WS-OBJECTS: Extending Service-Oriented Architecture with Hierarchical Composition of Client-Side Asynchronous Event-Processing Logic

Krzysztof Ostrowski and Ken Birman
Department of Computer Science
Cornell University
Ithaca, NY 14853, USA
{krzys,ken}@cs.cornell.edu

***Abstract*—There is a growing need for a new type of WS-*/SOA standards that could facilitate hierarchical, object-oriented composition of client-side executable code. This is especially true for the sorts of client-side logic embedded in AJAX and rich Internet applications, virtual worlds and MMORPGs; code that deals with issuing requests to servers, processing their responses, rendering UI, interacting with users, and processing asynchronous events from other client nodes. The paper offers an analysis of client-side composition patterns, a brief explanation why they lack adequate support from the existing web technologies, and design guidelines for client-side component integration environments to follow. The proposed guidelines have been successfully implemented in a prototype system [16]. Our analysis is thus strongly rooted in reality; it is based on real experiences with concrete application scenarios. The paper concludes by highlighting the key architectural aspects of our implementation with respect to the principles listed earlier.**

I. Introduction

The term *service-oriented architecture* (SOA) is often regarded as synonymous with *web services* (WS); indeed, the latter is by far the most successful realization of the SOA paradigm. The existing WS-* technologies are limiting, however, in that the functionality implemented by a web service is assumed to reside entirely at a remote server identified by a URI. Consequently, WS/SOA interoperability standards such as BPEL [9] are designed mostly to facilitate building server-side logic and assume that end user's client machines are passive consumers.

Accordingly, WS standards specify how *proxies* running on the client machines should format requests, address and locate remote services, and process responses, but they don't address scenarios in which clients might execute parts of the service logic locally, constitute parts of the service itself, or cooperate with one-another. Our work attempts to address this limitation.

In principle, SOA requires that services be *loosely coupled* and *interoperable* [4], but it makes no assumptions about code placement. In Jini [18], which is also a realization of the SOA paradigm, the Java code of the service could be automatically downloaded to the client and executed locally. Binding is done not via SOAP, but via Java interfaces. The downloaded code can be nothing more than a thin proxy for a remote service, but it could also include complex processing at the client machine, or even peer-to-peer functionality. In this regard, Jini approach offers more flexibility than WS. On the other hand, being truly language agnostic, WS is certainly more platform-independent and interoperable, and perhaps it is these factors that made it so successful. In this context, one can view our proposal as an attempt to port and integrate some of Jini's ideas into the established WS-* architecture in a platform- and language-agnostic manner to achieve combined benefits of the two approaches.

In many applications, client-side code in which WS proxies are embedded could be a complex piece of software with many independently developed moving parts; indeed, as we argue in the next section, different parts of such code could be created without any coordination on behalf of their developers. Thus, explicit support is needed for composing client-side modules that could be written independently and in different languages.

In the longer perspective, extending the WS paradigm in this manner is essential because focusing exclusively on the server side is at odds with the recent trends in Web programming. With technologies such as ActiveX, Flash, or AJAX, a browser on a typical Web surfer's machine is becoming increasingly involved in processing and rendering web content; interaction with the user is mostly local, and web requests are made only sporadically, to update parts of the displayed content.

These trends are likely to continue, since on one hand, the computational power of client machines is steadily increasing, while at the same time, large content providers are increasingly running into scalability challenges as their content gains on popularity. Shifting some part of the computational burden to client machines allows providers to offload servers in the data centers, and the clients benefit from improved responsiveness.

The trend to offload processing to clients is likely to get even deeper: to admit more users, even the responsibility for storing content could eventually be shifted to clients; these would use *peer-to-peer* (P2P) replication protocols to coordinate updates. Technologies based on this idea already exist [6], [16], [17]. Croquet [17] replicates content with *two-phase commit* (2PC). *Live distributed objects* [16] support user-defined protocols.

In past work [14], we argued that offloading work to clients is particularly beneficial for highly interactive, short-lived, and fast-changing content that is difficult to efficiently cache and index on the server side; such content can be hard to scale by simply deploying more servers. Currently, this includes mostly live chats, video conferences, massive-multiplayer online role-

playing games (MMORPG) such as World of Warcraft (WoW), and virtual worlds such as Second Life (SL), but we expect that these emerging technologies and platforms will increasingly converge and blend with the more traditional, AJAX-style and mashup-style Web content. Stronger focus on the client-side might, indeed, facilitate such technology convergence.

It isn't hard to see that in several of the scenarios mentioned earlier, ranging from AJAX-style applications and "Web 2.0" mashups to applications that incorporate elements of virtual reality and MMORPG-style entertainment, the client-side logic could become fairly elaborate and consist of multiple internal layers and components. Like most complex software artifacts, it could have a hierarchical structure, and its constituent parts could come from multiple sources. Hence, support for modular and hierarchical composition is essential. In the next section, we discuss client-side composition patterns in more detail.

Today, client-side logic is implemented mostly in JavaScript (JS), which effectively serves as a gluing layer. This is far from perfect. One issue is that as a language-specific technology, JS isn't platform-agnostic and interoperable. Another issue is that JS scripts are, for the most part, isolated chunks of code that may be mashed together side-by-side on the same page, but are not designed to be composed hierarchically and interoperate. Reusability has to be built-in, coded explicitly at design time. For example, if a given UI component with embedded JS is to superimpose financial data from a number of sources on a single chart, it has to be explicitly programmed to support all the different data and request formats, for all of these sources.

Ideally, we'd like to treat parts of the client code that handle data retrieval and view rendering as separate objects that could be composed as easily as one composes a workflow in BPEL: by dragging them onto a design sheet. The same data retrieval object could then be reused in combination with arbitrary view rendering objects. In order to implement this in JS, however, JS script developers would need to first agree on a number of practical issues, including which data structures to use to pass parameters, in what libraries and namespaces they should be defined, where these libraries should be stored, how to upgrade them, whether and how to convert method parameters if their types mismatch, how to name methods and avoid name clash, how to allocate threads and avoid deadlocks, whether to handle method calls asynchronously or not, etc.

By agreeing on issues such as those just listed above, developers would in effect be defining a JS-specific interoperability standard that extends the core JS framework. In this paper, we develop a set of design principles and guidelines for achieving a subset of these goals in a platform-independent manner.

This paper makes the following contributions.

