Supporting Reconfigurable Object Distribution for Customized Web Applications

Po-Hao Chang
RiverGlass, Inc.
60 Hazelwood Drive, Suite 216
Champaign, Illinois 61820
pchang@riverglassinc.com

Gul Agha
University of Illinois at Urbana-Champaign
201 North Goodwin Avenue
Urbana, Illinois 61801
agha@cs.uiuc.edu

ABSTRACT

In current practice, Web applications are tightly coupled with the platforms that a particular service provider intends to support and the execution scenario envisioned at the design time. The resulting applications do not adapt well to all clients and runtime execution contexts. The goal of our research is to develop methods and software to support *reconfigurable distributed applications* which can be customized to specific requirements. We view a Web application as a composition of *actors*, i.e. distributed active objects, and apply techniques of *generative programming* to develop a virtual application framework which separates the logic of objects from aspects relevant to object distribution on different platforms. We describe `ActorSpec`, a specification system allowing programmers to express desired object distribution and assisting application generators to produce highly customized versions of an application. The resulting flexibility facilitates the development of customizable Web applications on an increasingly complex Web infrastructure.

1. INTRODUCTION

Although the Web started as a network content delivery system, it has increasingly transformed itself into a gigantic service backbone with the Web browser serving as a consolidated gateway. For example, people use browsers to check emails, shop online, trade equities, manage accounts and even remotely control home appliances. The popularity of Web applications is the result of the convenience and ease enjoyed by end users. In theory, users employing any Web browser and platform to connect to the Internet can access all kinds of services around the world. In reality, the increasing heterogeneity and complexity of the Web is jeopardizing this ideal paradigm. In particular, diverse Web devices and protocols result in heterogeneity, and varying execution contexts create complexity which makes universal service access difficult.

In order to maintain universal accessibility, industry and practitioners had generally paid more attention to Web server technologies and ignored many rich features that are available in new browsers. However, because of the inherent latency in communication channels, Web servers are not good in handling certain user actions that require immediate responses – for example, actions such as *mouse move* and *drag-drop*. To address this problem, browser-centered Web applications using *Ajax* (Asynchronous JavaScript and XML) [11] have been proposed and won considerable acclaim for their responsiveness to users. However, browser-centered Web applications come with a price: not all browsers can access these applications. Thus the Web developer faces a dilemma: to provide universal access with a loss of efficiency, or vice-versa.

One could argue that it may not be a great sacrifice to support only advanced browsers, which have dominated the market. However, there is a second related difficulty: Web applications are inherently distributed, and these browsers allow great flexibility in application deployment across the client and the server–specifically, an application is composed of many components and the problem of *where* and *when* to distribute components becomes crucial; specifically, it is a question about location and timing.

Location: Validation on user input is a common task in many Web applications. On the one hand, because processing locally results in faster response time and reduces the server load, validation should be performed at the client if possible. On the other hand, if a service provider wants to conceal the validation code, or the input has to be validated against confidential data, the validation component should be deployed at the server.

Timing: Many Web applications have multiple levels of presentation such as layered menus. A common technique to implement multiple levels of presentation is embedding all components–whether such components are immediately visible or not–in a single page. The advantage of this structure is that once a page loaded, the user can immediately see the effect of triggering an event. On the contrary, representing all layers in a single page may not be efficient in an execution context where the available bandwidth is scarce.

The above discussion illustrates how to determine the preferred distribution is strongly dependent on an execution context; relevant parameters of such a context include network properties, security constraints, server load and even user behavior, hardly foreseen in advance. Following the principle of *separation of concerns* [6], *aspects* orthogonal to the application logic should be decoupled from logical components. In a traditional distributed environment, where a certain uniformity is taken for granted, the decisions on

SAC'07 March 11-15, 2007, Seoul, Korea

these aspects can be postponed until deployment time with little adverse implication. In contrast, Web application developers have to make right decisions early in the design phase: reverting them later could cost dearly because the diversity of programming paradigms on different platforms. Thus, the Web developer faces the second dilemma: how to find a good distribution scheme in various execution contexts.

