A Document-based Framework for Internet Application Control

Todd D. Hodes and Randy H. Katz
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720
{*hodes,randy*}@*cs.berkeley.edu*

Abstract

This paper motivates and details a document-based framework for manipulating the components that comprise distributed Internet applications. In the framework, XML documents are used to describe both server-side functionality and the mapping between a client's applications and the servers it accesses. Our system model contrasts with explicitly context-aware application designs, where location information must be explicitly manipulated by the application to affect change; instead, a middleware layer is interposed between client applications and services so that invocations between the two can be transparently remapped. This approach is useful for a subset of application domains, including our example domain of "remote control" of local resources (e.g., lights, stereo components, etc.). We illustrate how the framework allows for 1) remapping of a portion of an existing user interface to a new service, 2) viewing of arbitrary subsets and combinations of the available functionality, and 3) mixing dynamically-generated user interfaces with existing user interfaces.

The use of a document-based framework in addition to a conventional object-oriented programming language provides a number of key features. One of the most useful is that it exposes the mappings between programs/UIs and the objects to which they refer, thereby providing a standard location for manipulation of this indirection.

1 Introduction

Many university and industry groups have projects investigating compositional frameworks for large-scale distributed systems; examples include CalTech's InfoSpheres [11], MIT's Oxygen [3], UCB's Ninja [18], IBM Almaden's TSpaces [31], HP's e-Speak [8], Sun's Jini/EJB atop Java [28, 26], and OMG's CORBA [19]. The basic idea is to enable groups of remote objects on independent Internet hosts to be used together ("federated") to perform tasks via the provision of edifices such as component discovery (e.g., the Jini Lookup service or Ninja SDS [2]) and remote invocation.

Building large-scale software from distributed components, or "services,"[1] is a relatively new area of study, one that has challenges that are inherently different from those in monolithic or client/server program design.

1.1 The Challenge of Orthogonalizing Component Management and Component Usage

Conventional component-based software specifies component locations locally, internal to the application. Keeping such information at the level of the application complicates the process of adapting to components that fail or change due to mobility: the application must either deal with this itself, or a separate component must understand the application-specific configuration files/APIs. To reduce the burden on applications designers and enable generic service management middleware, the component management (locating/spawning/etc.) functionality and component usage functionality can be either partially or completely orthogonalized. This allows the designers to focus on exposing the aggregate functionality to the user, leaving the details of manipulating applications' component references to the middleware. Benefits of such partial or complete orthogonalization of layers is described in [7]. But, the remaining question is, how might we implement such layering in this domain?

[1] We call our framework components "services" to contrast with the more generic term "objects." A service is any entity that can be invoked over the Internet using a known messaging format. Thus, a web server is a "service," as might be a VCR that is connected to the network and advertising its control interface, as in [9].

1.2 The Challenge of Heterogeneous Interfaces

In the context of distributed evolution of components (i.e., by groups scattered across the Internet), there is the potential for independent class hierarchies to be built such that semantically identical objects may either not type-match due to a difference in the interface name, or not type-match because they have differing interfaces. For example, a three-state “on/off/dim” light switch may not type match with a continuous dimming light switch, or a LightSwitchInterface may not match a PowerSwitch-Interface. This leads to a situation where the aggregate system is not composed of objects with consistent interfaces and potentially different implementations; instead, it is composed of both heterogeneous interfaces and heterogeneous implementations. The former case is cleanly handled by any of the aforementioned distributed object systems, while the latter is not.

One approach to addressing these problems is to allow application programs to be downloaded on-the-fly to hand-held devices and uploaded to local computers [9]; for example, as Java applets. The difficulty of this approach, though, is that it does not allow the end-user to customize applications for interaction with a heterogeneous set of services as related entities. In other words, it cannot overcome minor differences in protocol — even for functionally identical services — because the applications are opaque. For example, in the Jini [28] model, a discovered service exposes its interface by passing a Java applet to the client. The applet is allowed to – even encouraged – to use an application-specific protocol (atop RMI) between itself and the host server. If light switch controls were designed using such a model, a user would need to download an applet each time he or she encountered a light switch from a different manufacturer or with different options. The end result of this is that, though the functionality is exposed, it is not in a form amenable to manipulation: the client/server protocol may be opaque.

Given an inability to standardize all functional interfaces and the need to avoid using only opaque mobile code, is there an intermediate solution that balances the need to expose interfaces with the need to agree on protocol standards?

1.3 A Solution Framework: Externalize Component State in Documents

This paper proposes that there is a commonality in the two challenges, and that a single framework can support solutions to each. This framework is a middleware layer that sits above any existing distributed object layer [13]. The new layer extends the traditional distributed component programming approach by forcing applications to expose a portion of their system state as documents.

