutputData` are generic abstractions of a request and a response, correspondingly. `InputData` can be used to process request data and `OutputData` can be used to generate a response.

Typically, services realize a *Filter* or a *Router* pattern. A filter is a service that has a single child and that will block or modify some requests. A router is a service that has several children, but routes any given request to only one of them.

Although services are very powerful, they are not too comfortable for programming stateful non-reentrant applications. Therefore, user interface components are usually `Widgets`:

```
interface Widget extends Service {
    void update(InputData data);
    void event(Path path, InputData input);
    void render(OutputData output);
}
```

Widgets extend services, but unlike them widgets are usually stateful and are always assumed to be non-reentrant. The widget methods form a request-response cycle that should proceed in the following order:

1. `update()` is called on all widgets in the hierarchy allowing them to read data intended for them from the request.
2. `event()` call is routed to a single widget in the hierarchy using the supplied Path. It allows widgets to react to specific user events.
3. `render()` calls are not guided by any conventions. If called, widget should render itself (though it may delegate the rendering to a template). The `render()` method should be idempotent, as it can be called arbitrary number of times.

Widgets also inherit an `action()` method. It may be used to interact with a single widget, e.g. for the purposes of making an asynchronous request through Ajax.

2.2 Actions

For all HTTP requests an action is generated at the top of the component hierarchy. The action travels down the hierarchy by each service forwarding it to its child or children (or modifying or blocking it). The framework is usually composed in such a way that services are at the top, below them are widgets. The two realms are connected by an adapter service that translates action calls to widget request-response cycle calls (`StandardWidgetAdapterService`). The translation process consists of the adapter calling the appropriate widget methods on its child. `Update()` and `render()` are always called, `event()` is called only when there exists a request parameter indicating the event path (`widgetEventPath`).

Although widgets extend services and therefore inherit an action method, their request-response cycle methods are usually used. The widget request-response cycle is a higher level abstraction than action calls and allows rendering and GUI logic code to be separated from each other. Event executes some GUI logic that may change the state of the widget. Rendering is not tied to any GUI logic and therefore may even be performed multiple times between events.

Executing the widget request-response cycle is the default case in `StandardWidgetAdapterService`. The alternative course is to continue propagating the action call to its child as an action. This is done if there exists a request parameter indicating the action path (`widgetActionPath`).

The routing process of action calls involves comparing component's id to part of the action destination path. The part of the hierarchy where components have identification starts with the user interface widgets. All the services and filter widgets above the named hierarchy have null as id. Above named hierarchy, the action call passes through every service. In the named hierarchy the action call is invoked only on the target widget.

Although all requests begin their life as actions, in the following discussion and in the following chapters actions usually refer to widget action calls, that are routed to a specific widget.

Widget action method provides a way to bypass the request-response cycle. It is a convenient way for a specific widget to return custom data, therefore especially suited to be the target of Ajax requests. Action call allows access to `InputData` and `OutputData`, providing direct control over the output response. Action calls impose less overhead, because the call is invoked only on a single widget, while the widget request-response involves multiple calls, many of which are invoked on all of the widgets in the named hierarchy.

2.2.1 Action Listeners

Often there is a need to make different types of action calls to one widget. Aranea framework offers a way which allows to use actions more flexibly. Analogously to events, actions can be differentiated by action listener name, which is read from a request parameter (serviceActionHandler). When an action is routed to a widget and the action has action listener name set, only the matching listener callbacks are invoked. The following method is used to attach action listeners to services and widgets:

```java
void addActionListener(Object actionId, ActionListener listener);

interface ActionListener extends Serializable {
  void processAction(Object actionId, InputData input, OutputData output);
}
```

Action listeners allow better separation of code. When several listeners are added with the same name, all of them are invoked, in the same order as they were added.

2.2.2 Example: AutoCompleteTextControl

AutoCompleteTextControl is a user interface component bundled with Aranea Uilib that makes use of action calls. Its behavior is similar to the text input box found in Google Suggest application [GS]. It uses regular HTML text input form element for text input. When typing a few characters, a list of suggestions appears below the text box. In addition to using the text box as a normal input facility, the user can also select a suitable match from the list (with either keyboard or mouse).