We address both problems–the need for universal accessibility and context-dependent distribution–by using *virtualization* on execution platforms to provide a uniform programming environment for objects. Specifically, we have developed a virtual application framework (see Fig. 1) which hides the incompatibilities in different platforms and models. Our framework supports reconfigurable object distribution through a specification interface. In particular, a customized version of an application can be derived through a specification of rules for object distribution.

One approach to support virtualization is to define a middleware. This approach has the disadvantage of requiring the deployment of new infrastructure on the platforms. Instead, we adopt the paradigm of generative programming [5] to design the framework: before deployment, the objects that compose a Web application undergo a generative process to fit the specification of object distribution and the target platforms. Using the terminology of Aspect-Oriented Programming (AOP) [2,7], aspects relevant to object distribution are woven into the code. The generative approach offers adaptability to address universal accessibility: developers can take advantage of the latest technologies as soon as the supporting generators and framework modules become available. At the same time, the approach provides flexibility needed to dynamically optimize the distribution of objects depending on the given configuration and requirements.

The rest of this paper is structured as follows. Section 2 presents a comprehensive overview of the virtual framework. Section 3 describes the way to specify objects and Section 4 explains the mechanism to translate specifications into object distribution. Section 5 reviews some related work, and the final section concludes the paper with a discussion.

2. VIRTUAL FRAMEWORK

We view a Web application as a composition of distributed active objects, or *actors*. Being distributed implies knowledge is encapsulated within objects; active means the actors are able to decouple method execution from synchronous method invocation [15]. We believe the distributed active object model is especially suitable for location agnostic application development in a dynamic distributed environment such as the Web.

2.1 Actor Model

We assume that distributed objects do not implicitly share information (e.g. through classes or shared variables) but interact with each other only through message passing. Such lack of sharing lets objects be placed freely. Moreover, we abstract away the details of the communication protocols. This leads us to the defining characteristics of the Actor model [1]. In the Actor model, each *actor* encapsulates a state and a set of methods that manipulate the state with a thread of control. In the basic Actor model, actors communicate with others using asynchronous messages; other message-passing paradigms, such as synchronous communication, must be built in terms of asynchronous messages. For our purposes, we simply represent synchronous messages directly. Note that object property set and get operations, and method invocations in various object systems, can be unified into synchronous message exchanges, while event models can be supported by asynchronous message exchanges.

We have designed `ActorScript`, an actor scripting language, to facilitate the development of Web applications. It borrows the control structure and most of its data types from JavaScript [12] so that the generating process can be simplified for many Web platforms. The major distinctive features are described as follows:

An actor is self-contained: Passing arguments in method invocation, throwing out exceptions and returning values mean exporting part of its internal state, which requires (deep) copying. In JavaScript, *functions* are also data but they cannot be exported. Instead, we design a new data type *port* to represent a method of a specific actor. A *port* can be exported and be called as a method.

An actor can call a method asynchronously: The caller continues its execution without waiting the call's return. If the caller and callee are on different platforms, or on a multithreaded platform, an asynchronous method invocation may result in concurrent execution. Events can be viewed as asynchronous method invocations.

An actor is created by a prototype: The creation prototype defines an actor's initial layout and behavior. We avoid the term *class* because in many object systems, a class is not only a "concept of design" but also an "implementation of design." An implementation is a concrete entity at runtime and hence *class variables* or *static variables* become common knowledge shared between objects. Moreover, *class* implies *type*. In our framework actors are *typeless* and they can change behavior at runtime.

2.2 Implementation

A framework is an environment for actors: it enables actor creation, delivers messages and empowers actors to respond to messages. We have designed a virtual framework for Web applications. It is virtual in two senses: first, the framework provides a *virtualization* over a heterogeneous distributed platforms; and second, instead of a native environment for actors written in `ActorScript`, the framework is a set of libraries, runtime systems (such as a garbage collector), and generators which accommodate actors to host platforms.