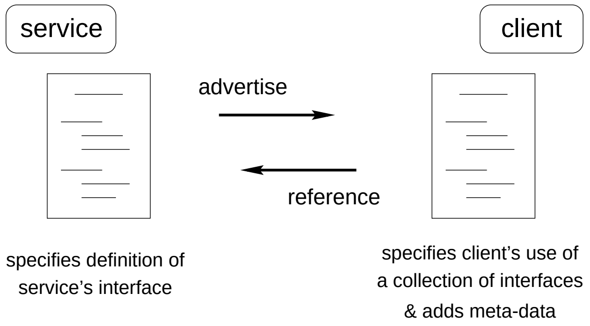


Figure 1: The Document-based Model: services are described by static documents that advertise the definition of their interface; clients maintain documents that indicate how the a collection of interfaces are used.

Specifically, a programs’ remote object usage is separated from its internals, in a manner exposing these locations rather than hiding them. This enables composition by allowing this state to be manipulated by editing the component description documents — a function that can be affected by a third-party independent of the original application. We call this novel use of standardized externalization a “document-based” approach. In the framework, application programs and user interface programs are associated with documents that provide either description of the available services or flexible association of user interfaces to these services.

This document-based distributed object management framework is illustrated in Figure 1.

The document-based approach doesn’t add any functionality that couldn’t otherwise be built into an application, but instead forces there to be a standard layer of indirection between certain references. This indirection is useful in separating the concerns of the applications and the middleware. It exposes the mapping between programs/UIs and the objects to which they refer, thereby providing a standard location for users (and their programs) to manipulate these mappings. Such a system model contrasts with explicitly context-aware application designs [24], where location information is either centralized or must be explicitly manipulated by the application to affect change. Instead, our approach is to interpose a middleware layer between client applications and services so that invocations between the two can be transparently remapped. This approach is useful for a subset of application domains, including that of "remote control" of local resources (e.g., lights, VCRs, stereo components, etc.), an example we will treat in detail in Section 5.

This disassociation of programs/UIs from the objects they reference is similar to the Model/View/Controller (M/V/C) architecture from Smalltalk [12]. In the M/V/C architecture, data (the model) is separated from the presentation of the data (the view) and events that manipulate the data (the controller). Similarly, documents in our system act as the glue that associates data to user interfaces/programs that manipulate and view that data.

How might this framework address the two challenges presented Sections 1.1 and 1.2: implementing component management outside individual applications, and providing a place where heterogeneity in interfaces can be detected and addressed?

To enable management/usage orthogonality, we can externalize references in the form of a standardized document format used by both the application and middleware. The document is referenced by the application whenever it wishes to make distributed object invocations, and is referenced and/or modified by the middleware to check or update component locations.

To address heterogeneity in interfaces, we can use our document model to externalize component descriptions and user interface mappings. This hybridizes features of the two basic approaches discussed above, allowing downloading/uploading of code fragments (as specified by the documents) while imposing only a standard for interface description and manipulation rather than application-specific interface types. Structural typing, similar to the approach of [25], can used for matching of expected client interfaces to advertised service interfaces rather than named typing. Because our documents are described in a dialect of XML, XML queries [4, 22] can be used for this structural type mapping, and matching on portions of an interface (a form of subtyping) is naturally supported. Specifically, the use of the document-based approach solves two aspects of the heterogeneous component interface problem: standardization of entity descriptions as a step toward interoperable manipulation of such entities, and specification of the mapping between components and user interfaces that can access them. It does not directly solve the final portion of the problem, that of generating entities that wrapper incompatible interfaces. We mention our proposed solution to this final portion of the problem — using intermediaries that provide pairwise mapping of method invocations based on structural typing — in Section 8's discussion of continuing work.

An additional feature of solving the heterogeneous interface problem in this way is that fractions of service descriptions that are not handled by any client program can be used directly to dynamically generate a user interface. This turns out to be quite useful in the domain of "remote control" applications, our area of primary investigation.

The rest of this paper elaborates on our document-based framework. It is structured as follows. Section 2 describes the investigation approach. Section 3 introduces XML, motivates its use, and describes the markup tags we use in the interface description documents. It also explains how these tags are used in communication endpoint resolution and choosing user interfaces. Section 4 gives details on automatic user interface generation. Sections 5–6 give examples of the use of the framework for "remote control" and show document markup examples along with their related applications. It includes examples of

- remapping of a portion of an existing room's user interface to a new room's set of controls (for example, due to movement of the terminal)
- exposing arbitrary subsets and combinations of the functionality available to the user, and
- mixing dynamically-generated user interfaces with existing user interfaces to address the case where a native user interfaces are not available for all the components the user wishes to access.

Section 7 describes related work, Section 8 describes continuing work, and, finally, Section 9 summarizes and concludes.