1) It motivates an extension of the WS-*/SOA architecture with interoperable mechanisms that can facilitate client-side composition of executable code – an alternative to server-side composition such as those based on BPEL. It discusses practical examples of client-side compositions, and abstracts them out into general composition patterns.
2) It outlines some of the key reasons why the existing web technologies do not adequately support our composition patterns. The paper focuses especially on the weaknesses of JS as a client-side gluing layer. In the course of this analysis, we formulate a set of general design principles that client-side composition technologies should follow.
3) It describes an example realization of these principles in a client-side composition platform we developed [16].

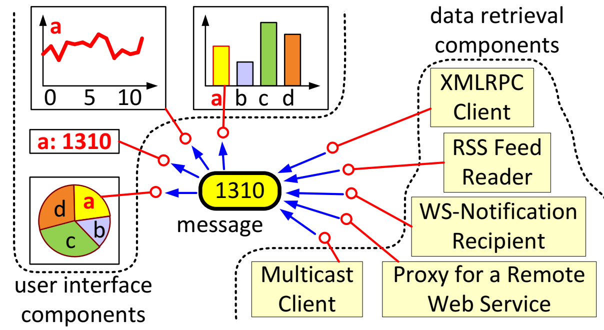

Fig. 1. UI and data retrieval components interact by explicit message-passing. Messages are exchanged locally: the components reside on the same machine, for example in the end user's browser. Modeling such component interactions as explicit asynchronous message exchanges enables decoupling and promotes component reuse. Here, each of the data retrieval components could be mashed together with any of the rendering components to present the data to the user.

Our analysis is strongly rooted in reality; it is based on over 1.5 years of practical experiences with a real system – our *live distributed objects* (LO) platform – and concrete application scenarios. The platform can be downloaded for free [16]. More information about the underlying programming model that the plaform is based on can be found in our prior work [14], [15].

II. Compositions

In this section, we discuss examples of client-side composition and abstract them out into general patterns. We point to limitations of the existing web technologies, with focus on JS. We formulate architectural guidelines for future extensions.

A. Data Retrieval and User Interface

This form of composition has already been mentioned once in the introduction, and leads to the following design principle.

Principle 1. *The part of client-side code that pulls information from remote services or other sources should be decoupled from the part that visualizes it and handles all interactions with end users; the two tasks should be treated as distinct concerns, and handled independently by separate components.*

For example, the data retrieval component for a stock quote might output a stream of *float* values; these could be consumed by a UI component that shows the last value as a number, in a text field on a web form (Fig. 1, Fig. 2). The decoupling allows us to substitute the UI component with a different one, e.g., one that shows the last 100 values in a table, one that shows them as a time series, or one that juxtaposes the last value with values from a few other sources on a pie or bar chart; this way, the same data could be interpreted and visualized in different ways just by replacing the frontend. Likewise, different types of data retrieval components could be placed at the backend

without replacing the frontend. One could retrieve values from a web service, another from a multicast channel, a publish-subscribe plaform, an RSS feed, or via WS-Notification. Any frontend could thus be used with any backend as long as data is fed to it in the matching format, using matching interfaces.

In the example on Fig. 1, messages are carrying *float* values, but in general, messages can be quite complex. One example is discussed in the following section: with shared folders, shared desktops, and other types of containers, messages exchanged between the data retrieval and UI rendering components could carry descriptions of objects enclosed in a container: airplanes, maps, etc. Encoding such descriptions might require elaborate data structures that would need to be somehow standardized.

Out of the box, this pattern is not supported in plain HTML: web content is generally addressed directly, loaded over HTTP, and browser must be able to recognize it. One can decouple processing in this manner using a scripting language such as JS. The difficulty in implementing this pattern using JS, or in general in language-centric approaches, is that data structures and interfaces for frontend-backend interactions would need to be defined in shared libraries that the interacting components would reference. This is often unrealistic, as explained below.

Note that unlike a regular Java application, which is assembled at design time, where all conflicts are easily resolved by a programmer, and where external dependencies can be bundled with the redistributable code, web mashups usually emerge as a result of drag-and-drop actions performed by the end users. In our example, an end user may try to drag a new data source onto an existing bar chart. Different users might successively add components downloaded from different sources. Creators of those components might not have anticipated their use in the particular type of a mashup, and end users will often lack the programming expertise necessary to deal with issues that arise during composition. This motivates the following principle.

Principle 2. *The platform should support composing client-side code dynamically on the end hosts; it should be assumed that by default, it can involve arbitrary components developed independently and never specifically designed to interoperate.*

Consider the consequence of this principle in an approach such as JS that relies on shared libraries. Clearly, one cannot expect that there would exist a single Internet-wide authority standardizing Java interfaces that every web developer would use. Furthermore, libraries tend to evolve; this brings the issue of managing their versions. If X requires an older version, and Y requires a newer one, linking the two may be impossible; most managed environments, including .NET and JVM, have limitations in this regard. This leads to the following principle.

Principle 3. *Composition cannot rely upon the existence of shared libraries, shared interfaces, shared data structures, or any other shared artifacts that require binding at compile-time. The runtime on the client should facilitate interaction between components that export binary-incompatible interfaces.*

With this sort of a *shared-nothing* approach, the only thing that a pair of components could have in common is the logical

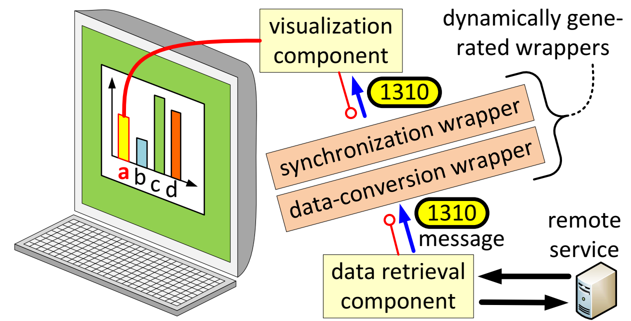

Fig. 2. To resolve binary incompatibilities between component interfaces and avoid synchronization dependencies, the runtime must be able to automatically inject dynamically generated wrappers that convert data structures and forward method calls between components and add asynchrony as needed through the use of event queues, futures, and similar types of mechanisms. The process of generating these wrappers could be facilitated through semantic annotations.

structure of their interfaces. Indeed, the reader will recognize in this the core principles that underpin the WS-* architecture: two components can interact if only they agree on *logical* data types, encoding, and the protocol. This leads to the following.