An actor framework includes one or more *Podiums*, which are abstractions of physical execution platforms. A podium is built on top of a host platform, and its function is that of a virtual machine. In the context of Web applications, we need two kinds of podiums:

A **client podium** provides an execution environment in the browser. We implement the client podium with a set of JavaScript libraries which are downloaded when an application launched. Because of the single-threaded nature of the browser, the basic structure of a client podium is a message loop. To deliver an asynchronous message to a local actor the podium places a request in a message queue; when the podium is idle, it picks a pending message from the queue and dispatches it.

A **server podium** is a more powerful virtual machine which can service multiple asynchronous messages concur-

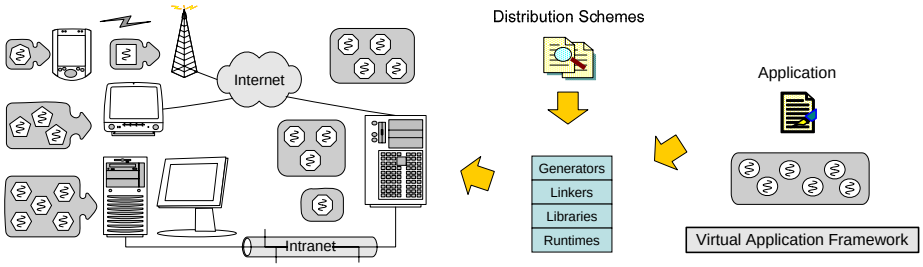

Figure 1: Virtual framework enables location and platform agnostic application development.

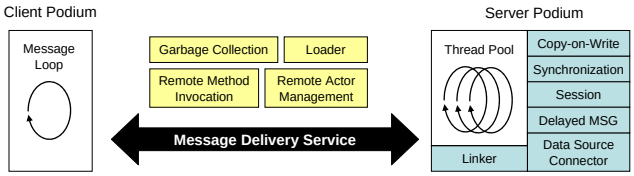

Figure 2: Podiums and framework services.

rently; therefore support for synchronization is required. Session control is also needed to maintain application-wide resources and to keep track of stateless HTTP requests. In the context of Web applications, the server has another role in application distribution: supplying actors, prototypes and static resources to the client podium. The *linker* and *loader* are designed for this purpose. The framework exposes a specification interface to control these modules for reconfiguration of actor distribution. We have implemented a server podium using *Rhino* [8] embedded in an *Apache Tomcat* [9].

The architecture of the virtual framework is depicted in Figure 2. *Message Delivery Service* supports synchronous and asynchronous message delivery between podiums. *Remote Actor Management* provides the transparency of remote actors: when an actor's reference is exported, it creates a proxy actor acting on behalf of the actual one in the destination podium. *Remote Method Invocation* implements message serialization and deserialization; it also handles remote exception propagation.

Using different podium implementations, we can adapt an application to a variety of execution platforms. But to customize an application for a specific execution context, we need to control the distribution of actors. A specification system is designed to couple with the framework for this purpose. The next section describes such a specification system.

3. SPECIFICATION RULES

Two important principles guide the design of `ActorSpec`, our specification system. First, we want a clean separation between an application and its specification requirements so that each can be reused independent of the other. Although programmers may be able to produce more efficient applications using a particular specification system, such hardwiring of concerns would compromise portability and flexibility of the code. Second, because we do not make assumptions about the runtime environment in which the application will be deployed, `ActorSpec` does not rely on runtime information–rather, a specification only relates actors to each other.

3.1 Actor Annotations

Many non-functional concerns, including those we are particularly interested in, can be specified by annotating actors. For example, in order to specify the host podium of an actor, an attribute **Podium** can be defined, and we can allocate an actor to a desired podium by annotating it with a *Podium Value*. Because we do not work with runtime information, the value of an attribute is invariant over the entire lifespan of an actor and it is safe to set an actor's attributes at the time of its creation. However, it is not generally possible to annotate a specific actor without the actor's unique runtime identification. Instead, we apply a rule to the set of all actors that are created by a prototype. This turns out to be reasonable for the applications we have looked at. The syntax for specifying that all actors created with prototype X have the same *value* of **attribute A** is as follows:

```
[prototype X]:[attribute A] = value;
```

For example, to allocate all `DateValidators` to the client:

```
DateValidator:Podium = Client;
```

Rules in this form are suitable for large or unique components, but not for small ones which are used for different purposes. For example, `Button` is a common component: we expect different buttons have different attribute values in different contexts.

