Aranea—Web Framework Construction and Integration Kit

Oleg Mürk
Dept. of Computer Science and Engineering,
Chalmers University of Technology,
SE-412 96 Göteborg, Sweden
oleg.myrk@gmail.com

Jevgeni Kabanov
Dept. of Computer Science,
University of Tartu,
J. Liivi 2, EE-50409 Tartu, Estonia
ekabanov@gmail.com

ABSTRACT

Currently there exist dozens of web controller frameworks that are incompatible, but at the same time have large portions of overlapping functionality that is implemented over and over again. Web programmers are facing limitations on code reuse, application and framework integration, extensibility, expressiveness of programming model and productivity.

In this paper we propose a minimalistic component model *Aranea* that is aimed at constructing and integrating server-side web controller frameworks in Java. It allows assembling most of available web programming models out of reusable components and patterns. We also show how to integrate different existing frameworks using Aranea as a common protocol. In its default configuration Aranea supports both developing sophisticated user interfaces using stateful components and nested processes as well as high-performance stateless components.

We propose to use this model as a platform for framework development, integration and research. This would allow combining different ideas and avoid reimplementing the same features repeatedly. An open source implementation of Aranea framework together with reusable controls, such as input forms and data lists, and a rendering engine are ready for real-life applications.

1. INTRODUCTION

During the last 10 years we have witnessed immense activity in the area of web framework design. Currently, there are more than 30 actively developed open source web frameworks in Java [10], let alone commercial products or other platforms like .NET and numerous dynamic languages. Not to mention in-house corporate frameworks that never saw public light. Many different and incompatible design philosophies are used, but even within one approach there are multiple frameworks that have small implementation differences and are consequently incompatible with each other.

The advantage of such a situation is that different approaches and ideas are tried out. Indeed, many very good ideas have been proposed during these years, many of which we will describe later in this paper. On a longer timescale the stronger (or better marketed) frameworks and approaches will survive, the weaker will diminish. However, in our opinion, such situation also has a lot of disadvantages.

PPPJ 2006, August 30–September 1, 2006, Mannheim, Germany.

1.1 Problem Description

First of all let's consider the problems of the web framework ecosystem from the viewpoint of application development. Framework user population is very fragmented as a result of having many incompatible frameworks with similar programming models. Each company or even project, is using a different web framework, which requires learning a different skill set. As a result, it is hard to find qualified work force for a given web framework. For the same reason it is even harder to reuse previously developed application code.

Moreover, it is sometimes useful to write different parts of the same application using different approaches, which might prove impossible, as the supporting frameworks are incompatible. Portal solutions that should facilitate integrating disparate applications provide very limited ways for components to communicate with each other. Finally, frameworks are often poorly designed, limiting expressiveness, productivity and quality.

System programmers face additional challenges. Creators of reusable components have to target one particular framework, consequently their market shrinks. Framework designers implement overlapping features over and over again, with each new feature added to each framework separately. Many useful ideas cannot be used together because they have been implemented in different frameworks.

We think that web framework market would win a lot if there were two or three popular platforms with orthogonal philosophies that would consolidate proponents of their approach. Application programmers would not have to learn a new web framework at the beginning of each project. Writing reusable components and application integration would be easier and more rewarding. Framework designers could try out new ideas much easier by writing extensions to the platform and targeting a large potential user-base.

1.2 Contributions

In this paper we will describe a component framework that we named *Aranea*. Aranea is written in Java and allows assembling server-side controller web frameworks out of reusable components and patterns. Aranea applications are pure Java and can be written without any static con-

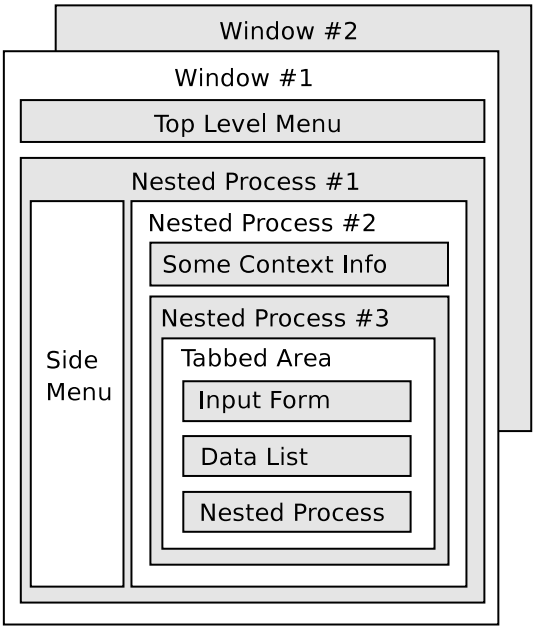

Figure 1: A sketch of a rich user interface.

figuration files. In Section 2 we describe our approach and motivation. We find that one of the strengths of this framework is its conceptual integrity—it has very few core concepts that are applied uniformly throughout the framework. The number of core interfaces is small, as is the number of methods in the interfaces. Components are easy to reuse, extend and test, because all external dependencies are injected into them. The details of the Aranea core abstractions are explained in Section 3.