Principle 4. *Client-side composition should be based upon structural conformance; if two components expose interfaces with similar structure, and there exists logical correspondence between events, calls, and fields in data structures, the runtime should treat these components as compatible and dynamically generate wrappers that translate method calls (Fig. 2).*

Structural conformance is feasible with existing technology, and it is supported by our *live distributed objects* platform. In the scenario on Fig. 1, if a components X generates a stream of *float* pairs that represent coordinates and component Y accepts *double* pairs, our platform will treat X and Y as compatible.

Note the connection to semantic WS composition [10]. Suppose that our components have semantic annotations and when the end user creates a new mashup, the runtime environment processes these annotations to determine semantic similarities, such as correspondence between events, methods, or fields in data structures, much in the same way semantic composition is used in WS to find correspondence between services and their parameters. In the absence of such semantic annotations, the process of matching components falls back onto a plain variant of structural conformance, where the corresponding interfaces, structures and fields are expected to have matching names.

Composing arbitrary pairs of components has another, more subtle consequence: when component X makes a synchronous call to component Y while holding a lock on a resource or data structure, a synchronization dependency is created between the two. This can easily lead to deadlocks. Normally, this is not an issue for components that were designed to work together and implicitly follow consistent policies, such as the order in which locks are acquired, but if components are coded independently and can be composed in an unpredictable manner, nothing can be guaranteed. This leads to the following principle.

Principle 5. *The runtime should avoid synchronization dependencies between components. Wherever possible, it should*

enforce strictly asynchronous interfaces: one-way method calls or explicit message passing. Alternatively, it should be possible to decorate components with semantic annotations that specify synchronization and locking aspects, for use by the runtime to inject wrappers that queue events, create futures etc. (Fig. 2).

In the live distributed objects platform, we adopted the latter approach: method calls into a component can be automatically made asynchronous and lock-free with a one-line annotation. The rationale for this decision is discussed in the next section.

With the ability to place autonomous data retrieval components at the backend and asynchrony in the model, the question arises whether these components should be able to create their own threads of execution, for example to poll RSS feeds, or to issue synchronous socket operations. In general, this would be convenient, but as it will become clear shortly, composition in our model is fine-grained and the number of client components can be large. This suggests the use of the asynchronous event-driven model not just for inter-component interactions, but also *inside* the components. This motivates the following principle.

Principle 6. *Client-side code should take the form of short and terminating asynchronous event handlers; alternatively, it should be translated or compiled into such form. The client-side runtime should offer lightweight scheduling, asynchronous I/O, and timer event mechanisms that can be used to implement background processing within components without the need to create threads on a per-component basis.*

By now, one may be tempted to view client-side components as miniature web services, and solve several of the difficulties we pointed to by marshaling calls between them much in the same way WS proxies marshal their calls to remote services, but this would be unacceptable for performance reasons. We've just pointed out that client-side composition is much more fine grained that WS composition. A typical client-side component would be much smaller, more lightweight than a typical web service (this becomes evident in the following section, when we consider components such as avatars and icons overlaid on a map). Consequently, cross-component interaction overheads must also be much lower. This leads to the following principle.

Principle 7. *Message-passing overhead should be minimal. In particular, the cost of marshaling calls or dynamic dispatch through reflection is unacceptable; for each pair of interacting components, the runtime must be able to generate a dedicated, efficient, optimized wrapper code, and load it into the process. This also rules out calls across processes and application domains: client-side components should share address space.*

This observation is essential, for it lets us filter out simplistic approaches. It also helps us to position client-side composition as a technology that lies somewhere in-between WS and Java: we require some of the interoperability characteristics of WS, while at the same time, we require performance characteristics similar to Java/.NET objects; this mix of requirements requires a new type of runtime architectures. It may require new types of language or compiler mechanisms and explicit support from browsers; client-side composition is thus a category on its own.

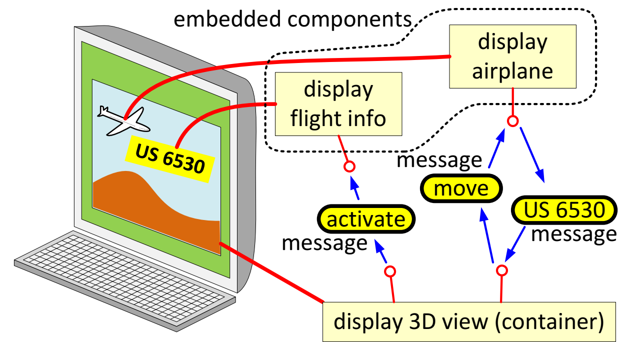

Fig. 3. A container loads, manages, and interacts with components embedded in it. Each embedded component, e.g., a plane icon or a pushpin, encapsulates all the logic for retrieving its own data and rendering itself on the screen. This way, the container (the 3D area view) can simultaneously display a variety of embedded components, even completely new types of components that might not have been anticipated at the time the container was being implemented.

B. Container and Embedded Objects

In this style of client-side composition, a *container* component may control multiple *contained* components: its elements. The container and its elements run concurrently, and each may own a portion of the screen, but the container loads, initializes, activates, and deactivates its elements, and it may interact with them via explicit message-passing as in the preceding section.

For example, a component that displays a 3D area view or a 2D map might contain components that represent commercial flights. For each flight managed by the container, a plane icon might be drawn within the area, hovering over an appropriate portion of the map (Fig. 3). The encapsulating container might activate and deactivate the embedded plane icons, depending on which airplanes are close to and within the viewing angle of the user's camera. Plane icons, however, are otherwise separate components that encapsulate all their rendering logic, and the logic for retrieving their planes' positions from remote sources and converting them to 2D coordinates for screen positioning.

The separation of the embedded components' view rendering and data retrieval logics from the logic of the encapsulating container (the 3D view component) is essential, for this allows the container to manage arbitrary components as elements as long as they support the standard container-contained control interface. In contrast, in the existing AJAX-style applications, such as Google Maps, support for embedded elements (such as pushpins) has to be built-in at design time; it is not possible to embed just any sort of web content within such a map and associate it with a location on the map unless the map *explicitly* implements this sort of interoperability (and often it doesn't).