3.2 Selection by Actor Genealogy

We want to select an actor not only by its prototype but also by its context. The immediate candidate for an actor's context is its creator. We may select a set of actors not only by their prototype (as above), but also by their creators' prototypes. This motivates the second rule for our specification scheme. In order to specify that all actors of Y created by an actor of X share the same *value* of **attribute A**, we write:

```
[prototype X]>[prototype Y]:[attribute A] = value;
```

For example, we can allocate different `Button` actors to different podiums, based on their creators' prototypes:

```
OrderForm>Button:Podium = Client;
InventoryForm>Button:Podium = Client;
```

Note that the idea of this specification is based on the reuse pattern of "has-a" relationship [10] which says that

an object is composed with other objects, and the former (called a *container* object) is a natural context for the contained objects. However, the containment relation is not always unique and static: an actor can be contained in more than one actor and its ownership may be transferred at runtime. On the other hand, the creator of an actor is unique and non-transferable. Our observations suggest that, in practice, the primary owner of an object is usually its creator. This justifies the use of creatorship to approximate ownership for the purposes of our specifications.

A generalization of the creator relation is to specify an actor by its *genealogy*–extending creatorship to more generations. Note that the genealogy can be determined at creation time and remains invariant for an actor. For example, the following rule says the attribute value of an actor of prototype X_i is decided by examining its genealogy for up to i generations.

```
[X0]>...>[Xi]:[attribute A] = value;
```

Obviously, examining the rules for more than one generation can lead to conflicting rules for the same attributes. We use the principle that a more specific rule overrides a less specific one. Because the genealogy ordering is linear, this resolves the conflict in all cases.

3.3 Implementation of Prototype

Annotating a set of actors of a prototype with specific attributes implies that the actors use a specialized implementation of that prototype which produces actors obeying the given specification. Note in our model, a prototype is a "concept of design" instead of an "implementation of design." From another prospective, a specification rule annotates a prototype implementation. For example, we may define an attribute controlling the loading policy of prototype **PrototypeLoad** with two possible values *PreLoad* and *OnDemand* as follows:

```
SubMenu:PrototypeLoad = OnDemand;
PricePanel>GridControl:PrototypeLoad = PreLoad;
CalcPanel>GridControl:PrototypeLoad = OnDemand;
```

4. RECONFIGURING DISTRIBUTION

The process of customizing an application according to an actor distribution specification requires two elements. First, the application framework has to expose some parameters–*attributes* in the specification system–which support reconfigurable actor distributions. Second, we need a mechanism which enforces specification rules for customized applications.

4.1 Attributes for Distribution

To control actor distribution, we consider the following questions: First, which podium does the actor reside in? Second, which podium is the actor created in? Note that an actor may be created in one podium and be deployed in another. Third, when does the actor move? For example, an actor may migrate immediately after its creation, or it may migrate on demand. Fourth, when is the prototype implementation loaded? An implementation may be loaded with another prototype implementation to which it is linked, or it may be loaded on demand?

While the second and third question suggest the possibility of *actor migration*, currently the implementation does not support actor migration. This is because such migration requires complex runtime support and is not feasible for 'lightweight' actors. Instead, our framework supports a more limited form of mobility: an actor may be created in one podium where it may be more efficient to access the resources needed for its creation, and after creation, the actor is sent to a different target podium where the actor will likely have more local interactions. For example, although a personalized information panel should be deployed in a client, a server with direct access to the relevant database is a better place to create it.

Because of the asymmetry of Web platforms, we only consider one-way mobility: a client actor can be created in a server and migrate, but not vice-versa. Observe that the cost of loading into a client podium is much greater than that into a server podium. We have two different timings for the loading behavior of a mobile actor to a client podium: an actor may be deployed to the client podium as soon as a client actor has its reference, or the actor may wait in the server podium until it is needed at the client podium. The two different timing strategies–eager deployment and lazy deployment–are defined in an obvious way. However, our implementation ignores the much less significant overhead of loading prototypes into a server: all server prototypes are loaded when the application launches.