In different configurations of Aranea components we can mimic principles and patterns of most existing server-side web controller frameworks as well as combine them arbitrarily. Possible configurations are described in Section 4. We concentrate on implementation of server-side controllers, but we also intend to support programming model where most of UI is implemented on the client-side and server-side contains only coarse-grained stateful components corresponding roughly to active use-cases.

Of particular interest is the configuration supporting programming model that allows expressing a rich user interface as a dynamic hierarchical composition of components that maintain call stacks of nested processes (we refer to such processes as *flows* further on). As an example of rich user interface at extreme, consider Figure 1: multiple interacting windows per user session, each window contains a stack of flows, flows can call nested flows that after completing return values, flows can display additional UI (side-menu, context information) even when a nested flow is executing, flows can contain tabbed areas, wizards, input forms, lists, other controls and even other flows.

Further, the framework facilitates both event-based and sequential programming (using continuations). The programming model is quite similar to the one used in the Smalltalk web framework Seaside [21], but has completely different implementation and is more general in terms of where sequential programming can be applied. This topic is discussed in Section 7.1 as one of extensions.

All web frameworks have to handle such aspects as configuration, security, error handling and concurrency. We explain how Aranea handles these issues in Section 5.

One of the most important differentiating factors of Aranea, is in our mind its ability to serve as a vehicle for integration of existing frameworks due to its orthogonal design. We discuss this topic in Section 6.

Finally, we see Aranea as a research platform. It it very easy to try out a new feature without having to write an entire web framework. A framework is assembled out of independent reusable components, so essentially everything can be reconfigured, extended or replaced. If Aranea becomes popular, writing a new component for Aranea would also mean a large potential user base.

Naturally, there are still numerous framework extensions to be made and further directions to be pursued. These are described in Section 7. Last, but not least, Aranea is based on great ideas originating from prior work of many people. When possible, we reference the original source of idea at the time of introducing it. In Section 8 we compare Aranea with some of the existing frameworks.

2. BACKGROUND

As we mentioned in the introduction, our aim is to support most of programming models and patterns available in the existing controller frameworks. We present here, unavoidably incomplete and subjective, list of profound ideas used in contemporary web controller frameworks.

The first important alternative is using stateless or reentrant components for high performance and low memory footprint, available in such frameworks as Struts [3] and WebWork [19].

Another important approach is using hierarchical composition of stateful non-reentrant components with event-based programming model, available in such frameworks as JSF [8], ASP.NET [4], Seaside [21], Wicket [20], Tapestry [5]. This model is often used for developing rich UI, but generally poses higher demands on server's CPU and memory.

The next abstraction useful especially for developing rich UI is nested processes, often referred to as *modal* processes, present for instance in such web frameworks as WASH [26], Cocoon [2], Spring Web Flow [18] and RIFE [14]. They are often referred to by the name of implementation mechanism—*continuations*. The original idea comes from Scheme [25], [22].

All of these continuation frameworks provide one top-level call-stack—essentially flows are like function calls spanning multiple web requests. A significant innovation can be found in framework Seaside [21], where continuations are combined with component model and call stack can be present at any level of component hierarchy.

Yet another important model is using asynchronous requests and partial page updates, coined Ajax [1]. This allows decreasing server-side state representation demands and increases responsiveness of UI. At the extreme, this allows creating essentially fat client applications with a sophisticated UI within browser.

We would also like to support different forms of metaprogramming such as domain specific language for describing

UI as a state machine in Spring Web Flow [18] and domain-driven design as implemented in Ruby on Rails [16] or RIFE/Crud [15]. This often requires framework support for dynamic composition and component run-time configuration.

3. CORE ABSTRACTIONS

Aranea framework is based on the abstraction of components arranged in a dynamic hierarchy and two component subtypes: services that model reentrant controllers and widgets that model non-reentrant stateful controllers. In this section we examine their interfaces and core implementation ideas. We omit some non-essential details for brevity.

3.1 Components

At the core of Aranea lies a notion of components arranged into a dynamic hierarchy that follows the *Composite* pattern extended with certain mechanisms for communication. This abstraction is captured by the following interface:

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

A component is an entity that

- Has a life-cycle that begins with an `init()` call and ends with a `destroy()` call.
- Can be signaled to be disabled and then enabled again.
- Has an `Environment` that is passed to it by its parent or creator during initialization.
- Can propagate `Message`s to its children.

We imply here that a component will have a parent and may have children. Aranea actually implies that the component would realize a certain flavor of the *Composite* pattern that requires each child to have a unique identifier in relation to its parent. These identifiers can then be combined to create a full identifier that allows finding the component starting from the hierarchy root. Note that the hierarchy is not static and can be modified at any time by any parent.

The hierarchy we have arranged from our components so far is inert. To allow some communication between different components we need to examine in detail the notions of `Environment` and `Message`.