There are several problems with implementing this scenario using existing technologies. First, although JS can modify the appearance of web page elements, the script and the document structure on which it operates are disjoint. The document thus acts like a global variable against which code is executed. This breaks encapsulation and modularity, and makes it difficult to follow the modern component-oriented development approach. Indeed, JS code is often custom-made for a particular web site and monolithic. Separation of visual content from executable

code also cripples dynamic reflection capabilities: even though one can write JS code that dynamically loads scripts or content at the client-side, the two are processed in very different ways.

A further issue is that, while AJAX supports 2D forms, other types of visual content, such as the 3D window on Fig. 3, have to be coded as Flash objects or natively in JS. As noted earlier, if Google Maps were to allow pinning Flash animations to a map, the rendering engine would need to explicitly implement this. Likewise, a Flash animation cannot easily embed JS script of HTML code. Thus, once the developer leaves the document hierarchy and drops into raw scripting, where drawing is done manually, there's no turning back: one can no longer rely on a browser support or interoperable standards; beyond this point, logic is monolithic: any flexibility has to be explicitly built-in.

In this context, we favor the approach adopted in Microsoft's *object linking and embedding* (OLE) [2] standard, where every visual component within a compound document is responsible for drawing and serializing itself. The problems just discussed can be avoided if we uniformly adopt the object-oriented perspective: think of UI rendering as a client-side service, and parts of a web page as components that provide such service among other client-side functionality they might offer. The role of the container is then limited to controlling the lifetimes of objects embedded in it, interacting with them, and managing its local portion of the document hierarchy, as postulated earlier. In this model, embedded components excecute concurrently with the container; the latter no longer needs to participate in rendering their UI or managing their data other than binding them to the display, loading resources, controlling camera or perspective. We can summarize this discussion in the following principles.

Principle 8. *Containers should be decoupled from elements embedded in them; the latter should encapsulate their element-specific logic such as UI rendering, data retrieval, or storage. Container code should be limited to handling generic aspects, such as controlling the lifetimes of the embedded elements.*

Principle 9. *The runtime must provide standardized management APIs, through which containers can control lifetimes of their components, enumerate or query supported interfaces, bind their elements to the display, unbind, move, change their sizes or perspectives, request re-rendering, drag-and-drop, etc.*

In practice, in order for a container to display the embedded components, it has to load and parse their descriptions in some standardized format. This leads to the following observation.

Principle 10. *The platform should support marshaling arbitrary types of components to portable specifications expressed in a platform-independent language that one can programmatically store, send over the network, unmarshal, and instantiate.*

In the preceding section we postulated separation of UI and data retrieval logic. This should be applied to containers, too. Actions such as adding new elements to the container will thus also need to be represented as events carrying descriptions of elements to embed; the types of these events could be complex.

Now, let's return to our example. Our area view (container) might want to highlight all airplanes that meet certain criteria, or bind to display only airplanes within a certain viewing angle and distance from the user's camera, e.g., to limit the number of airplanes on the screen for performance reasons. Since, as postulated earlier, the container shouldn't contain any element-specific logic, it must get this information from the embedded components: it needs to *locally* interact with these components.

This is again hard to achieve in the HTML/JS model, since there is no standard way for elements of the document to communicate. The OLE approach is again superior: each viewable component, besides standard UI interfaces for view rendering, can expose custom programmatic interfaces. To implement this in JS, there would need to exist a 1-1 correspondence between UI elements and the associated JS objects that perform all the data retrieval and that handle calls from other objects. This is not only inelegant, but raises security concerns; each JS object could bypass such event dispatch mechanism and interact with any part of the web page, or with objects that may be unrelated in the document hierarchy. We can summarize this as follows.

Principle 11. *Components should be able to expose asynchronous event-based interfaces alongside user interfaces. The runtime should not distinguish visual content from non-visual components; it should be able to handle both types uniformly.*

Principle 12. *Runtime must be aware of the hierarchical relationships between components; these should only be able to interact with components directly related in the hierarchy, such as embedded objects, encapsulating containers, or other components they've been otherwise explicitly associated with.*

Finally, note that elements of a web page are untyped. In our example, the container may want to differently treat embedded objects such as planes as opposed to 2D map overlays, weather information channels, and video feed pushpins. They could be displayed in different portions of the view, and the container might need to interact with them differently. Although JS has limited reflection and introspection, this functionality appears to be neither convenient, nor particularly useful in this context, for again, as pointed out earlier, JS lacks object abstraction that would encompass graphical content and its associated scripting logic. This observation leads to the following postulate.

Principle 13. *All client-side components should be strongly typed. The type system should encompass content types as well as event-based interfaces; component type should thus be determined by factors such as the list and structure of all event-based and graphical interfaces they expose, but also semantic annotations that might capture various non-functional aspects. The runtime should perform dynamic type checks during composition and provide reflection and introspection capabilities.*

To conclude, let's put the container-contained pattern in context of the preceding section, where we advocate asynchronous event-based decoupling. Note that for certain cross-component interactions, asynchronous interfaces might be impractical. For example, UI controls in platforms such as Windows Forms or Microsoft 's XNA framework are implemented by overriding a number of callbacks that have to run synchronously; this may be necessary, e.g., to render the view in a hidden buffer before

copying the complete scene into foreground. In such cases, the invoked component also has to run in the context of the callee, for it might require direct access to memory or handles shared with the callee; hosting it in a separate application domain, or isolating it in its own address space, would slash performance. That's why to handle synchronization dependencies in the live objects platform, we chose to rely on semantic annotations and automatically generated wrappers (Fig. 2) to inject asynchrony into cross-component calls rather than enforcing strictly asynchronous interfaces. We summarize this discussion as follows.

Principle 14. *The platform should support synchronous interfaces side-by-side with asynchronous event-based interfaces to support interactions such as those between container and its elements; pairs of such components may have to share resources, handles, and address space, and interact synchronously.*

In practice, in order to avoid synchronization dependencies, the platform may require, e.g., that calls are only synchronous in one direction (container calling its element). Whether A can synchronously invoke B can be inferred from their types (e.g., from the information embedded in their semantic annotations).

C. Application and Transport Layers

In this pattern, *application-level* logic that formats and serializes data sent over the network, applies client-side processing such as compression, encryption, or fragmentation, and applies updates to local data structures on the client is decoupled from *transport-level* logic that handles physical packet transmission.