As the above discussion suggests, an application framework exposes three attributes on actor distribution:

```
1. Podium = {Client | Server | Mobile}
2. ActorLoad = {PreLoad | OnDemand}
3. PrototypeLoad = {PreLoad | OnDemand}
```

Using `ActorSpec` with these attributes, the developer can customize an application for different execution contexts with different actor distribution schemes.

4.2 Resolving Specification Rules

Specification rules can be used to express actor distribution schemes for different execution scenarios, for example, for less secure or less capable browsers, most actors are allocated on the server; for a connection network with limited bandwidth, uncommonly used actors are tagged with *OnDemand*.

We designed methods to resolve specification rules into useful information which guides the generator to produce customized applications following actor distribution schemes. Statically an application is a composition of actor prototypes; each prototype may depend on other prototypes and itself (one prototype can have multiple versions so we need to keep self-dependency information). Because it is not necessary to know the interface to invoke a method of an actor (actors are *typeless* and method invocation is realized through message exchange), that prototype A depends on prototype B just means an actor of A may create an actor of B. We can define *Prototype Dependency Graph* as follows:

DEFINITION 1. ***Prototype Dependency Graph*** *is a directed graph where a vertex represents a prototype and an arc denotes the dependency of two prototypes.*

The prototype dependency graph can be derived by inspecting the composition structure of the application. Note it may contain cycles and self-loops. Obviously it is unique for an application. In contrast, an executable application

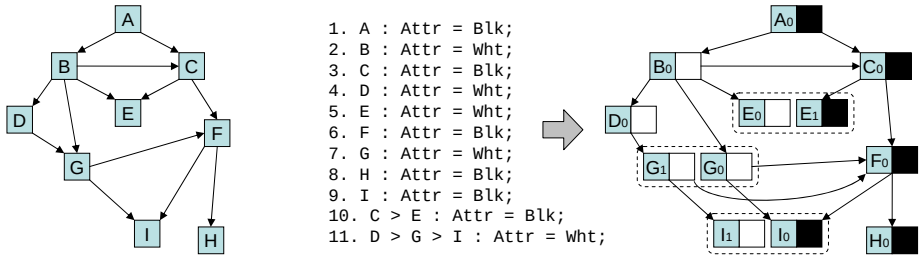

Figure 3: Combining the Prototype Dependency Graph (left) with Specification Rules (center), an Implementation Linkage Graph (right) can be derived.

image is a composition of prototype implementations. Similarly we define *Implementation Linkage Graph* to capture the linkage structure of prototype implementations.

DEFINITION 2. ***Implementation Linkage Graph*** *is a directed graph where a vertex represents a prototype implementation, and an arc denotes the linkage relation of two prototype implementation. A vertex also includes one or more attributes which characterize the implementation.*

An implementation linkage graph contains information about a specific version of an application. One application may have several implementation linkage graphs with different requirements. In our case the requirement is written in specification rules. Customization is achieved through transforming the composition of concepts of design into a composition of implementations of design. Figure 3 demonstrates the transformation. From the graph we can see prototype E, G and I have two different implementations.

We developed an algorithm for the transformation: given a prototype dependency graph with a set of specification rules, generate an implementation linkage graph consistent to these rules. We also proved our algorithm is correct. For the limited number of pages, we use an informal way to describe the algorithm and sketch the proofs. The formal algorithm and detailed proofs are available upon request. First we make the following assumptions to simplify the description: 1. There is only one attribute **Attr** in specification rules: multiple attributes can be combined into a new one. 2. For each prototype X, we have a rule `X:Attr = Value;`, i.e. there is a default implementation of each prototype. Since each prototype must have at least an implementation, the assumption is valid. The most compact form of rules can be easily computed. 3. The prototype dependency graph contains no cycles and self-loops. It is not always valid and our algorithm does work in existence of them. The assumption is just for convenience of description.