2 Project Approach

Leveraging the eXtensible Markup Language (XML) [30] for syntax, we develop our document schema as an XML document type definition (DTD). The schema, called ISL, provides markup tags for language-independent service descriptions and for mapping UIs (programs) to referenced services and vice-versa. We then build software that can heuristically generate UIs from these service descriptions without associated custom UIs, and allows mix-and-match use of custom and generated UIs. Additionally, we built an index application that lists the collection of available UIs and services, allowing a combination of them to be interactively selected for presentation on the user's machine. Finally, we prototype applications that use the model, manually constructing and editing documents to simulate how programs would automatically manipulate them.

Our prototype application is a "universal remote control" based on a set of location-based services [9, 10]. The application provides software remote control of various rooms' devices from a mobile, wirelessly-connected laptop computer. Manipulations of application documents allows the controls to adapt as the environment changes around the user. Specifically, the manipulations provide for

- the remapping of a portion of an existing user interface to a new room control, e.g., due to movement of the terminal,
- viewing of arbitrary subsets and combinations of the functionality available, and
- mixing dynamically-generated user interfaces with custom user interfaces to address inconsistencies due to platform heterogeneity.

This functionality is easily represented as operations on documents containing the associations between between programs/UIs from the services they reference, exactly the model described above.

3 The ISL Interface Specification Language

3.1 XML

We have chosen to build atop the extensible markup language (XML) for our schema design, leveraging its allowances for the creation of custom, application-specific markup languages.

XML is an SGML subset providing self-describing custom markup in the form of hierarchical named-values and advanced facilities for referencing other documents (ala the HTML <href> tag). It is one protocol among a group that is touted as the successors to HTML. (The companion protocols are XSL for style sheets and XLL for linking mechanisms.) XML includes the ability to specify, discover, and combine a group of associated document schemata — otherwise known as document type definitions (DTDs). Examples include a growing set of metadata markup proposals such as Resource Description Format (RDF) and the Dublin Core.

Unlike HTML, the set of tags in XML in flexible; the tag syntax is defined by a document's associated DTDs. A key property of XML, then, is that it is dependent on these schema to be useful, and dependent on agreements in schema to allow interoperability. Thus, the problem of defining schema syntax (the tag set and their relationship) and agreeing on how a schema's associated "browsers" (borrowing the HTML term) semantically interpret these tags is of critical importance to XML's success.

We believe there is a natural synergy between XML's need for schemata and the specification requirements of Internet distributed object systems — the former provides a self-describing and extensible syntax with a rapidly expanding set of metadata tags; the latter provides a programming model for "Internet objects" described in XML.

3.2 ISL Usage

The key challenge in implementing our approach is defining a single schema that:

- denotes services' interfaces,
- associates relevant programs and UIs to collections of services, or, vice-versa, lists the service interfaces expected by particular programs
- can compose and decompose based on constituent elements, and
- allows for incorporation of service-specific metadata (i.e., data that should not affect existing functionality that does not expect it).

We specifiy that a single document format is shared among all entities in the system. A reference to a service looks identical to the description of the service, which allows the use of structural type matching to resolve such references. Encoding the descriptions as XML documents allows middleware entities to detect documents they may want to modify via the use of XML queries [4, 22]; thus providing substructural matching (matching against just a portion of the description) in addition to structure matching against the entire document tree. Additionally, the use of XML allows documents to be programmatically modified using the Document Object Model (DOM) [1].

As adjuncts to servers, documents act as static interface definitions, and are analogous to CORBA object IDL descriptions or the result of introspection on a Java class. As adjuncts to clients, documents act as a stable but manipulable (composable/decomposable) format for specifying service collections and references, defining interactions between services in a collection, defining the service interfaces expected by programs and user interfaces, and storing arbitrary metadata about referents. Alongside proxies — entities that act as both a server and a client — a single document fulfills both duties.

3.3 ISL Syntax

Our document markup language is called ISL, an acronym for "Interface Specification Language." We use six tags in our initial minimal design. Other tags that appear in our documents are assumed to be application-specific metadata, and can be ignored by programs that do not understand them. We now describe each tag in turn. The ISL DTD is provided in the Appendix A.

The <service> tag is a container tag. It has one optional attribute, "name", which is either a string or reference identifying the type/class of the interface being described. It can contain at most a single <label> tag, zero or one <addrspec> tags, any number of <ui> tags, and any number of <method> tags. When converted to a user interface, an <service> is instantiated as a container for widgets (a "frame").

The <label> tag provides a text description of the service which contains it. It has no optional attributes. It can contain no additional internal tags except those providing text formatting. When converted to a user interface, the <label> tag is used as a title for its parent service's frame.

The <addrspec> (address specification) tag indicates the address and port number on which its parent service listens for method invocations and events. Instantiating a service causes this tag to be added to its description; a service that has not been allocated (and thus has no addrspec) is called "unpinned." The tag can contain no additional internal tags and does not have any optional attributes. When converted to a user interface, the <addrspec> tag is used as the location to which any method calls are sent (currently via string-based UDP messages to facilitate ease of multiplexing, multicast support, and a degree of language/system independence).