`Environment` is captured by the following interface:

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

Environment is a discovery mechanism allowing children to discover services (named *contexts*) provided by their parents without actually knowing, which parent has provided it. Looking up a context is done by calling the environment `getEntry()` method passing some well-known context name as the key. By a convention this well-known name is the interface class realized by the context. The following example illustrates how environment can be used:

```
L10nContext locCtx = (L10nContext)
  getEnvironment().getEntry(L10nContext.class);
String message = locCtx.localize("message.key");
```

Environment may contain entries added by any of the current component ancestors, however the current component direct parent has complete control over the exact entries that the current component can discover. It can add new entries, override old ones as well as remove (or rather filter out) entries it does not want the child component to access. This is done by wrapping the grandparent `Environment` into a proxy that will allow only specific entries to be looked up from the grandparent.

`Message` is captured in the following interface:

```
interface Message {
  void send(Object key, Component comp);
}
```

While the environment allows communicating with the component parents, messages allow communicating with the component descendants (indirect children). `Message` is basically an adaptation of the *Visitor* pattern to our flavor of *Composite*. The idea is that a component `propagate(m)` method will just call message `m.send(...)` method for each of its children passing the message both their instances and identifiers. The message can then propagate itself further or call any other component methods.

It is easy to see that messages allow constructing both broadcasting (just sending the message to all of the components under the current component) and routed messages that receive a relative "path" from the current component and route the message to the intended one. The following example illustrates a component broadcasting some message to all its descendants (`BroadcastMessage` will call `execute` for all component under current):

```
Message myEvent = new BroadcastMessage() {
  public void execute(Component comp) {
    if (comp instanceof MyDataListener)
      ((MyDataListener) comp).setMyData(data);
  }
}
myEvent.send(null, rootComponent);
```

3.2 Services

Although component hierarchy is a very powerful concept and messaging is enough to do most of the communication, it is comfortable to define a specialized component type that is closer to the *Controller* pattern. We call this component `Service` and it is captured by the following interface:

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

`Service` is basically an abstraction of a reentrant controller in our hierarchy of components. The `InputData` and `OutputData` are simple generic abstractions over, correspondingly, a request and a response, which allow the controller to process request data and generate the response. The `Path` is an abstracted representation of the full path to

the service from the root. It allows services to route the request to the one service it is intended for. Since service is also a component it can enrich the environment with additional contexts that can be used by its children.

3.3 Widgets

Although services are very flexible, they are not too comfortable for programming stateful non-reentrant components (GUI abstractions often being such). To do that we introduce the notion of a `Widget`, which is captured by the following interface:

```
interface Widget extends Service {
  void update(InputData data);
  void event(Path path, InputData input);
  void process();
  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 the 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. `process()` is also called on all the widgets in the hierarchy allowing them to prepare for rendering whether or not the widget has received an event.

4. `render()` calls are not guided by any conventions. If called, widget should render itself (though it may delegate the rendering to e.g. template). The `render()` method should be idempotent, as it can be called arbitrary number of times after a `process()` call before an `update()` call.

Although widgets also inherit an `action()` method, it may not be called during the widget request-response cycle. The only time it is allowed is after a `process()` call, but before an `update()` call. It may be used to interact with a single widget, e.g. for the purposes of making an asynchronous request through Ajax [1].

Standard widget implementation allows setting event listeners that enable further discrimination between `action()`/`event()` calls to the same widget.

So far we called our components stateful or non-stateful without discussing the *persistence* of this state. A typical framework would introduce predefined scopes of persistence, however in Aranea we have very natural scopes for all our components—their lifetime. In Aranea one can just use the component object fields and assume that they will persist until the component is destroyed. If the session router is used, then the root component under it will live as long as the user session. This means that in Aranea state management is invisible to the programmer, as most components live as long as they are needed.

3.4 Flows

To support flows (nested processes) we construct a flow container widget that essentially hosts a stack of widgets (where only the top widget is active at any time) and enriches their environment with the following context:

```
interface FlowContext {
  void start(Widget flow, Handler handler);
  void replace(Widget flow);

  void finish(Object result);
  void cancel();
}
```

This context is available in standard widget implementation by calling `getFlowCtx()`. Its methods are used as follows:

- Flow `A` running in a flow container can start a nested flow `B` by calling `start(new B(...), null)`. The data passed to the flow `B` constructor can be thought as incoming parameters to the nested process. The flow `A` then becomes inactive and flow `B` gets initialized.

- When flow `B` is finished interacting with the user, it calls `finish(...)` passing the return value to the method. Alternatively flow `B` can call the `cancel()` method if the flow was terminated by user without completing its task and thus without a return value. In both cases flow `B` is destroyed and flow `A` is reactivated.

- Instead of finishing or canceling, flow `B` can also replace itself by a flow `C` calling `replace(new C(...))`. In such a case flow `B` gets destroyed, flow `C` gets initialized and activated, while flow `A` continues to be inactive. When flow `C` will finish flow `A` will get reactivated.

`Handler` callback interface is used when the calling flow needs to somehow react to the called flow finishing or canceling:

```
interface Handler {
  void onFinish(Object returnValue);
  void onCancel();
}
```

It is possible to use continuations to realize synchronous (blocking) semantics of flow invocation, as shown in the section 7, in which case the `Handler` interface is redundant.

4. FRAMEWORK ASSEMBLY

Now that we are familiar with the core abstractions we can examine how the actual web framework is assembled. First of all it is comfortable to enumerate the component types that repeatedly occur in the framework:

Filter A component that contains one child and chooses depending on the request parameters whether to route calls to it or not.

Router A component that contains many children, but routes calls to only one of them depending on the request parameters.

Broadcaster A component that has many children and routes calls to all of them.

Adapter A component that translates calls from one protocol to another (e.g. from service to a widget or from Servlet [6] to a service).

Container A component that allows some type of children to function by enabling some particular protocol or functionality.

Of course of all of these component types also enrich the environment and send messages when needed.

Aranea framework is nothing, but a hierarchy (often looking like a chain) of components fulfilling independent tasks. There is no predefined way of assembling it. Instead we show how to assemble frameworks that can host a flat namespace of reentrant controllers (á la Struts [3] actions), a flat namespace of non-reentrant stateful controllers (á la JSF [8] components) and nested stateful flows (á la Spring Web Flow [18]). Finally we also consider how to merge all these approaches in one assembly.

4.1 Reentrant Controllers

The first model is easy to implement by arranging the framework in a chain by containment (similar to pattern *Chain-of-Responsibility*), which starting from the root looks as follows:

1. Servlet [6] adapter component that translates the servlet `doPost()` and `doGet()` to Aranea service `action()` calls.
2. HTTP filter service that sets the correct headers (including caching) and character encoding. Generally this step consists of a chain of multiple filters.
3. URL path router service that routes the request to one of the child services using the URL path after servlet. One path will be marked as default.
4. A number of custom application services, each registered under a specific URL to the URL path router service that correspond to the reentrant controllers. We call these services *actions*.

The idea is that the first component object actually contains the second as a field, the second actually contains the third and so on. Routers keep their children in a `Map`. When `action()` calls arrive each component propagates them down the chain.

The execution model of this framework will look as follows:

- The request coming to the root URL will be routed to the default service.
- When custom services are invoked they can render an HTML response (optionally delegating it to a template) and insert into it URL paths of other custom services, allowing to route next request to them.
- A custom service may also issue an HTTP redirect directly sending the user to another custom service. This is useful when the former service performs some action that should not be repeated (e.g. money transfer).

Note that in this assembly `Path` is not used at all and actions are routed by the request URL.

Both filter and router services are stateful and reentrant. Router services could either create a new stateless action for each request (like WebWork [19] does) or route request to existing reentrant actions (like Struts [3] does). Router services could allow adding and removing (or enabling and disabling) child actions at runtime, although care must be taken to avoid destroying action that can be active on another thread.

We have shown above how analogues of Struts and WebWork actions fit into this architecture. WebWork interceptors could be implemented as a chain of filter services that decide based on `InputData` and `OutputData` whether to enrich them and then delegate work to the child service. There could be filter services both before action router and after. The former would be shared between all actions while the latter would be private for each action instance.

4.2 Stateful Non-Reentrant Controllers

To emulate the stateful non-reentrant controllers we will need to host widgets in the user session. To do that we assemble the framework as follows:

1. Servlet [6] adapter component.
2. Session router that creates a new service for each new session and passes the `action()` call to the associated service.
3. Synchronizing filter service that let's only one request proceed at a time.
4. HTTP filter service.
5. Widget adapter service that translates a service `action()` call into a widget `update()/event()/process()/render()` request-response cycle.
6. Widget container widget that will read from request the path to the widget that the event should be routed to and call `event()` with the correct path.
7. Page container widget that will allow the current child widget to replace itself with a new one.
8. Application root widget which in many cases is the login widget.

This setup is illustrated on Figure 2.

A real custom application would most probably have login widget as the application root. After authenticating login widget would replace itself with the actual root widget, which in most cases would be the application menu (which would also contain another page container widget as its child).

The menu would contain a mapping of menu items to widget classes (or more generally factories) and would start the appropriate widget in the child page container when the user clicks a menu item. The custom application widgets would be able to navigate among each other using the page context added by the page container to their environment.

The execution model of this framework will look as follows:

- The request coming to the root URL will be routed to the application root widget. If this is a new user session, a new session service will be created by the session router.

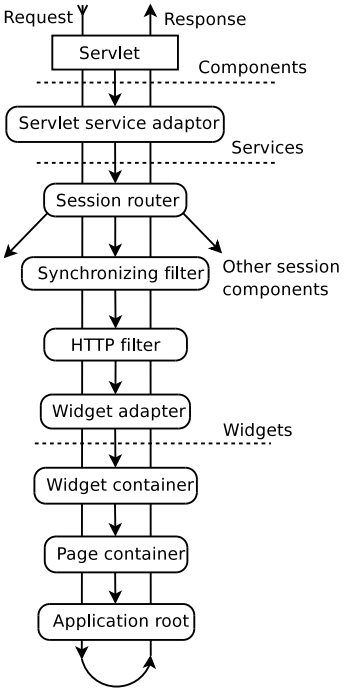

Figure 2: Framework assembly for hosting pages

- Only one request will be processed at once (due to synchronizing filter). This means that widget developers should never worry about concurrency.
- The widget may render a response, however it has no way of directly referencing other widgets by URLs. Therefore it must send all events from HTML to itself.
- Upon receiving an event the widget might replace itself with another widget (optionally passing it data as a constructor parameter) using the context provided by the page container widget. Generally all modifications of the widget hierarchy (e.g. adding/removing children) can only be done during event part of the request-response cycle.