We use Figure 3 as an example: the process is shown in Figure 4. **Rule** $1 \sim 9$ are default implementations, so we build the initial implementation linkage graph by marking each vertex of the prototype dependency graph with *Implementation 0* and the specified value. With **Rule** 10, the specified value of E from C conflicts with E_0, so we create a new implementation vertex of E, i.e. E_1, with correct value. Then we remove the arc from C_0 to E_0 and add an arc from C_0 to E_1 because the rule specifies the path from C to E. Adding **Rule** 11 is more sophisticated: in addition to I, G needs a new implementation G_1 although the rule does not specify G (the new vertex has the same value as the original one). Without G_1, G_0 would have an undeterministic choice of I_0 and I_1, and the information $D : G : I$ of the rule would be lost.

From the example, we generalize Algorithm 1:

Algorithm 1 Generate an implementation linkage graph

1: create the initial graph with all single prototype rules $R = (p, value)$
2: **for all** genealogy rules R in an ascending order of $length(R)$ **do**
3: add $R = ((p_0, p_1, \ldots, p_n), value)$
4: **end for**

We separate rules into those with a prototype $(p, value)$ and those with a genealogy $((p_0, p_1, \ldots, p_n), value)$. We define the length of a rule as the number of prototypes in the rule. Adding the rule in an ascending order of $length(R)$ means the more specific rules will be added later and override the less ones. To add a genealogy rule R, we have a subroutine shown in Algorithm 2. Note in line 8, when the arc (v_0, v_1) is removed, it is possible that no implementation vertices have an arc to v_1. It means the rule is overspecified but does not result in an error. The implementation v_1 is no longer reachable from the root and not part of the final graph.

Algorithm 2 Add a genealogy rule

1: **for all** path $(v_0, v_1, \ldots, v_n)$ such that all v_i is an implementation of p_i **do**
2: **for** $i = 1$ to n **do**
3: generate a new implementation v'_i of p_i
4: copy the value and arcs of v_i to v'_i except the arc (v_i, v_{i+1})
5: add the arc (v'_{i-1}, v'_i)
6: **end for**
7: set the value of v'_n to $value$
8: remove the arc (v_0, v_1)
9: **end for**

To prove the correctness of the algorithm, we need to show that the result implementation linkage graph G' is a valid implementation of the prototype dependency graph G, and it also correctly follows the specification rules.

For the first claim, we have to prove for each path $p = (v_0, v_1, \ldots, v_n)$ in G starting from the root (the application), there exists a unique path $p' = (v'_0, v'_1, \ldots, v'_n)$ in G' where for all $i \in [0, n]$, v'_i is an implementation of v_i. It means if

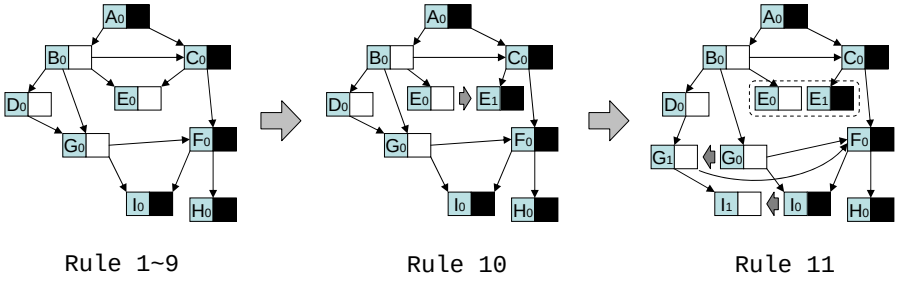

Figure 4: The algorithm adds rules to the implementation linkage graph.

an actor can be created under a genealogy in the design of G, it can be created under the same genealogy with correct implementations in G'. It can be proved by M.I. on the number of rules added: the initial case is trivial since the two graphs are identical in topology, and it can be shown that the claim holds after a new rule added. The reverse direction is also true but it does not matter: if there were no p for a p', the p' would have never happened in the runtime trace.