The <method> tag defines the name of a method that can be invoked on the service in which it is contained. It has two optional attributes: "name", which is name of the method call, and "lexType", which indicated the lexical type of messages returned due to the method call (the list of lexical types is described below). The <method> tag can include (only) zero or more <param> tags. When used in automatic user interface generation, each <method> tag is mapped to a frame with contents. The name of the method is placed on a button at the top of this frame; pressing this button invokes the method call. Method invocations and returns are asynchronous, event-based messages rather than blocking remote procedure calls. Thus, update events ("replies") can actually occur at any time, independent of the manual invocations at the client. In this manner, <method> tags can also be used as a means for subscribing to pushed updates from a service.

The <param> tag indicates a parameter to the <method> tag that encloses it. It has two optional attributes, "lexType", indicating the lexical type of the parameter, and "optional", a boolean tag that indicates whether the parameter is required or optional. The <param> tag may have no additional internal tags, and its contents are assumed to be the name of the parameter. For UI generation, parameters are mapped to individual user interface widget objects. Widget objects are used for user input and to marshall the parameters for method invocations. Mapping from lexical type to UI widgets is described in Section 4.

The <ui> tag is used to associate a particular program to the service in which it is specified. It is unique in that there is nothing analogus to it in conventional server-side IDLs — it is useful only for clients and proxies. The contents of the tag string indicates either the name of an existing user interface object (assumed to be known or discoverable out-of-band) that will reference the document, or the address and port number from where such a user interface object can be downloaded. It has one possible attribute, "lang," indicating the language of the indicated program. There can be multiple UI tags for each service, at all levels of the service description hierarchy in an ISL document. To work with our framework, the indicated applications need to reference the document (e.g., add their own interface descriptions to it if they choose to expose one), thereby respecting the indirection exposed by the document-based approach.

4 User Interface Generation

4.1 Basic Approach

Many of the mechanics of generating user interfaces from interface descriptions were described in the preceeding section. The remaining features to be discussed are the heuristic mapping from lexical types to user interface widgets, and how custom user interfaces indicated by a <ui> tag can be intermingled with these custom user interfaces.

Dynamic user interface generation is only useful in a limited number of application domains: there is limited internal state maintenance and a lack of protocol transitions. Though both of these limitations can be addresses through additional markup, doing so blurs the distinction between the declarative nature of current design and traditional full-featured scripting languages.

We currently have implemented mappings only from primitive lexical types to objects wrapped around Tk [21] UI widgets in the MASH toolkit [15]. Permissable lexical types include int, real, boolean, enum, string. The int and real type can have an optional range modifier. They are mapped to widgets as follows: an int or real with a "range" modifier is mapped to a scale widget (a slider). Without a range modifier, they are mapped to an entry widget (a type-in box). A boolean is mapped to a checkbutton widget (a toggle switch). An enum is mapped to a list of radio-buttons (one-of-N list selection). A string is mapped to an entry widget. Structured types are expected to be expressed as (possibly hierarchical) collections in/of <service> tags, treated as aggregates through the structural type matching.

Co-mingling generated collections and existing UIs referenced in <ui> tags is done at a granularity of individual services. Thus, all services receive a frame, and it is filled with either the custom-generated contents mapped from <method> and <param> tags, or the existing UI. The latter is handed a handle to this window and is expected to instantiate itself as a child of that window.

4.2 Return Values

One approach to dealing with return values is to require per-method output type descriptions, and require that components respect these output descriptions. In this case, separate entities may be required on the reverse path to remap this data is there are type mismatches. We have not incorporated this extension into our software but believe it to be a straightforward extension. Instead, we simplify things by forcing the input and output types to be identical — a form of call-by-reference for all the arguments. This avoids the need for output specifications entirely. The simplified approach has the important benefit that reverse (response) paths can be the same as the forward (invocation) paths, just in opposite order. In the more general case, a completely separate response path might be needed, which both complicates the problem of attempting to set up such a path and adds more elements that must be checked by the user for semantics preservation. An addition important advantage is that it allows all the widgets on automatically generated user interfaces to be updated directly from the contents of the method call results, a convenient mechanism given our focus on control applications (which are amenable to use with automatic UI generation because).

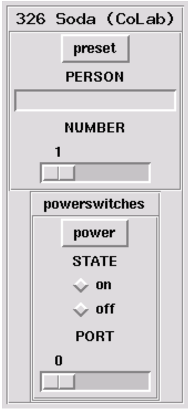
5 The Framework in Action

We now illustrate some examples. Each highlights a different element of the design of the overall architecture. We limit the scope of the examples to a single (varying) collection of services being referenced by a single (varying) user interface. Conceptually, though, the framework is amenable for use with transformational/proxy entities that are both referenced as a service by a user interface and perform references to other services to fulfill the incoming request.