The Aranea implementation of AutoCompleteTextControl extends TextControl, which is a regular text input form element. The client-side JavaScript code of the control is from script.aculo.us library [SJ].

The autocomplete control is rendered as a regular text input box (HTML input tag with type="text"). The additional autocompleting behavior is attached to the text box with JavaScript code. After a character is typed in the text box, an Ajax request to the server is initiated in the background. The request specifies autocomplete control widget as the action path and the user-typed text as a request parameter. Since the request is asked to be routed as an action, the widget request-response cycle is bypassed. The action listener, which is invoked on the autocomplete widget, reads the user-typed text from the request parameter and finds appropriate matches to it. It then constructs the HTML for outputting the matches and writes it directly to the response. When the Ajax request completes in the browser, a JavaScript callback function is invoked, that parses the response from the autocomplete widget and displays the list of matches accordingly.

The autocomplete control can be used in the same way as regular text input control. The only difference is that you have to provide a `DataProvider` that constructs the list of suggestions. The following code demonstrates the usage of autocomplete control in a form:

```
AutoCompleteTextControl actc = new AutoCompleteTextControl();
actc.setDataProvider(new DataProvider() {
  public List getSuggestions(String input) {
    // return a list of Strings
  }
});
form.addElement("acinput", "#Label", actc, new StringData(), false);
```

The following code would be placed in a JSP template:

```
<ui:autoCompleteTextInput id="acinput"/>
```

2.3 Actions in JavaScript API

In the course of this work, the JavaScript API of Aranea framework was enhanced with support for action calls. The following functions were added to AraneaPage:

```
function action(element, actionId, actionTarget, actionParam,
         actionCallback, options);
function action_6(systemForm, actionId, actionTarget, actionParam,
         actionCallback, options);
```

Their use is analogous to the usage of `event` and `event_6` functions. An event causes the whole form to be submitted, and the widget request-response cycle to be invoked. Form submission means that the browser will initiate a HTTP request and the user has to wait for the new page to be loaded.

An action is a lightweight call compared to an event. In case of action, an asynchronous Ajax request is initiated in the background. The web page in the browser is not changed and the user can continue working with it while the Ajax request is processed in the server. When the request completes, a JavaScript callback function is invoked, that processes the results.

Under Aranea JavaScript API's hood, the Ajax request is initiated with the help of Prototype library [PJ], which conceals the browser-specific differences of the whole process.

The following JavaScript fragment demonstrates initiating an action call to the widget that is rendering the JSP template (`${widgetId}` evaluates to enclosing widget id):

```
araneaPage().action(null, 'actionId', '${widgetId}', 'someValue',
function(transport) {
  // do something with transport.responseText
});
```

2.4 Unsynchronized Actions

A general framework contract states that widgets model non-reentrant (stateful) controllers. In order to prevent re-entrancy to the widget hierarchy, there exists a synchronizing filter service that lets only one request proceed at a time.

With the usage of Ajax requests to call actions in the background, there exists a case when some actions take considerable amount of time and multiple such actions are needed to be called simultaneously. The synchronizing filter service in Aranea 1.0 makes this impossible to achieve: only one request is processed at a time, which results all of simultaneously fired requests to be processed sequentially.

In the course of this work Aranea synchronizing filter service was enhanced to allow multiple requests to be processed simultaneously.

In Aranea 1.1 `StandardHttpSessionRouterService` divides requests to two categories: synchronized and unsynchronized requests. All unsynchronized requests are processed immediately. The synchronized requests are processed sequentially.

The decision whether to synchronize the request is made using by checking the presence and value of a request parameter (`sync=false`). Backwards compatibility is fully preserved: the requests are synchronized by default.

`StandardHttpSessionRouterService` handles propagating session updates to cluster nodes. This is performed by taking the component hierarchy below it and serializing it into a session attribute. When a `HttpSession` attribute is set, application server will propagate the changed session to other cluster nodes. The serialization of component hierarchy can only be performed when there are no requests being processed inside that component hierarchy.