For the second claim, we have to show that for each path p', the *value* of p'_n is the value specified by the most specific rule applicable to p. It can be shown that the *value* of p'_n follows the latest added rule applicable to p. Since our algorithm adds rules from less specific to more, it follows.

4.3 Generating Customized Applications

With the *Implementation Linkage Graph*, it is straightforward to generate a customized application image. First the generator constructs prototype implementations if they are not prebuilt. For a prototype written purely in `ActorScript`, there are three possible implementations: *Server Implementation* works in the server podium and creates actors there; it is generated in the server language with interfaces to the server podium. *Mobile Implementation* has two parts: the server part creates actors and the client part includes the methods of these actors. The newly created actor will be tagged with its **ActorLoad** attribute. *Client Implementation* works in the client podium and creates actor there. The **PrototypeLoad** attribute affects the linkage not the logic part of the implementation.

The next step is providing correct linkage for each prototype implementation. For example in Figure 3, the implementation G_1 has the references to F_0 and I_1. Conceptually the statements in G_0

```
var actor = new F();   var actor = new I();
```

will be replaced with

```
var actor = new F0();  var actor = new I1();
```

Recall that the server podium loads all server prototypes (including the server part of mobile prototypes) as soon as the application starts: linkage to these prototypes are also resolved at that time. When a server prototype is used at the first time, the server podium will try to resolve the unresolved client linkage. Now the **PrototypeLoad** attribute plays its role: if the value is *PreLoad*, the client podium will load it now; if the value is *OnDemand*, the client podium will load a stub of the actual prototype. Either way, the server prototype resolves its linkage.

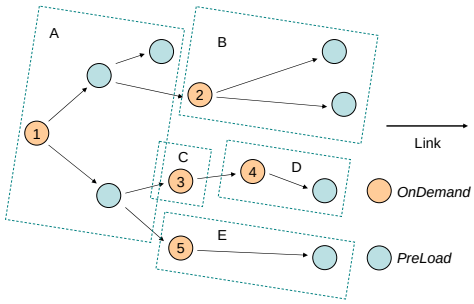

Figure 5: PrototypeLoad organizes client prototypes into loading batches.

We have different linking policy for client prototypes. A client prototype will try to resolve its linkage to other prototypes when the loader is ready to load it. As a result, loading a client prototype may trigger a chain of prototype loads through the linkage. In Figure 5 we can see the attribute **PrototypeLoad** organizes client prototypes into several loading batches: when a prototype needs to resolve a client prototype which is the root of a loading batch, all prototypes in the batch will be loaded.

The **ActorLoad** attribute on mobile actors works in a similar way. When a reference to a *PreLoad* mobile actor is going to be sent to the client podium, the loader tries to inspect its references to other mobile actors: those tagged with *PreLoad* will be loaded with it together while the references to the rest will be filled with stubs. These actors loaded together also form a loading batch as client prototypes.

5. RELATED WORK

Metaprograms are programs that manipulate other programs (or themselves) as their data and writing such programs is called metaprogramming [4]. Our specification system uses specification schemes (metaprograms) to modify applications–it is a metaprogramming system. There are many applications of metaprogramming. The *Meta Object Protocol* (MOP) [14] and Aspect-Oriented Programming (AOP) [2, 7] are two of them with a common goal to provide a mechanism to separate functional and non-functional concerns.

The basic idea of MOP is using *meta objects* to control *base objects*: an MOP is a protocol about expressing and executing meta objects. To put our research into the context, the loader and linker of the virtual framework are meta objects which govern actors' placement and move-

ment: the attributes associated with each actor are part of the metaprogram of a distribution scheme. Several two-level systems [3, 17] had been built and demonstrated the feasibility to use meta objects to achieve separation of non-functional concerns such as backup policies, multimedia QoS and synchronization from applications. Although there is no strict definition, many MOP-enabled systems support runtime meta objects. Runtime meta objects provides more flexibility and extensibility at the cost of overhead in runtime system support.