The first example shows an XML document that describes the interface to a portion of the functionality available in Soda Hall's "CoLab" ("Collaboration Laboratory," borrowing Xerox PARC's terminology) and the resulting automatically generated user interface to it, as shown in Figure 2. The document describes two services, one contained in the other. The outer service implements a method for setting a preset for the entire room; the inner service is one of the services referenced by the outer one (i.e., one of the things affected by the preset) — an interface to a pair of power switches in the room. These two services, though notated and used in this hierarchical manner in this example, can also be controlled independently of one another. The <param> tags contain various lexical types, illustrating our use of heuristic mapping to widgets. This utility of this functionality is that it allows users the possibility to interact with dynamically discovered services where otherwise there is no available client program.

```
<service name="326">
        <label> Soda (CoLab) </label>
        <addrspec>spade.cs.berkeley.edu/0001</addrspec>
        <method name='preset'>
                <param lextype="string"> person </param>
                <param lextype="int:range=1-8"> number </param>
        </method>
        <service name='powerswitches'>
                <label>powerswitches</label>
                <addrspec>spade.cs.berkeley.edu/0002</addrspec>
                <method name='power'>
                        <param lextype="enum:on,off"> state </param>
                        <param lextype="int:range=0-1"> port </param>
                </method>
        </service>
</service>
```

(a) XML document



(b) User interface

Figure 2: An example document and generated user interface.

The second example illustrates combining a downloaded user interface with a generated one. The document is identical to that in the previous example except a single new tag is added: a <ui> tag to the internal (power switch) service, as shown in Figure 3(a). This causes that service's interface to be replaced by the UI object referenced in the tag rather than generated on-the-fly. The resulting difference is illustrated in Figure 3(b). This example illustrates how dynamic extensions to existing applications can be seamlessly incorporated using our architecture, a form of "plug-in" architecture similar to that used in Photoshop.

The third example illustrates use of the indirection exposed by our "document-based" model. A referent under a multiple-service <ui> tag is replaced. The document fragment shown in Figure 4(a) is assumed to be used by an existing application. The user interface for this application is a custom-designed monolithic interface referenced in the topmost <ui> tag. In Figure 4(b), one of the component services in the container has been replaced. Because the type of the referenced service remains the same, only the <addrspec> tag changes. The result of this change is that the application looks the same, but a portion of it now references a new service. This function illustrates the possibility for remapping interfaces due to, e.g., terminal mobility or fault tolerance. Specifically, the example takes a portion of the document describing the interface to the 405 Soda Hall seminar room and remaps the light switch to the one in the CoLab.

The fourth and final example illustrates the ability to easily specify the use a subset of the available functionality. The document in Figure 5(a) is the same as that from Figure 4, except all the internal services referenced from the outermost container object have been omitted. The resulting user interface is presented in Figure 5(b). This example shows how a user can easily elide material not considered relevant or not frequently used. In this case, we leave only the interface to the light switch exposed, simulating the case where the user has chosen to save screen real estate because, e.g., controlling only the lights is the most common usage.

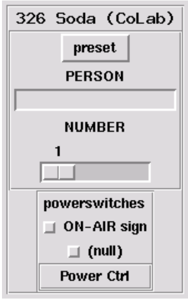
6 The User Environment

In addition to building software to parse ISL and generate appropriate interfaces, we need to provide the user with a way to manage the set of documents and available interfaces. We provide this functionality through use an "index" application, called "UC" (for "universal client"), and shown in Figure 6. One the left side of the application, all services are listed by "type" and address specification. Each type has an associated document and an associated user interface. When one of the check-buttons beside an service name is set, the associated user interface is displayed for use by the user. Locally edited files are listed in the index with their name preceded by a hyphen. This figure illustrates a case where the user has selected to interact with three services through two user interfaces. The SodaLights UI is composed of a native light application for the Soda Hall CoLab and an automatically-generated UI for room 405. The CameraUI is another native MASH shell user interface.

The Index application provides a front-end a service discovery service (e.g., [2, 27]). Once a component is discovered, if a client-side user interface exactly matching the component name is available, it is used when then

```
<service name="326">
        <label> Soda (CoLab) </label>
        <addrspec>spade.cs.berkeley.edu/0001</addrspec>
        <method name='preset'>
                <param lextype="string"> person </param>
                <param lextype="int:range=1-8"> number </param>
        </method>
        <service name='powerswitches'>
                <label>powerswitches</label>
                <addrspec>spade.cs.berkeley.edu/0002</addrspec>
                <ui lang=mash> PowerSwitchUI </addrspec>
                <method name='power'>
                        <param lextype="enum:on,off"> state </param>
                        <param lextype="int:range=0-1"> port </param>
                </method>
        </service>
</service>
```

(a) XML document



(b) User interface

Figure 3: An example document and associated user interface, this time where a <ui> tag allows for the incorporation of a custom UI in addition to the generated components.

component is selected. If not, then the components' ISL file is downloaded (component advertisements must include a URL that points to its associated ISL file) and a user interface is generated automatically when selected. In our current prototype, other operations must be affected via manual editing of the documents; we are in the process of creating helper applications that automate common manipulations such as our cannonical "remap to local room controls" example.