- The hierarchy of widgets under the application root widget (e.g. GUI elements like forms or tabs) may be arranged using usual *Composite* widget implementations as no special routing is needed anymore.

In the real setup page container widget may be emulated using flow container widget that allows replacing the current flow with a new one.

Such an execution model is very similar to that of Wicket [20], JSF [8] or Tapestry [5] although these frameworks separate the pages from the rest of components (by declaring a special subclass) and add special support for markup components that compose the actual presentation of the page.

4.3 Stateful Non-Reentrant Controllers with Flows

To add nested processes we basically need only to replace the page container with a flow container in the previous model:

1. Servlet [6] adapter component.
2. Session router service.
3. Synchronizing filter service.
4. HTTP filter service.
5. Widget adapter service.
6. Widget container widget.
7. Flow container widget that will allow to run nested processes.
8. Application root flow widget which in many cases is the login flow.

The execution model here is very similar to the one outlined in Subsection 4.2. The only difference is that the application root flow may start a new subflow instead of replacing itself with another widget.

This model is similar to that of Spring WebFlow [18], although Spring WebFlow uses Push-Down Finite State Automaton to simulate the same navigation pattern and consequently it has only one top-level call stack. In our model call stacks can appear at any level of widget composition hierarchy, which makes our model considerably more flexible.

4.4 Combining the Models

It is also relatively easy to combine these models, modifying the model shown on figure 2 by putting a URL path router service before the session router, map the session router to a particular URL path and put a flow container in the end.

The combined model is useful, since reentrant stateless services allow to download files from database and send other semi-static data comfortably to the user. They can also be used to serve parts of the application that has the highest demand and thus load.

5. FRAMEWORK ASPECTS

Next we examine some typical web framework aspects and how they are realized in Aranea.

5.1 Configuration

The first aspect that we want to examine is *configuration*. We have repeated throughout the paper that the components should form a dynamic hierarchy, however it is comfortable to use a static configuration to wire the parts of the hierarchy that form the framework core.

To do that one can use just plain Java combining a hierarchy of objects using setter methods and constructors. But in reality it is more comfortable to use some configuration mechanism, like an IoC container. We use in our configuration examples Spring [17] IoC container and wire the components together as beans. Note that even such static configuration contains elements of dynamicity, since some components (á la root user session service) are wired not as instances, but via a factory that returns a new service for each session.

5.2 Security

The most common aspect of security that frameworks have to deal with is *authorization*. A common task is to determine, whether or not the current user has enough privileges to see a given page, component or GUI element. In many frameworks the pages or components are mapped to a particular URL, which can also be accessed directly by sending an HTTP request. In such cases it is also important to restrict the URLs accessible by the user to only those he is authorized to see.

When programming in Aranea using stateless re-entrant services they might also be mapped to particular URLs that need to be protected. But when programming in Aranea using widgets and flows (a stateful programming model) there is no *general* way to start flows by sending HTTP requests. Thus the only things that need protection are usually the menu (which can be assigned privileges per every menu item) and the active flow and widgets (which can only receive the events they subscribe to).

This simplifies the authorization model to checking whether you have enough privileges to start the flow *before* starting it. Since most use-cases should have enough privileges to start all their subflows it is usually enough to assign coarse-grained privileges to use-cases that can be started from the menu as well as fine-grained privileges for some particular actions (like editing instead of viewing).

5.3 Error Handling

When an exception occurs the framework must give the user (or the programmer) an informative message and also provide some recovery possibilities. Aranea peculiarity is that since an exception can occur at any level of hierarchy the informing and recovery may be specific to this place in the hierarchy. Default behavior for Aranea components is just to propagate the error up the hierarchy to the first exception handler component

For example it might be required to be able to cancel a flow that has thrown an exception and return back to the flow that invoked the faulty flow. A logical solution is to let the flow container (and other similar components) to handle their children's exceptions by rendering an informative error subpage instead in place of the flow. The error page can then allow canceling flows by sending events to the flow container.

With such approach when we have several flow containers on one HTML page, then if two or more flows under different containers fail, they will independently show error subpages allowing to cancel the particular faulty flows. Note also that such approach will leave the usual navigation elements like menus intact, which will allow the user to navigate the application as usual.

5.4 Concurrency

Execution model of Aranea is such that each web request is processed on one Java thread, which makes system considerably easier to debug. By default Aranea does not synchronize component calls. It does, however, protect from trying to destroy a working component. If a service or widget currently in the middle of some method call will be destroyed, the destroyer will wait until it returns from the call. To protect from deadlock and livelock, after some time the lock will be released with a warning.

When we want to synchronize the actual calls (as we need for example with widgets) we can use the synchronizing service that allows only one `action()` call to take place simultaneously. This service can be used when configuring the Aranea framework to synchronize calls on e.g. browser window threads. This allows to program assuming that only one request per browser window is processed at any moment of time. Note that widgets should *always* be behind a synchronizing filter and cannot process concurrent calls.

6. INTEGRATION SCENARIOS