For example, an embedded video may be rendered from an MPEG-2 stream. The video may be first downloaded as a byte stream by a *transport object*, converted by an *application-level* object into a sequence of video frames, and finally passed on to a UI object for on-screen rendering (Fig. 4). By decoupling transport logic from higher layers, we gain much flexibility, for example to feed the byte stream either from a server via HTTP, or use a peer-to-peer protocol such as BitTorrent. Likewise, the same transport may be used to load other sorts of content, e.g., other video formats, images, textures, or even client-side code.

For the most part, existing technologies do not support this; most content types can be loaded only using one of the built-in protocols, such as HTTP. For other types of transport, one would typically build a custom ActiveX control that encapsulates transport, decoding, and rendering logic (such as for movie streaming). This discussion motivates the following principle.

Principle 15. *Any major component and piece of content on a web page should be able to specify a custom transport to use for downloading any visual, non-visual, passive, or executable content or resource, including images, textures, scripts, descriptions of embedded components or other types of parameters. It should be possible to explicitly embed a specification of the custom transport component within the body of the web page.*

The lack of logical decoupling of the transport layer cripples also interoperability standards such as WS-Notification. These determine not only the structure and format of events, but also details of the distributed protocol used for dissemination, thus

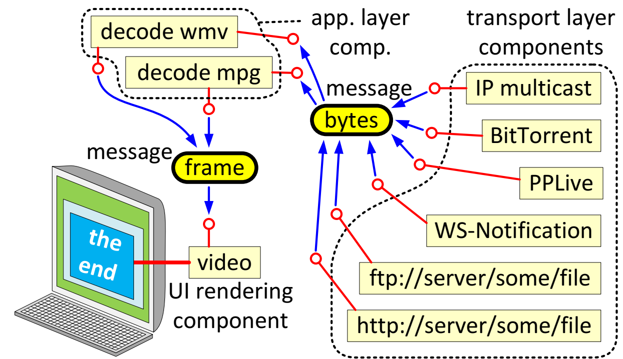

Fig. 4. Transport components encapsulate all aspects involved in the physical transmission of packets; higher layers work with pre-downloaded byte streams.

eliminating any flexibility in this respect. We've discussed this in past work [13]. The architecture we proposed to address the shortcomings of WS-* standards involves client-side logic that might perform tasks such as peer-to-peer data forwarding. This proposal is essential in making such architectures possible.

Going further, one can apply this reasoning to client-service access protocols, including HTTP GET, POST, and SOAP. An underlying transport component could handle aspects such as locating the remote service (perhaps among the clients if it was offloaded), establishing a connection (perhaps using additional mechanisms, such as hole-punching to pass through NATs and firewalls), authenticate, negotiate security protocols, and so on. Once established, a logical point-to-point channel would then be passed on to the application-level component to handle formatting, encoding, and serialization, match requests with responses, or serialize outgoing calls. This sort of flexibility opens a wide range of possibilities; e.g., one can use it to incorporate peer-to-peer service discovery mechanisms based on JXTA [1], and fault-tolerance mechanisms such as those described in [3].

Principle 16. *It should be possible to use a custom transport component for remote method calls by embedding its description within the document. The transport would encapsulate all logic for establishing a session with a remote host, over which SOAP and other application-level protocols could be placed.*

Upon closer inspection, both application- and transport-level components may themselves be modular and hierarchical. One example was just suggested above: each transport component can be further decomposed into sub-components that encapsulate tasks such as discovery, hole-punching, and authentication.

In the introduction, we mentioned that offloading content to clients would require the use of distributed protocols such as multicast; these protocols also often involve several layers. For example, the base layer that forwards updates between clients could depend on an external membership service. The protocol for retrieving membership would ideally be encapsulated as a separate object, so that it could be easily replaced, or used with other types of protocols that use membership to self-organize. Perhaps the most natural way to support such compositions is to treat functional sub-components as parameters; thus, a group membership protocol stack would be a parameter passed to the

multicast protocol stack; the latter should be able to work with any membership protocol that feeds the necessary information. Functional and non-functional requirements can be expressed as the types of the requested parameters much in the same way one specifies types of parameters in Java method signatures.

Principle 17. *It should be possible to parameterize any component with other components. Specifications of components passed as parameters should be recursively embedded within the document, as a part of the specification of the component being parameterized. The parameterized component should be able to place constraints on the types of each of its parameters.*

One may question whether embedding descriptions of multi-layer protocol stacks in documents loaded by the clients is the right approach, for it increases document sizes and costs time to parse and interpret these descriptions at the client. Indeed, except for our platform, we know of no other examples of this approach. Croquet is also based on peer-to-peer replication, but it uses a fixed protocol: a variant of two-phase commit (2PC).

In past work [14], we argued that different types of content require different replication semantics. Some types of content, such as documents that users can simultaneously edit, require stronger consistency, such as totally ordered reliable multicast, and tend to be inherently non-scalable. Other types of content, such as video streams, may not require sophisticated reliability or ordering properties, but need to scale to thousands of users, offer low jitter and flow control, etc. Many of these properties are mutually exclusive, hence no one-size-fits-all solution can exist. Furthermore, as noted in [14], even if we focused on one type of content, such as documents that support collaborative editing, upon reviewing past work in the area we find a great variety of different approaches to this seemingly trivial task. By permitting flexible protocol stack compositions on the end host, we open up the possibility for innovative solutions to be immediately tested and deployed. In contrast, standardization of concrete distributed protocols and peer-to-peer interactions patterns (as is the case in Croquet, but also in the WS-* family of specifications) tends to benefit a fraction of applications.

D. Controller and Controlled Objects

A typical web page contains many elements that need to be fetched via HTTP and services to connect to over POST, GET, or SOAP, all of which requires establishing TCP connections. All such connections are managed by a single TCP stack. With some of the extensions we proposed, we could apply the same pattern to other types of connections and distributed protocols: for example, a single page might contain multiple components that download data from the same publish-subscribe platform, establish encrypted connections to the same remote server, etc. Rather than having each component work independently of all the others, and internally create its own private instance of the protocol stack, its own connection, session, and data structures, one might prefer to have a single *controller* component in the process that simultaneously controls multiple logical channels, sessions, or connections, and can reduce overhead by applying cross-channel optimizations: aggregation, batching, clustering.