7 Related Work

This approach is quite similar in flavor to the use of shell scripting to associate arbitrary programs using the UNIX pipe facility [14]. Instead of using just file handles (i.e., stdin, stdout, stderr) to differentiate and route data, structured data and cross-network references can be expressed in a document.

The Configurable Chrome group [17] of the Mozilla.org open source project is investigating a related effort. This project aims to allow users to add buttons with arbitrary functions to the Mozilla web browser (outside the main browser window) via specifications notated in a XUL document [16]. Similarly, the WinAmp MP3 client provides facilities for customizing the user interface via changing widget bitmaps, or "skins" [29]. Our approach is similar to these two approaches in the use of a document as a UI description, but adds the ability to reference full client-side interfaces in the documents rather than just widgets, and proposes that such descriptions be manipulated as the program runs.

The TSpaces project at IBM Almaden had proposed MoDAL [5] as an XML-based interpreted application description language for mobile Internet devices. In the system, documents includes such information as screen size, button tags, and label tags, and there is a two-level hierarchy: MoDAL applications are downloaded to devices running the MoDAL interpreter. In contrast, we espouse using documents as peers to application programs; i.e., there is 1) a "platform" which contains the interpreter and class libraries, 2) a traditional language (i.e. a scripting language) layer amenable to wiring together platform components, creating custom user interfaces, and being passed around on the network [15], and 3) application documents that describe interfaces and how opaque programs are combined.

8 Continuing Work

A logical next step of this work is dealing with mismatched service "types." For example, assume a light switch in some locale implements a different interface than the one in the user's home environment. Rather than require the use of a dynamically-generated user interface, we'd prefer to allow for the use of an existing user-interface. To do so, we must transparently remap method invocations to the new location and also remap the call parameters to match the new type. Incorporating such functionality allows far more flexibility in the reuse of existing user interfaces and intermingling of

```
<service name="405">
        <label> 405 Soda (HTSR) </label>
        <addrspec>htsr.cs.berkeley.edu/0000</addrspec>
        <ui lang='tcl/tk'>htsr.cs.berkeley.edu/6903</ui>
        <service name='lights'>
                <label>lights</label>
                <addrspec>htsr.cs.berkeley.edu/6902</addrspec>
                <method name='power'>
                        <param lextype="enum:on,off,dim"> state </param>
                </method>
        </service>
        ...
</service>
```

(a) Original XML document

```
<service name="405">
        <label> 405 Soda (HTSR) </label>
        <addrspec>htsr.cs.berkeley.edu/0000</addrspec>
        <ui lang='tcl/tk'>htsr.cs.berkeley.edu/6903</ui>
        <service name='lights'>
                <label>lights</label>
                <addrspec> spade.cs.berkeley.edu/9999 </addrspec>
                <method name='power'>
                        <param lextype="enum:on,off,dim"> state </param>
                </method>
        </service>
        ...
</service>
```

(b) Document with replaced referent

Figure 4: Remapping of function by replacing a referent under a multiple-service <ui> tag. A fragment of the "original" document is show in (a); the modified document is shown in (b), where the only difference is the new <addrspec> tag. (The <addrspec> tags are highlighted.)

existing interfaces and discovered services, but requires the use of external transformational operators that provide type coercion for method calls. Fortunately, such transformational operators could be written once, reused, and shared among the community of users; additionally, they could be chained together in order to provide new type-to-type coercions [6]. This functionality is a natural extension of our framework. We are in the process of implementing it by applying the document type conversion approach of Ockerbloom [20] to interface conversion in a system comprised of heterogeneous distributed components. The approach allows the underlying system to evolve without forcing agreement on particular component interfaces. I.e, in the language of documents, without requiring a single specific intermediate format per document style. Instead, independent pairwise conversions can be combined in chains ("paths" using the language of Ninja [18]) to allow end-to-end interoperability amongst semantically matching – but differently typed – components. Matching transformational operators are determined through the use of evolutionary structural type matching of components [25], and implement a form of wrappering [23] to encapsulate one interface in a form usable by another. Additionally, because such transformational operators can only match based on structure, there is the potential for semantic discrepancies. To address this, our approach is to require users to "bless" the use of a particular mapping operator(s) for use in particular situations. Though such a function cannot be automated (i.e, a computer cannot understand the semantics described in a textual documentation), the overhead can be reduced by allowing such decisions to be made once and shared among groups of users.