AOP takes a more static approach: instead of having meta objects working at runtime, an AOP tool weaves non-functional aspects back to functional components. The generators in our application framework have a similar role as an AOP tool: it consumes annotations to generate customized prototype implementations. Supposedly the approach can produce more efficient applications but the *join point models* which dictate the places where aspects and applications meet have more limitations. Some researchers applied these principles on Web systems to separate concerns of component distributions [13, 16].

AOP and MOPs focus more on a modular framework to structure concerns. This allows the programmer to customize an application to a very high degree but requires careful analysis in early phase to avoid an intrusive design. On the other hand, our design leaves the burden to the runtime environment designer, who decides the attributes to be exported and implements a mechanism to interpret the specifications, through runtime systems and code generators. Instead of implementing customized aspects or doing metaprogramming, the programmer annotates the application. Our approach is less powerful because a pre-designed framework only allows limited customization. We argue that the trade-off can be justified: it keeps Web development simple and the ability to reconfigure object distribution is sufficient for this problem domain. A different runtime environment can be designed for a different domain when needed. In addition, most of these tools and systems assume a homogeneous system, which is not valid on the Web as we indicated.

6. DISCUSSION AND CONCLUSION

We described a new approach to build customizable Web applications through a virtual framework and a specification system. The virtual framework hides the diversity of platforms on the Web and the specification system facilitates building reconfigurable component distribution to adapt to varying execution contexts.

Our framework design is extensible because of the nature of actor communication via message exchanges. It is easy to develop other server podiums to work with our client podium and vice versa. A future direction of our work is to extend the framework to include more podiums in different roles. For example, some Internet intermediary platforms allow partial processing on cached data or work as surrogates for very simple clients such as cellphones and PDAs. The extension can promote the *two-tier* architecture of our framework to *n-tier* and provide availability to a wider range of applications.

`ActorSpec` is flexible: The rules and methods work for specifications based on object annotation, including concerns unrelated to object distribution. For example to customize security settings, a security attribute can be defined. Our system can identify those objects requiring special treatment on security, i.e., special implementations. However, a limitation of `ActorSpec` is that it resolves rules statically and does not work on dynamic features. Hence this makes it possible to statically check invalid or conflicting specification rules, but it is non-trivial to find *bad* rules, since 'badness' is usually related to runtime behavior. For example, a bad distribution scheme such as interleaving server and client actors along an execution path results in multiple hops in the Internet.

7. REFERENCES

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] Aspect-Oriented Software Association. http://www.aosd.net/.

[3] Mark Astley, Daniel Sturman, and Gul Agha. Customizable middleware for modular distributed software. *Communications of ACM*, 44(5):99–107, 2001.

[4] Jonathan Bartlett. The art of metaprogramming part 1-3, 2005-2006. IBM DeveloperWorks.

[5] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[6] Edsger W. Dijkstra. *A Principle of Programming*. Prentice Hall, 1997.

[7] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming. *Communications of ACM*, 44(10), 2001.

[8] Mozilla Foundation. Rhino: Javascript for Java. http://www.mozilla.org/rhino/.

[9] The Apache Software Foundation. Apache tomcat. http://tomcat.apache.org/.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[11] Jesse James Garrett. Ajax: A New Approach to Web Applications, February 2005.

[12] ECMA International. *ECMAScript Language Specification*, December 1999.

[13] Mik Kersten and Gail C. Murphy. Atlas: a case study in building a Web-based learning environment using aspect-oriented programming. *ACM SIGPLAN Notices*, 34(10):340–352, 1999.

[14] Gregor Kiczales, Jim Des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[15] R. Greg Lavender and Douglas C. Schmidt. Active object: an object behavioral pattern for concurrent programming. *Proc.Pattern Languages of Programs,*, 1995.

[16] Eli Tilevich, Stephan Urbanski, Yannis Smaragdakis, and Marc Fleury. Aspectizing server-side distribution. In *Proceedings of the Automated Software Engineering (ASE) Conference*. IEEE Press, October 2003.

[17] Nalini Venkatasubramanian, Gul Agha, and Carolyn Talcott. A metaobject framework for qos-based distributed resource management. In *Third International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE '99), December*, 1999.