In this section we describe our vision of how web controller frameworks could be integrated with Aranea or among each other. In practice, we have so far integrated Aranea only with one internal framework with stateful Portlet-like [12] components, where Aranea components were hosted within the latter framework, but we are considering integrating with such frameworks as Wicket [20], JSF [8], Tapestry [5], Spring WebFlow [18], Struts [3], and WebWork [19].

Integration is notorious for being hard to create generalizations about. Each integration scenario has its own set of specialized problems and we find that this article is not the right place to write about them. For this reason we keep this section intentionally very abstract and high-level and try to describe general principles of web controller framework integration without drowning in implementation details.

In the following we assume that depending on their nature it is possible to model components of frameworks we want to integrate as one of:

- *service-like*—reentrant and/or stateless component[1],
- *widget-like*—non-reentrant stateful component.

Note that both notions consist of two contracts: interface of component and contract of the container of the component.

In our abstraction we have essentially the following integration scenarios:

- service-service,
- service-widget,
- widget-service,
- widget-widget.

Here, for instance, "service-widget" should be read as: "*service*-like component of framework X containing *widget*-like component of framework Y". In homogeneous (i.e. service-service and widget-widget) integration scenarios one has to find a mapping between service (resp. widget) interface methods invocations of two frameworks. Although we do not find this mapping trivial, there is little we can say without considering specialized details of particular frameworks. However, our experience shows that, thanks to minimalistic and orthogonal interfaces and extensibility of Aranea, the task becomes more tractable than with other monolithic frameworks. We now concentrate on heterogeneous cases of server-widget and widget-service integration. They can also occur within Aranea framework itself, but are more typical when disparate frameworks using different programming models are integrated.

In service-widget scenario, generally, each web request is processed by some service and then the response is rendered

[1]Note that Servlets [6] are, for instance, service-like components.

by possibly different service, whereas both services can be reentrant and/or stateless. As a result, such services cannot host themselves the widgets whose life-time spans multiple requests handled by different services. Consequently, widget instances should be maintained in stateful widget container service(s) with longer life-span. At each request such services would call `update()` and `event()` methods of the contained widgets. Widgets would be instantiated by services processing the request and rendered using `render()` method by services generating the response. Each service processing a request should explicitly decide which widgets are to be kept further, all the rest are to be destroyed (within current session). As the services are generally reentrant, it is important to exclude concurrent access to the widgets belonging to the same session. The simplest solution is to synchronize on session at each web request that accesses widgets.

In widget-service scenario, services should be contained in service container widgets in the position within widget hierarchy most suitable for rendering the service. On widget `update()`, the data entitled for the contained service should be memorized. On widget `render()` the memorized data should be passed to the `action()` method of contained service to render the response. If the service responds with redirection, which means that the request should not be rerun, the service should be replaced with the service to which the redirection points. After that and on all subsequent renderings the `action()` method of the new service should be called with redirection parameters.

Coming back to not-so-abstract reality, when integrating frameworks the following issues should be handled with care:

- How data is represented in the web request and how output of multiple components coexists in the generated web response.
- Namespaces (e.g. field identifiers in web request) of contained components should not mix with the namespace of container components, which in general means appending to the names a prefix representing location of contained component within the container.
- State management, especially session state history management (browser's back, forward, refresh and new window navigation commands) and keeping part of the state on the client, should match between integrated components. We explore this topic further in Subsection 7.2.
- A related issue to consider is view integration. Many web frameworks support web components that are tightly integrated with some variant of templating. Consequently it is important that these templating technologies could be intermixed easily.

Incompatibilities in these aspects lead to a lot of mundane protocol conversion code, or even force modifying integrated components and/or frameworks.

Generalized solutions to these issues could be standardized as *Aranea Protocol*. As compared to such protocol, current Aranea Java interfaces are relatively loose—i.e. functionality can be considerably customized by using `Message` protocol and extending core interfaces (`InputData`, `OutputData`, `Component`) with new subintefaces.

Altogether, we envision the following integration scenarios with respect to Aranea:

Guest Aranea components (resp. services or widgets) are hosted within components of framework X that comply to the Aranea component (resp. service or widget) container contract.

Host Aranea components host components (resp. service-like or widget-like) of framework X through an adapter component that wraps framework X components into Aranea component (resp. service or widget) interface.

Protocol Framework X components provide Aranea component (resp. service or widget) container contract that hosts framework Y components wrapped into Aranea component (resp. service or widget) interface using an adapter component.

7. EXTENSIONS AND FUTURE WORK

In this section we discuss important functionality that is not yet implemented in Aranea. In some cases we have very clear idea how to do it, in other cases our understanding is more vague.

7.1 Blocking Calls and Continuations

Consider the following very simple scenario:

1. When user clicks a button, we start a new subflow.
2. When this subflow eventually completes we want to assign its return value to some text field.

In event-driven programming model the following code would be typical:

```
OnClickListener listener = new OnClickListener() {
  void onClick() {
    Handler handler = new Handler() {
      void onFinish(Object result) {
        field.setText((String)result);
      }
    }
    getFlowCtx().
      start(new SubFlow(), handler);
  }
}
button.addOnClickListener(listener);
```

What strikes here is the need to use multiple event listeners, and as a result writing multiple anonymous classes that are clumsy Java equivalent of syntactical closures. What we would like to write is:

```
OnClickListener listener = new OnClickListener() {
  void onClick() {
    String result = (String)getFlowCtx().
              call(new SubFlow());
    label.setText(result);
  }
}
button.addOnClickListener(listener);
```

What happens here is that flow is now called using blocking semantics.

Typically blocking behavior is implemented by suspending executed thread and waiting on some concurrency primitive like semaphore or monitor. The disadvantage of such solution is that operating system threads are expensive, so

using an extra thread for each user session would be a major overkill—most application servers use a limited pool of worker threads that would be exhausted very fast. Besides, threads cannot be serialized and migrated to other cluster nodes. A more conceptual problem is that suspended thread contains information regarding processing of the whole web request, whereas it can be woken up by a different web request. Also, in Java blocking threads would retain ownership of all monitors.

In [25] and [22] *continuations* were proposed to solve the blocking problem in web applications, described above. Continuation can be thought of as a lightweight snapshot of thread's call stack that can be resumed multiple times. There still remains the problem that both thread and continuation contain information regarding processing of the whole request, but can be woken up by a different web request.

To solve this problem *partial continuations* [23] can be used. Essentially, the difference is that the snapshot of call stack is taken not from the root, but starting from some stack frame that we will call *boundary*. In case of Aranea, the boundary will be the stack frame of event handler invocation that may contain blocking statements. So in case of our previous example the boundary will be invocation of method `onClick()`. When we need to wait for an event, the following should be executed:

1. Take current partial continuation,
2. Register it as an event handler,
3. Escape to the boundary.

Similar approach can be also applied to services though mimicking such frameworks as Cocoon [2] and RIFE [14]. We'd like to stress that by applying continuations to widget event handlers we can create a more powerful programming model because there can be simultaneous linear flows at different places of the same widget hierarchy, e.g. in each flow container. This programming model is similar to that of Smalltalk web framework Seaside [21] that uses continuations to provide analogous blocking call semantics of flows, but not event handlers in general.

Java does not have native support for continuations, but luckily there exists experimental library [7] that allows suspending current partial continuation and later resuming it. Aranea currently does not have implementation of this approach, however, it should be relatively easy to do that. Event handlers containing blocking statements should be instrumented with additional logic denoting continuation boundaries. We could use e.g. AspectJ [24] to do that.

Altogether we view blocking calls as a rather easily implementable syntactic sugar above the core framework. At the same time we find that combining event-based and sequential programming in a component framework is a very powerful idea because different parts of application logic can be expressed using the most suitable tool.

7.2 State Management

We have also solutions to the following problems related to state management:

- *Optimizing memory consumption*—in high performance applications low memory consumption of session state representation is essential. The interface of `Component` has methods `disable()` and `enable()` that allow releasing all unnecessary resources when disabled.
- *Client-side state*—part of session state can be kept within web response that later becomes web request or within a cookie [13]. This also allows reducing server-side memory consumption.
- *Navigation history*—supporting (or sensibly ignoring) browser's back, forward, refresh and new window navigation commands. This can be useful both for usability, decreasing server-side state representation or for integrating with other frameworks that rely on browser's navigation commands as the only navigation mechanism.

7.3 Integration and Portals

It is important to note that so far we have described only applications that are configured before deployment and work within one Java virtual machine (or homogeneous cluster). There are portal applications that would benefit from dynamic reconfiguration and using widgets or flows deployed to another environment. The latter could happen for multiple reasons such as using a different platform (like .NET), co-location of web application with database or just administrative reasons.

One possible approach is to integrate with Portlet [12] specification together with remote integration protocol WSRP [9]. Unfortunately portlets cannot be composed into hierarchies and have many limitations on how they can communicate with each other. There is also no notion of nested process in portlets. Finally, portal implementations that we are aware of allow reconfiguring portals only by redeployment.

It should be easy to assemble out of Aranea components a portal application that would contain multiple pre-packaged applications, communicating with each other, but the configuration would have to be read on deployment. One further direction is to integrate Aranea with some component framework allowing dynamic reconfiguration, such as OSGi [11].