The difficulty of this approach is not in creating these mapping operators and storing them in a shared repository, but instead that of building the use of them into the end-user software. Users should be able to visually manipulate service mappings and the correct transformations should be done automatically. As a concrete example, this means that when a new light switch is discovered, the user should be able to indicate which program element should manipulate it, and any required remapping of method calls — i.e., document manipulations — should be done automatically, though possibly heuristically. Additionally, users should then be able to

```
<service name="405">
        <label> 405 Soda (HTSR) </label>
        <addrspec>htsr.cs.berkeley.edu/0000</addrspec>
        <service name='lights'>
                <label>lights</label>
                <addrspec>htsr.cs.berkeley.edu/0000</addrspec>
                <method name='power'>
                        <param lextype="enum:on,off,dim"> state </param>
                </method>
        </service>
</service>
```

(a) XML document

(b) User interface

Figure 5: Subsetting functionality. The example illustrates how functionality can be aggregated or subsetted by modifying the document associated with a program. The full description of the interface to 405 Soda has been cut down so that only a single service remains. The user interface is updated accordingly.

easily modify these mappings.

Another important extension of this work is designing how to notate one service's use of other so as to allow for the "proxy" transformational services described above; this requires separation of input and output interface descriptions. This requires extension and modification of our schema to allow such notation and extension to the software to utilize it.

Yet another area under investigation is the need for output type descriptions as described in Section 4. Similarly, to support the evolvable structural type matching of [25], we will need to add "ignorable" and "optional" attributes.

Finally, in order to allow for programmers to more easily use this document-based model — without having to manually create interface description documents — we would like to automatically generate the documents from existing Java objects and other distributed component system pieces. To do so, we can leverage the CORBA Interface Definition Language (IDL) and Java reflection API to create descriptions in our XML schema. ISL will need to be extended to support structured parameter types in order to allow such a mapping, a straightforward, but clearly important, extension.

9 Summary

Large-scale federation (composition, management, and control) of distributed Internet components requires reconsideration of how applications are structured; an ideal is to allow manipulation in response to changes in needs or component availability while not burdening application designers with details they may not require. We argue that distributed object models used with traditional application development techniques are insufficient for this purpose because they make remote object references look like static, unchanging pointers. Instead, changes should be affected at the level of a middleware layer (as they are now), but also allowed to be communicated upward to the applications in a manner allowing the applications to either ignore such changes or use them. This paper proposes a new model, a *document-based* framework, for description and interaction with Internet services. We describe a simple version of the framework in the context of remote control applications, illustrating how it allows for:

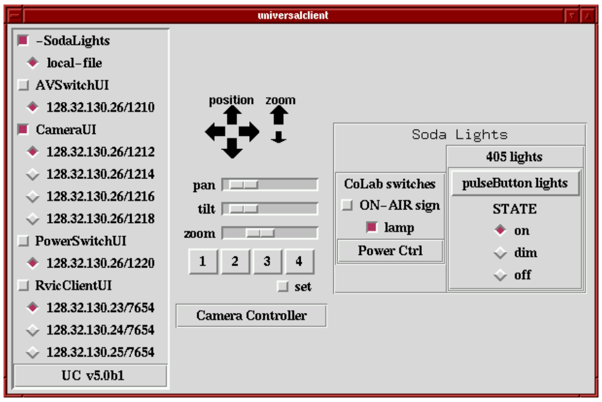


Figure 6: The Index application lists all locally available interfaces and allows the user to interactively select which ones he or she wishes to use. Illustrated here, the user has selected the aggregate user interface to some power switches and an interface to a remote-controllable camera.

- the remapping of a portion of an existing user interface,

- viewing of arbitrary subsets and combinations of functionality, and
- mixing dynamically-generated user interfaces with existing user interfaces.

The use of a document-based framework exposes an indirection between programs/UIs and the remote objects to which they refer, making this mapping explicitly manipulable, and can be used to generate user interfaces when custom ones are not available or unacceptable. It also forms the basis for continuing work on addressing interface heterogeneity through the use of structural typing, paired with wrappering in "proxy" transformational operators.

To implement our scheme, we use an XML-based language, ISL, and accompanying software. ISL:

- denotes services' available functionality, or interface, in a manner designed for interpretation and ease of manipulation at clients,
- can flexibly compose and decompose based on constituent elements, and
- allows for easy incorporation of service-specific meta-data via the self-describing, extensible nature of XML.

10 Availability

The software described herein is available in prototype form as part of the MASH toolkit, which can be downloaded from `http://www-mash.cs.berkeley.edu/mash/`.

11 Acknowledgments

The authors would like to thank the students, faculty, and staff of the MASH, Ninja, and Iceberg projects at UCB. Thanks to Michelle Munson at IBM Almaden for fruitful late-night discussions on future directions. We would also like to thank the anonymous reviewers, whose detailed commentary led to improvements in this paper (we hope). This work was supported in part by grants from Ericsson, Intel, Sprint, and Motorola, DARPA through contract DABT63-98-C-0038, the NSF through infrastructure grant CDA 94-01156, and the California MICRO program.