The unsynchronized action enhancements introduced in Aranea 1.1 make the tracking of the requests more complicated. It now involves using a read-write lock (which prefers waiting readers over waiting writers). All the requests that are processed acquire a read lock and every time a request completes a write lock is attempted to obtain. If obtained, the write lock guarantees that no other requests are processed during the usage of the lock and it is safe to serialize the component hierarchy.

The following code is a simplified version of the synchronization logic in the filter, demonstrating only the request tracking aspect (for component hierarchy serialization purpose):

```
private ReadWriteLock lock = new ReaderPreferenceReadWriteLock();
protected void action(Path path, InputData input, OutputData output) {
  lock.readLock().acquire();
  try {
    // propagate action call to child service
  } finally {
    lock.readLock().release();
  }
  synchronized (this) {
    // attempt to acquire the write lock, fail immediately if unable
    if (lock.writeLock().attempt(0)) {
      try {
        // serialize component hierarchy and write it to a session
attribute
      } finally {
        lock.writeLock().release();
      }
    }
  }
}
```

In Aranea 1.0, all requests are synchronized, which means that at the end of every request, `StandardHttpSessionRouterService` can always safely perform the serialization process. In Aranea 1.1 however, when there is a constant flow of overlapping unsynchronized requests, the opportunity to propagate session updates to cluster nodes may occur rarely or even never.

The Aranea JavaScript API is also enhanced to support unsynchronized actions. The following functions gained an additional parameter, that controls the synchronization logic:

```
function getActionSubmitURL(systemForm, actionId, actionTarget,
actionParam, sync);
```

```
function action(element, actionId, actionTarget, actionParam,
actionCallback, options, sync);
function action_6(systemForm, actionId, actionTarget, actionParam,
actionCallback, options, sync);
```

3 Partial Rendering

This chapter introduces Aranea's feature of updating web page regions via Ajax requests, problems associated with its current implementation and steps necessary to implement partial rendering.

3.1 Update Regions

In Aranea all events in the web page, such as clicking a link or a button, cause the system form to be submitted, in turn causing a full update of the page. When only a small part of the page changes, the resources that are spent on composing, transmitting, loading and rendering the parts of the page that remain identical can be considered wasted. The server has to render the whole page again, all of it has to be transmitted back to the browser and loading a huge page consumes considerable amount computing power of the client-side host. It is desirable to communicate small changes to the page back to the browser in a more efficient way.

In Aranea there is a feature, that allows to do partial page updating. Browser initiates an Ajax HTTP request in the background. All of the widget hierarchy is rendered as usual in the server side, but only some parts of the page are extracted, sent back to the browser in HTTP response and their contents replaced in browser with JavaScript.

The page regions that are dynamically updatable, are called update regions. These are defined in JSP templates by `ui:updateregion` tags. Each region is given a unique identifier. The `ui:updateregion` tag writes a HTML *span* tag (or `tbody` tag in the case of `ui:updateregionrows` tag) and HTML comments around the region contents. The comments are used to identify the region in the server-side filter, that extracts the region contents from the rest of the page. The span is used to do client-side replacement of the region contents in browser, using DOM facilities.

For example, given the following fragment of a JSP page for the widget with a full id of `foo.bar`

```
<ui:updateRegion id="numberRegion">
  The generated random number is
  <c:out value="${widget.randomNumber}"/>.
</ui:updateRegion>
```

the result of rendering the update region tags and its content would be:

```
<span id="foo.bar.numberRegion">
  <!--BEGIN:foo.bar.numberRegion-->
    The generated random number is 13.
  <!--END:foo.bar.numberRegion-->
</span>
```

When an event occurs in the page, the default behavior is to do a form submit which results in a full page reload. The usage of update regions can be enabled by adding `updateRegions` attribute to a tag that would cause the event (button, link, any control). The value of that attribute must contain comma-separated list of region identifiers, for example:

```
<ui:eventButton eventId="generateRandomNumber"
                updateRegions="numberRegion,someOtherRegion"/>
```

When an event occurs in the page and the event source has any update regions specified, then instead of a regular form submit an Ajax request to the server is initiated. The request contains all the form elements values as a normal form submit would, plus a request parameter with comma-separated identifiers of the update regions.