Another related direction is to develop a remote integration protocol that would allow creating a widget that would be a proxy to a widget located in another environment. One important issue would be minimizing the number of round-trips.

7.4 Fat Client

Lately, more and more web applications started using asynchronous requests to update parts of the page without resubmitting and refreshing the whole page. Some applications even implement most of UI logic on the client-side and use web server essentially as a container for the business layer. The enabling technology is called Ajax [1] and is essentially a small API that allows sending web requests to the server. We think that this trend will continue and in future most application will use this approach to a varying extent.

The first option is when UI logic is still implemented on the server-side, but in order to make web pages more responsive sometimes ad-hoc asynchronous requests are used to update page structure without refreshing the whole page. This can be accomplished in Aranea using either messages

or the fact that widgets extend services and consequently have `action(input,output)` method. Within widget, some kind of simple event handling logic could be implemented.

Another option is when all UI implemented on the client-side within browser and server-side controller acts essentially as a business layer. Although business layer is often stateless, we find that Aranea could be used to create a coarse-grained server-side representation of UI state, essentially representing activated use-cases, modeled most naturally as flows. Client-side UI would we able to only execute commands making sense in the context of current server-side UI state. Such approach is very convenient for enforcing complex stateful authorization rules and data validation would have to be performed on the server-side in any case.

8. RELATED WORK

As it was mentioned before, Aranea draws its ideas from multiple frameworks such as Struts [3], WebWork [19], JavaServer Faces [8], ASP.NET [4], Wicket [20], Tapestry [5], WASH [26], Cocoon [2], Seaside [21], Spring Web Flow [18], and RIFE [14]. When possible we have referenced the original source of idea at the moment of introducing it.

Although we were not aware of Seaside [21] when developing this framework, we have to acknowledge that rich UI programming interface of widgets and flows is almost identical with programming interface of Seaside, but the design of Seaside differs a lot and it is not intended as a component model for web framework construction and integration.

9. ACKNOWLEDGEMENTS

Development of Aranea implementation has been supported by Webmedia, Ltd. We are grateful to Maksim Boiko, who prototyped the implementation in his bachelor thesis and Konstantin Tretyakov, who provided valuable input as well as help with developing the presentation module. This work was partially supported by Estonian Science Foundation grant No. 6713.

10. SUMMARY

In this paper we have motivated and described a component model for assembling web controller frameworks. We see it as a platform for framework development, integration and research.

There exists an open source implementation of Aranea framework available at `http://araneaframework.org/`. It is bundled together with reusable controls, such as input forms and data lists and advanced JSP-based rendering engine. This framework has been used in real projects and we find it ready for production use. Interested reader can also find at this address an extended version of this article with many details that had to be omitted here.

11. REFERENCES

[1] Ajax. Wikipedia encyclopedia article available at `http://en.wikipedia.org/wiki/AJAX`.

[2] Apache Cocoon project. Available at `http://cocoon.apache.org/`.

[3] Apache Struts project. Available at `http://struts.apache.org/`.

[4] ASP.NET. Available at `http://asp.net/`.

[5] Jakarta Tapestry. Available at `http://jakarta.apache.org/tapestry/`.

[6] Java Servlet 2.4 Specification (JSR-000154). Available at `http://www.jcp.org/aboutJava/communityprocess/final/jsr154/index.html`.

[7] The Javaflow component, Jakarta Commons project. Available at `http://jakarta.apache.org/commons/sandbox/javaflow/index.html`.

[8] JavaServer Faces technology. Available at `http://java.sun.com/javaee/javaserverfaces/`.

[9] OASIS Web Services for Remote Portlets. Available at `www.oasis-open.org/committees/wsrp/`.

[10] Open source web frameworks in Java. Available at `http://java-source.net/open-source/web-frameworks`.

[11] OSGi Service Platform. Available at `http://www.osgi.org/`.

[12] Portlet Specification (JSR-000168). Available at `http://www.jcp.org/aboutJava/communityprocess/final/jsr168/`.

[13] RFC 2109 - HTTP State Management Mechanism. Available at `http://www.faqs.org/rfcs/rfc2109.html`.

[14] RIFE. Available at `http://rifers.org/`.

[15] RIFE/Crud. Available at `http://rifers.org/wiki/display/rifecrud/`.

[16] Ruby on Rails. Available at `http://www.rubyonrails.org/`.

[17] Spring. Available at `http://springframework.org`.

[18] Spring Web Flow. Available at `http://opensource.atlassian.com/confluence/spring/display/WEBFLOW/`.

[19] WebWork, OpenSymphony project. Available at `http://struts.apache.org/`.

[20] Wicket. Available at `http://wicket.sourceforge.net/`.

[21] S. Ducasse, A. Lienhard and L. Renggli. Seaside — a multiple control flow web application framework. *ESUG 2004 Research Track*, pages 231–257, September 2004.

[22] P. T. Graunke, S. Krishnamurthi, V. der Hoeven and M. Felleisen. Programming the web with high-level programming languages. In *European Symposium on Programming (ESOP 2001)*, 2001.

[23] R. Hieb, K. Dybvig and C. W. Anderson, III. Subcontinuations. *Lisp and Symbolic Computation*, 7(1):83–110, 1994.

[24] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001. Project web site: `http://www.eclipse.org/aspectj/`.

[25] C. Queinnec. The influence of browsers on evaluators or, continuations to program web servers. *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 23–33, 2000.

[26] P. Thiemann. An embedded domain-specific language for type-safe server-side web-scripting. Available at `http://www.informatik.uni-freiburg.de/~thiemann/haskell/WASH/`.