Principle 18. *It should be possible to specify components as relative to a controller that acts as a factory for and manages a collection of subordinate objects. The runtime should reuse the same instance of the controller component to support multiple controlled components within the same page, or across pages in a browser process, to facilitate cross-channel optimizations.*

For example, our page may contain a single specification of a publish-subscribe protocol stack embedded in it, and dozens of relative references that point to the stack as a *controller* that creates and manages all logical channels representing publish-subscribe topics used within the page. When loading the page, the browser would launch a single instance of such controller, and reuse it multiple times, requesting a channel from it every time it encounters a reference relative to this controller.

This pattern is hard to code in JS: controller is shared across scripts and pages, which may require explicit browser support.

E. Content and Infrastructure Layers

Assembling a downloaded page and numerous other client-side tasks require support from the runtime, the functionality of which is limited to only a small number of built-in options. For example, in Section II-C, we pointed out that fetching data and establishing client-service sessions is generally limited to HTTP; we proposed custom transports to go around this limitation (principles 15-16). The same pattern could be applied to other forms of client-side processing assisted by the runtime. For example, whenever an identifier in a web page needs to be resolved, one might wish to specify a custom name resolution component. Whenever some section of a web page needs to be parsed, one might wish to supply a custom parsing component that implements language extensions unsupported by the base standard, decrypts an obfuscated specification, calculates a fingerprint of a component specification to verify against a database of secure components, or performs custom type-checking.

Principle 19. *Any aspect of content processing assisted by the runtime infrastructure, including parsing, name resolution, decoding, and type-checking, should be possible to customize by recursively embedding a specification of the component that is supposed to replace the respective infrastructure service.*

While powerful, this idea raises security concerns: runtime is implicitly trusted not to contain malicious code and to correctly implement security protocols. If a component can override infrastructure services, such as naming and type-checking, it could potentially impersonate critical services, falsely claim to be secure, authorized, verified, or signed with a trusted key. To avoid this, one should be able to restrict the kinds of components that are permitted to override the runtime. Earlier, we postulated that components be typed, and their types support semantic annotations, so we propose that this determination be based on component types and occur as part of type-checking. In what follows, we just briefly sketch one possible approach.

The mechanism might work as follows: suppose the runtime loads a component A of type T_A as part of a web page, and the description of A has an embedded description of component B of type T_B that replaces an infrastructure service or performs

some verification on A; we can refer to B as A's *authenticator*. If the authenticator B is given, then before loading component A, the runtime first loads the authenticator B, passes to it A's description, and lets B authenticate it (for example, parse all semantic annotations in A to verify that they're legitimate). If B approves, the runtime tags the type of A with a certificate that a component of type T_B has vouched for it. One can think of type T_A of the component being verified in this manner as qualified with type T_B of its custom authenticator, $T_B :: T_A$.

Now, suppose that component C accepts A as a parameter. The definition of component C might request that its parameter be of type T_P; in such case, during composition, the runtime must only check that $T_A <: T_P$. Alternatively, the definition of C might state that its parameter has to be of type $T_V :: T_P$. The runtime would then have to verify not only that $T_A <: T_P$, but also that the description of component A has been checked and approved by an authenticator of the right type, $T_B <: T_V$.

Using this pattern, for example, a confidential collaboration component C might specify that it needs to be composed with a trusted communication channel; it might take such a channel as one of its template parameters. The trust could be specified via semantic annotations in the type of its parameter. C could further require that the truthfulness of these annotations (the fact that the channel is trusted) be verified by an authenticator of the appropriate type. The pattern could be used recursively.

The above presentation is necessarily terse; in-depth discussion of type-checking and security is beyond the scope of this paper. Analyzing the security implications of the composition pattern presented in this section is the subject of a future study.

III. Architecture

In this section, we summarize the core elements of our *live distributed objects* (LO) architecture, as an example realization of the principles listed in the preceding section. We encourage the reader to consult [14], [15], [16] for further details.

The previous section motivates an object-oriented approach, in which all visual and non-visual content, as well as elements of the underlying communication protocol stacks and runtime infrastructure, are treated uniformly as reusable components.

Accordingly, in our prototype platform [15], [16] we model any functionality accessible to a client as an abstract object – a *live distributed object* (LO). We apply this metaphor uniformly to web services, windows in which visual content is rendered, filters that transform data, distributed protocol instances, and even parts of our runtime infrastructure. Each LO is accessible via a *proxy*, a client-side component that exposes event-driven interfaces. The client-side logic is expressed as composition of such proxies into larger units; web applications are networks of interconnected proxies (Fig. 5). The client-side runtime assists with composition; it provides type-checking and reflection.

We mentioned that LO could represent UI elements, remote services, or protocol instances. Accordingly, our proxies could interact with the UI, but also make remote WS calls, or interact with proxies on other nodes in a peer-to-peer fashion. Irrespective of its type, a proxy interacts with other proxies exclusively by explicitly sending or receiving events through its *endpoints*.

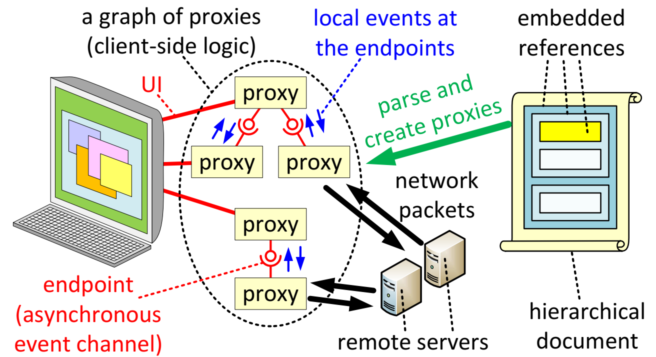

Fig. 5. References embedded in a web document are used to create a graph of interconnected proxies that can render UI, contact remote services, transform or filter data on the client, or participate on distributed peer-to-peer protocols.

Event-passing decouples proxies from one-another. It is assisted by the client-side runtime environment, which can generate wrappers to match interfaces. Each proxy can also expose any number of synchronous user-interface endpoints to bind to 2D or 3D displays alongside asynchronous event-based endpoints.

Proxy compositions are expressed in a platform-independent language, as XML strings we call LO *references*. A reference is a portable set of instructions for constructing a proxy for a given LO. It has a hierarchical structure and resembles a small web page. One can embed it in a larger document, store it in a file or in a communication channel, send it over the network, pass it as an argument or as a result of a WS call, and so on.