A Schema DTD

The document type definition (DTD) for the inital, minimal version of ISL is as follows:

```
<!ELEMENT service (label?, addrspec?, ui*,
                  method*, service*)>
<!ATTLIST service
   name  CDATA   #REQUIRED>
<!ELEMENT method (param*)>
<!ATTLIST method
   name  CDATA   #REQUIRED>
<!ELEMENT param (#PCDATA)>
<!ATTLIST param
   name  CDATA   #REQUIRED
   lexType  (int | real | boolean | enum
                  | string | ...) 'string'
   optional  #BOOLEAN>
<!ELEMENT label (#PCDATA)>
<!ELEMENT addrspec (#PCDATA)>
<!ELEMENT ui (#PCDATA)>
```

References

[1] Tim Bray and Lauren Wood. The W3C Document Object Model (DOM) – A Programmer's View of Documents. *The Gilbane Report on Open Information and Document Systems*, 6(4):1–13, 1998.

[2] Steven Czerwinski, Ben Zhao, Todd Hodes, Anthony Joseph, and Randy Katz. An Architecture for a Secure Service Dis covery Service. In *Proceedings of the Fifth Annual International Conference on Mobile Computing and Networking*, Seattle, WA, August 1999. ACM.

[3] Michael L. Dertouzos. The Future of Computing. *Scientific American*, August 1999.

[4] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A Query Language for XML, August 1998. `http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/`.

[5] K. Eustice, T. Lehman, A. Morales, M. C. Muson, S. Edlund, and M. Guillen. A Universal Information Appliance. *IBM Systems Journal*, pages 454–474, August 1999. (to appear).

[6] A. Fox, S. Gribble, Y. Chawathe, and E. Brewer. Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives,. *IEEE Personal Communications, special issue on adaptation*, August 1998.

[7] Armando Fox. *A Framework for Separating Server Scalability and Availability from Internet Application Functionality*. PhD thesis, University of California, Berkeley, 1998.

[8] Hewlett-Packard. e-Speak White Paper. http://www.hp.com/go/e-speak/, 1997.

[9] Todd Hodes, Randy Katz, E. Servan-Schreiber, and Larry Rowe. Composable Ad hoc Mobile Services for Universal Interaction. *Proceedings of the 3rd ACM International Conference on Mobile Computing and Networking*, pages 1–12, September 1997.

[10] Todd Hodes, Mark Newman, Steve McCanne, James Landay, and Randy Katz. Shared Remote Control of a Video Conferencing Application. *SPIE Multimedia Computing and Networking*, pages 17–28, January 1999.

[11] InfoSpheres. The InfoSpheres Project. http://infospheres.cs.caltech.edu.

[12] G. Krasner and S. T. Pope. A Cookbook for Using the Model View Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, August/September 1988.

[13] David Krieger and Richard Adler. The Emergence of Distributed Component Platforms. *IEEE Computer Magazine*, pages 43–53, March 1998.

[14] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison-Wesley, Reading, MA, Nov 1989.

[15] Steven McCanne et al. Toward a Common Infrastructure for Multimedia-Networking Middleware. *Proc. 7th Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '97)*, May 1997.

[16] Mozilla.org. Introduction to a XUL (XML-based User Interface Language) Document. http://www.mozilla.org/xpfe/xptoolkit/xulintro.html.

[17] Mozilla.org. Mozilla Configurable Chrome. http://www.mozilla.org/aurora/config.htm.

[18] Ninja. The Ninja Project. http://ninja.cs.berkeley.edu.

[19] Object Management Group. Common Object Request Broker Architecture. http://www.omg.org/.

[20] John Mark Ockerbloom. *Mediating Among Diverse Data Formats*. PhD thesis, Carnegie-Mellon University, 1998.

[21] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Publishing Company, Reading, MA, 1994.

[22] Jonathan Robie, Joe Lapp, and David Schach. XML Query Language (XQL). In *QL '98 - The Query Languages Workshop*, December 1998.

[23] M. T. Roth and P. Schwarz. Don't Srap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. *Proceedings of 23rd VLDB Conference*, 1997.

[24] Bill N. Schilit, Norman I. Adams, and Roy Want. Context-Aware Computing Applications. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 85–90. IEEE Computer Society, December 1994.

[25] Mike Spreitzer and Andrew Begel. More Flexible Data Types. *IEEE Eighth International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprisese*, 1999.

[26] Sun Microsystems. Enterprise Java Beans. http://java.sun.com/ejb.

[27] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. *Service Location Protocol Internet Draft #17, draft-ieft-svrloc-protocol-17.txt*. IETF, 1997.

[28] Jim Waldo. The Jini Architecture for Network-centric Computing. *Communications of the ACM*, pages 76–82, July 1999.

[29] WinAmp. WinAmp skins. http://www.winamp.com/skins/.

[30] World Wide Web Consortium. eXtensible Markup Language. http://w3c.org/XML/.

[31] P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. T Spaces. *IBM Systems Journal*, 37(3):454–474, August 1998.