When the request arrives at server-side, it passes through `StandardUpdateRegionFilterService` filter. That filter detects the presence of the request attribute, that contains the identifiers of the update regions. If this request attribute is absent, the filter lets the request through unmodified and takes no further action. In the opposite case the filter lets the request execute as normal, but captures the output data that would be written to the response. For each region that is listed in the request parameter the unique start and end region identifiers (HTML comments that contain the region names) are searched from the captured data. These identifiers serve as markers, which are used in extracting the region contents. The data that was written to response is rolled back and `StandardUpdateRegionFilterService` writes its own data to the response: pairs containing the identifiers and contents of the regions that were extracted.

In browser, on completion of the Ajax request, the JavaScript code, for each pair, searches for an element from the document with the unique identifier of the update region (`document.getElementById`), and replaces its current contents with new one (HTMLElement's `innerHTML` property allows to conveniently change it).

3.2 Rendering Encapsulation

The update regions feature offers speedup in server-client data transmission and browser page updating, by reducing the size of the data, but it does not improve server-side page rendering. The render phase in the widget hierarchy is still carried out in full. This means the fewer changes page has during an event, the more processing power is wasted on composing parts of the page that will be thrown away and the more speedup there is to gain by implementing partial rendering. The goal of this chapter is to examine what changes in Aranea framework are needed to implement an update regions implementation with partial rendering.

Partial rendering means that instead of always having to start the render phase from the root widget (and thus rendering all widgets in the hierarchy), the render method could be called on only those widgets that contain the regions we wish to extract. This is not trivial to accomplish, because there are two kinds of dependencies during the render phase:

(a) Some widgets depend on the information that is provided by other widgets during the rendering phase. If the rendering of some widgets is omitted, then some data is not made available and rendering may fail.

(b) When tags are rendered, some of them make data available to the tags below them. While passing the data between tags of the same widget's template does not impose a problem, some relations cross the borders of widgets. In the latter cases, the rendering may fail if rendering of some widgets is omitted.

3.2.1 Widget Dependencies

The widgets that cause the first type of dependencies make use of the data exchange features of `OutputData`:

```
void pushAttribute(Object key, Object value);
Object popAttribute(Object key);
Object getAttribute(Object key);
```

The problem is not that widgets publish information to their descendants, but that they do it only during the rendering phase. The problem can be solved by providing the data via environment, this way it is always accessible and does not depend on a specific phase of the widget request-response cycle. After that, the `OutputData` attributes can be removed altogether to simplify the framework.

3.2.2 Tag Dependencies

Tags exchange information during rendering via the facilities of JSP `PageContext` (its request scope). The following methods from `BaseTag` class, which all Aranea Uilib tags extend, are used:

```
protected void addContextEntry(String key, Object value);
protected Object getContextEntry(String key);
protected Object requireContextEntry(String key);
```

As stated earlier, exchanging data between tags of the same widget is not a problem, because a widget is the smallest unit of rendering. However, information exchange between tags which are rendered under different widgets causes problematic dependencies. The solution to this problem is to restrict the visibility of information that can be published by tags. It means that internal communication stays the same, but external communication must be solved another way.

The information visibility restriction is applied on widget borders. It means that we deliberately forbid carrying JSP `PageContext` (request scope) data over to another widget. This enforces a cleaner encapsulation of widget rendering and minimize dependencies during rendering phase. Better encapsulation and fewer dependencies between widgets simplify application code and enable simpler reuse of widgets.

While many portions of application code and JSP templates can be refactored to fit with the new rendering encapsulation requirements, there are some framework tags that don't fit with it and need to be completely reworked. Several Uilib tags in Aranea 1.1 were removed and others were changed to fit with the new approach.

3.3 Partial Rendering with Update Regions

With rendering encapsulation in Aranea 1.1, the `StandardUpdateRegionFilterService` filter can be enhanced to use partial rendering.

Given a list of update region names, the filter has to figure out which widget each region belongs to. Although update regions have their identifier composed by prefixing their name with enclosing widget identifier, this process may not always be reversible because the name that was given to the update region in the template may contain dots or it may overlap with the name of a child widget of the context widget. There also exist global update regions, which have a globally unique names – the enclosing widget identifier is not prefixed to their names. Because of that, there is no way to tell which widget a global update region belongs to.