The concept of a reference generalizes, abstracts and unifies concepts such as WSDL [5] specifications and JS scripts. Each of these is, actually, a complete specification of an executable client-side logic. In case of WSDL, it is the logic of a service proxy, specified in a declarative fashion. In case of a JS script, code is given explicitly. In our platform, one can encapsulate WS proxy stubs and JS scripts as different classes of references and mash them up together with other types of components.

In contrast to static non-executable page elements decoupled from JS scripts that, in turn, lack state or visual representation, a reference in our platform produces a live and stateful proxy that can have any number of visual representations and exposes arbitrary event-based interfaces; it can have internal state and run in the background (using timer APIs). Our platform does not distinguish UI, non-UI, static or dynamic types of content; applications are composed entirely of interconnected proxies.

Composing references yields a larger reference. References are hierarchical documents similar to web pages in HTML and serve as the equivalents of these in our framework. Our "web pages" thus do not contain text or other static elements (unless the text is passed as a parameter to a proxy); they are simply hierarchical "recipes" for constructing proxies. When this sort of a document is loaded on the client machine, the client-side runtime parses the hierarchy of references, then uses embedded instructions to construct a hierarchical network of interconnected proxies. If the proxies expose any graphical endpoints, the document can be displayed; but it could equally well describe an application without user-interface (e.g., a network service).

The simplest type of proxy in our platform is one manually coded in .NET. Each such component has a 384-bit identifier that identifies the library in which it is stored, identifier within the library, and version numbers. The 384-bit identifier is the simplest type of a reference; if the client's runtime encounters one while parsing a document, it tries to locate the respective library (locally or from some remote repository), load it into the process, and instantiate the class that implements the proxy. The runtime uses reflection to analyze the structure of the class and custom .NET attributes, and infers the list of all endpoints, incoming and outgoing events and their types. It also extracts semantic annotations from .NET attributes, and appends them to type metadata. The platform maintains its own type system, independent of .NET, that describes all types and components found in the libraries it has dynamically loaded. As postulated earlier, these types are determined only by the logical structure of interfaces and custom semantic annotations, not by physical .NET classes used to implement them. When comparing two component types, the platform relies on its own type metadata. Our system uses the mechanism just described also to analyze itself and bootstrap all of the predefined types and components.

When creating a proxy of a library component, the platform invokes the constructor to create the .NET object implementing it, and passes to it a runtime context, which includes scheduler, clock, timer, network, and filesystem interfaces, as well as the interfaces for creating endpoints and proxies, reflection, etc.

In its constructor, the proxy object constructs all its UI and non-UI endpoints and exposes them as .NET properties. From this moment, endpoints are the only way the proxy can interact with other parts of the application. The most common type of endpoint is a bidirectional interface: the proxy exposes method calls through an *incoming* interface, and requests an *outgoing* interface through which it can calls methods of other proxies. Endpoints are meant to be connected into pairs: the only operation one can perform on an endpoint is to connect it to another endpoint of a matching type. When this happens, proxies bind to one-another's interfaces. Each endpoint exposed by a single proxy can only be connected to one other endpoint at any time.

The proxies composed at runtime can be defined in different .NET libraries, but as mentioned earlier, we define types based on the logical structure of interfaces, not binary compatibility. To connect two proxies, it may not be enough to just exchange their .NET interfaces. In these cases, the platform generates wrappers. The proxy that would make method calls receives a reference to the *frontend* .NET object that transforms method calls into a standardized representation, and passes them down to a *backend* .NET object, which then unpacks and directs all calls to the target proxy (Fig. 6). When loading a .NET library, the platform dynamically generates frontend and backend code for all components, compiles and dynamically loads it into the process for future use. The overhead of doing this is negligible.

The platform hosts all proxies in the same process and app domain, and method calls are passed directly between proxies (or through frontend and backend). If a component is annotated accordingly, the platform can automatically generate a wrapper that places all calls into the proxy on the proxy's private lock-free queue, and process them all asynchronously at a later time in batched mode (Fig. 6). Just like data-conversion wrappers, nonblocking queues and scheduling code are generated by our system automatically when it bootstraps itself, for all proxies.

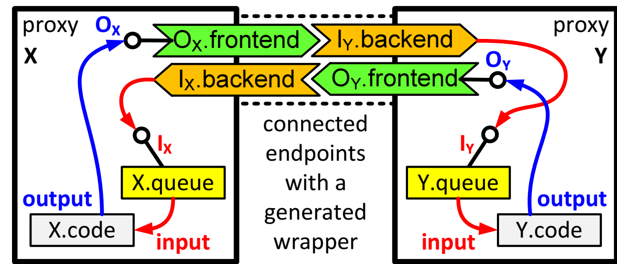

Fig. 6. A pair of connected proxies that interact via automatically generated wrappers and queues. The system auto-generates wrappers at bootstrap time.

As mentioned earlier, all interactions of a proxy are tunneled via its endpoints. To obtain an endpoint from a proxy, one must have previously created it from a reference. A container proxy can thus easily obtain endpoints from the proxies of embedded objects it created, and connect to them to interact; on the other hand, proxies of embedded objects can't just randomly interact with one-another. Interactions in our system are only possible for proxies related in the hierarchy, as we've postulated earlier.

Library components can be parameterized; in such case, the .NET object may be a generic class, and it may receive values or references to other objects, in its constructor. Templates of this sort can be used to create complex hierarchical references. If used in a document, a reference to such template component has all of its parameters embedded; these can include, in particular, embedded references to components it uses internally. This enables many of the patterns discussed in section II; e.g., to separate UI from data retrieval logic, we model UI and data retrieval as separate objects and implement them as separate proxies in a running system; the reference to the data retrieval object is passed as a parameter to the view rendering object.

Our XML language of LO references supports other types of compositions, such as factories, folders, and the controller-controlled pattern discussed earlier. We omit details for brevity.

When passing an LO reference as a parameter, the runtime dynamically type-checks to see if the argument type matches the type of the formal parameter of a template. This involves comparing the logical structures of interfaces and the semantic annotations. Annotations are parsed by pluggable, user-defined modules. These modules can also be defined in .NET libraries. After parsing, compiled annotations are stored as part of type metadata. When comparing types, annotations compiled by the same module are compared pairwise by invoking the module's type-checking interface. By default, the runtime doesn't verify the truthfulness of these annotations (it only compares them).

In a longer perspective, we hope to use semantic annotations to characterize the event patterns coming in or out of an object [15], but they can be used for any purpose, e.g., to mark objects as secure or reliable, attach signatures, fingerprints, and access permissions that could be compared as a part of type-checking.