The update region filter makes itself available in the environment using the following interface introduced in Aranea 1.1:

```
interface UpdateRegionContext {
  void addDocumentRegion(String documentRegionId, String widgetId);
}
```

The `ui:updateRegion` and `ui:updateRegionRows` tags will use that context to register the region and the widget that it is rendered in. That way the filter can later match the region identifiers to appropriate widgets.

Using the map that associates update region names with widgets, the `StandardUpdateRegionFilterService` filter creates a set of widget names to render and for each widget name, a set of update region names to extract. The set of widget names will be processed in such a way that using the resulting set, no widget will be rendered more than once (a descendant widget will be removed from the set if any of its parents are present; the set of update regions of the descendant widget will be appended to the parent's update region set). This is needed because Aranea's rendering model is hierarchical: a widget itself is responsible for rendering its children.

For each widget in the resulting set, a `Message` is sent to it that invokes the `render()` method on that widget. Similarly to the update region filter behavior previously, the data that was written to the response during rendering is captured and update region contents are extracted from that. The output data is rolled back and the filter writes to the response pairs consisting of update region identifiers and contents. On the client-side the response is processed by JavaScript code that updates each region's contents in the web page.

3.4 Update Regions and JavaScript API

In addition to page region updates, there may be several more accompanying tasks for each update regions Ajax request:

(a) updating message area contents;

(b) updating transaction id value in hidden form element;

(c) opening popup windows;

(d) reloading page entirely.

In Aranea 1.0, tasks (a) and (c) are accomplished with their own dedicated global update regions. Tasks (b) and (d) are handled specially by the update regions JavaScript implementation.

Having special cases in the implementation for specific tasks is not a very good design. One alternative would be to use dedicated global update regions for all of tasks above. The update region contents for tasks (c) and (d), maybe even (b), would contain only JavaScript code. These tasks could be more elegantly solved by sending only the essential data with the response, not the JavaScript code.

In an attempt to address the aforementioned problems, a more generic way of passing data to client-side during Ajax requests is introduced in Aranea 1.1. All of the data fragments that are returned with an Ajax request from the update region filter are called regions. Regions are classified by type. There may be more than one region of each type in the result. Update region filter itself constructs page regions from partial rendering results and also transaction id and reload regions. Other regions (popup windows, messages) are constructed by other services and collected from them by update region filter.

In client-side each region type has a handler function, that is called when region data is received from an Ajax response. The registration of region handlers can be performed in Aranea JavaScript API like this:

```
AraneaPage.addRegionHandler( 'regionTypeName', {
  process: function(content) {
    // Do something with content
  }
});
```

Currently passing data to client-side with regions is used only with Ajax requests targeted to update region filter. In the future we wish to extend this to actions too. This would make server-client data communication even more frequent and transparent to the programmer.

Conclusions

The main goal of this work was to introduce new Ajax-related features to Aranea web framework.

Several modifications to the framework were performed to support rendering encapsulation. Rendering encapsulation eliminates dependencies during the render phase of widget request-response cycle. It was a necessity for implementing partial rendering in Aranea update region filter. As a result it is no longer necessary to always render the whole widget hierarchy and rendering widgets separately is automatically supported by design. This gives a significant performance boost to updating web page parts by Ajax requests, which is one of the central Ajax-related features in Aranea.

Aranea synchronization filter was enhanced to support unsynchronized actions. This allows multiple lengthy Ajax request to be processed simultaneously by framework components.

Aranea JavaScript API saw several improvements. Partial rendering moved some processing logic to client-side, introducing a generic mechanism of transferring custom data from server-side framework services to client-side JavaScript code. Making action calls and unsynchronized action calls programmatically is now well supported by Aranea JavaScript API.

Altogether several simplification to core framework principles were made. The changes decreased code complexity in the framework and should help towards easier maintainability of applications developed with Aranea.

Aranea Ajax

Alar Kvell

Bakalaureusetöö

Kokkuvõte

Viimastel aastatel on paljud veebirakendused tulnud tavalistele töölauarakendustele omaduste ja võimaluste vallas üha lähemale. Uute tehnoloogiate, nagu Ajax, kasutusele võtmine võimaldab veebirakendustel pakkuda palju interaktiivsemat kasutajakogemust, mis oli varem omane ainult töölauarakendustele.

Käesoleva töö põhieesmärkideks on Aranea veebiraamistiku Ajax tehnoloogiate kasutamisest ülevaate omandamine ning veebiraamistiku Ajax-võimaluste täiustamine.

Antud töö käsitleb muudatusi, mis Aranea veebiraamistikus sisse viidi: *action* kutsete toe lisamine JavaScripti API-sse ja mittesünkroniseeritud *action* kutsete toe lisamine veebiraamistiku sünkroniseerimise filtrisse.

Antud töö kirjeldab ka üht põhilist Ajax'it kasutavat võimalust Aranea's: veebilehe uuendamist osade kaupa (*update regions*). Töö uurib probleeme selle praeguses realisatsioonis ning kirjeldab esitluse kapseldamise ideed ning selle abil osalise joonistamise realiseerimist. Antud täiendused võimaldavad olulist kiirusevõitu veebilehe osade kaupa uuendamisel.

Kõik muudatused on saadaval Aranea veebiraamistiku väljalaske 1.1-M1 koosseisus. Kokkuvõttes tehti olulisi lihtsustusi mitmetesse Aranea raamistiku põhiprintsiipidesse. Tehtud muudatused vähendasid koodi keerukust raamistikus ning peaksid kaasa aitama Aranea raamistikuga arendatud rakenduste koodi paremale hallatavusele.

Bibliography

[Fi99] R. Fielding et al. *Hypertext Transfer Protocol (HTTP/1.1).* 1999. http://www.w3.org/Protocols/HTTP/1.1/rfc2616.pdf (28.05.2007)

[Ga05] J. J. Garrett. *Ajax: A New Approach to Web Applications*. 2005. http://www.adaptivepath.com/publications/essays/archives/000385.php (28.05.2007)

[Li06] T. Liebeck. *Comparing the Google Web Toolkit to Echo2*. 2006. *http://www.theserverside.com/news/thread.tss?thread_id=40804 (28.05.2007)*

[MK06] O. Mürk, J. Kabanov. *Aranea: web framework construction and integration kit*. In *PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 163–172, New York, NY, USA, 2006. ACM Press.

[ZM06] N. C. Zakas, J. McPeak, J. Fawcett. *Professional Ajax.* Wiley Publishing, Inc., Indianapolis, IN, USA, 2006.

[AN] *About Native XMLHTTP.* http://msdn2.microsoft.com/en-us/library/ms537505.aspx (28.05.2007)

[DW] *Direct Web Remoting.* http://getahead.org/dwr (28.05.2007)

[DO] *Document Object Model.* http://www.w3.org/DOM/ (28.05.2007)

[EW] *Echo2 Web Framework.* http://www.nextapp.com/platform/echo2/echo/ (28.05.2007)

[EL99] *ECMAScript Language Specification (ECMA-262).* 1999. http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf (28.05.2007)

[Gi] Gmail http://gmail.google.com/ (28.05.2007)

[GM] Google Maps http://maps.google.com/ (28.05.2007)

[GS] *Google Suggest.* *http://www.google.com/webhp?complete=1&hl=en* (28.05.2007)

[GW] *Google Web Toolkit.*
http://code.google.com/webtoolkit/ (28.05.2007)

[HD] *HTML & DHTML Reference.*
http://msdn.microsoft.com/library/default.asp?url=/workshop/author/dhtml/reference/dhtml_reference_entry.asp (28.05.2007)

[JJ] *jQuery JavaScript Library.*
http://jquery.com/ (28.05.2007)

[PJ] *Prototype JavaScript Framework.*
http://www.prototypejs.org/ (28.05.2007)

[SJ] *script.aculo.us JavaScript library.*
http://script.aculo.us/ (28.05.2007)

[SS] *SWT: The Standard Widget Toolkit.*
http://www.eclipse.org/swt/ (28.05.2007)

[EH02] *The Extensible HyperText Markup Language*. 2002.
http://www.w3.org/TR/xhtml1/ (28.05.2007)

[ST] *The Swing Tutorial.*
http://java.sun.com/docs/books/tutorial/uiswing/ (28.05.2007)