To customize the process of type-checking, we allow an LO reference X to contain an embedded reference C marked as its *authenticator*. If the latter is present when constructing a proxy from X, the runtime first creates a proxy of the authenticator C, connects to its endpoint, and passes X's reference to C to let the authenticator decide whether the reference is legitimate, by parsing semantic annotations or some subset it cares about. If this succeeds, X's reference has a certificate attached to its type stating that an object of a certain type has vouched for X's integrity. A template object can request that its parameter have certificate issued by an authenticator of a particular type.

Besides compositions defined statically within the structure of the references, the platform supports dynamic compositions. A proxy may dynamically download serialized references from communication channels and other remote sources, deserialize, use them to create proxies, and then connect to their endpoints. We use this dynamic reflection capability to implement shared folders, shared desktops, and other sorts of container patterns.

One could also use dynamic reflection to seamlessly migrate between server-side and client-side execution. Initially, the proxy constructed from a reference might be just a WS proxy stub that performs no local processing at the client and directs all requests to a server, but if needed, can dynamically fetch a reference of another proxy that can do client-side processing, instantiate it internally, and then instead of issuing requests to a remote server, send them to this internally maintained proxy. One example of how this could be useful would be for a group of clients to seamlessly shift between updating their state at a centralized web service, and a peer-to-peer mode, where state is downloaded locally and replicated on all clients, which then send updates directly to one-another over a multicast channel.

IV. Related Work

Our work has been inspired by a rich body of prior research on typed component integration platforms, including OLE [2] and Jini [18], protocol composition frameworks, such as BAST [7], and web-like P2P environments, e.g., Croquet [17]; a more comprehensive discussion of these can be found in [14].

With respect to language-specific technologies such as OLE and Jini, our architecture is more interoperable. For example, principles 2, 3, and 4 were not needed for the sorts of applications for which OLE or Jini were designed. Principle 2 appears to be quite unique to the style of composition we advocated in this paper, and its influences are seen throughout our architecture (e.g., in Principle 5). The corresponding mechanisms, such as automatic generation of wrappers for binary-incompatible code or lock-free queues to resolve deadlocks, are not present in any of the major typed component-integfration platforms, such as OLE/COM, Jini, Java, or .NET. With respect to Jini, we have inherited the concept that *everything is an object*, but in our live distributed objects model we took this concept even further; we proposed that even type-checking be customizable, thus extending the object abstraction to parts of the runtime.

In comparison to server-side composition technologies such as BPEL [9], our client-side components are smaller and more fine-grained; principles such as 5, 6, and 7 do not apply to web services. Composition patterns also differ significantly, and the requirement to support UI has peculiar consequences, such as principles 11 or 14. Work on integrating web services with Jini (e.g., [8]) or peer-to-peer technologies such as JXTA (e.g., [1]) has also been focusing almost exclusively on the service-side.

In designing particular aspects of our system we have been inspired by a great many other technologies; in particular languages such as XAML [11], work on structural subtyping [12], event-driven architectures such as SEDA [19] and many others.

V. Conclusions

Client-side composition is a core part of the WS-* architecture, but is inadequately supported by the existing technologies. We have built a platform that supports major composition patterns and offers flexibility unseen in existing platforms. Our work demonstrates that client-side composition is feasible and practical. The technical guidelines we proposed may serve as a basis for extending existing WS-* standards and technologies. The live distributed objects platform is available for free [16].

Acknowledgements

Our work has been supported in part by grants from AFRL, AFOSR, NSF, and Intel Corporation. We would like to thank Daniel Freedman for his comments.

References

[1] F. Banaei-kashani, C. chien Chen, and C. Shahabi, "WSPDS: Web Services peer-to-peer discovery service," *ISWS*, 2004.
[2] K. Brockschmidt, *Inside OLE*. Microsoft Press, 1995.
[3] J. Cardoso, "Semantic integration of web services and peer-to-peer networks to achieve fault-tolerance," *GrC*, 2006.
[4] J. Cardoso and A. Sheth, *Semantic Web Services, Processes and Applications (Semantic Web and Beyond: Computing for Human Experience)*. Springer-Verlag New York, 2006.
[5] E. Christensen *et al.*, "WSDL1.1," *http://www.w3.org/TR/wsdl*.
[6] S. Douglas, E. Tanin, A. Harwood, and S. Karunasekera, "Enabling Massively Multi-Player Online Gaming Applications on a P2P Architecture," *ICIA*, 2005.
[7] B. Garbinato and R. Guerraoui, "Flexible proto-col composition in bast," *ICDCS*, 1998.
[8] Y. Huang and D. Walker, "Extensions to Web Service techniques for integrating Jini into a Service-Oriented Architecture for the Grid," *ICCS*, 2003.
[9] IBM *et al.*, "Business Process Execution Language for Web Services (BPEL)," *http://ibm.com/developerworks/*, 2007.
[10] B. Lin, Q. Li, and N. Gu, "A semantic specification framework for analyzing functional composability of autonomous web services," *ICWS*, 2007.
[11] L. MacVittie, *XAML in a Nutshell*. O'Reilly Media, 2006.
[12] K. Ostermann, "Nominal and structural subtyping in component-based programming," *JOT 7(1), Jan-Feb 2007*.
[13] K. Ostrowski, K. Birman, and D. Dolev, "Extensible Architecture for High-Performance, Scalable, Reliable Publish-Subscribe Eventing and Notification," *JWSR*, Oct-Dec 2007.
[14] K. Ostrowski, "Live Distributed Objects," Ph.D. Dissertation, Cornell University, 2008, http://hdl.handle.net/1813/10881.
[15] K. Ostrowski, K. Birman, D. Dolev, and J. H. Ahnn, "Programming with Live Distributed Objects," *ECOOP*, 2008.
[16] K. Ostrowski *et al.*, "Live Distributed Objects (the project's website at Cornell)," *http://liveobjects.cs.cornell.edu/*, 2008.
[17] D. Smith, A. Kay, A. Raab, and D. Reed, "Croquet: A Collaboration System Architecture," *C5*, 2003.
[18] J. Waldo, "The jini architecture for network-centric computing," *CACM 42, 7 (Jul. 1999), pp. 76-82.*, 1999.
[19] M. Welsh, D. Culler, and E. Brewer, "Seda: architecture for well-conditioned, scalable internet services," *SOSP*, 2001.