Object Collaboration in the Orca Web Framework

Lauritz Thamsen

Hasso-Plattner-Institut
Prof. Dr. Robert Hirschfeld, Robert Krahn, Michael Perscheid

Abstract. Browsers and servers execute web applications cooperatively. Application parts exchange information through the Hypertext Transfer Protocol, which is unidirectional by design. Requests from servers to clients are not part of the protocol. Polling mechanisms address this, but only allow event notifications and no responses from clients to servers. Furthermore, HTTP packages consist of name-value-pairs that contain object representations and address functionality through URLs. While these abstractions allow communication between heterogenous hosts, they are unnecessary general when applied in homogenous environments and tend to scatter implementations.
As part of our Orca web framework we provide transparent message passing. Developers implement complete applications including collaboration in a single object-oriented language. Objects invoke methods of objects through messages and the framework forwards all messages to remote objects transparently. Developers neither create nor serialize manually and use remote objects in their programs as if they were local. Remote invocations are transparent and on the abstraction level of object-oriented programming. We compare Orca's message-based collaboration approach to direct usage of HTTP and show that collaboration can be implemented less tangled and less scattered with Orca.

Keywords: Web Applications, Distributed Objects, Remote Messaging, Orca

1 Introduction

From its beginnings in the 1990s, the Web has become an important platform for real applications in less then one and a half decades. Today, browsers and servers run web applications and manipulate the user interface dynamically. The execution corresponds to the client-server-model. That is, several distributed clients actively use a single passive server. Necessary client-server communication is based on the unidirectional Hypertext Transfer Protocol [8] (HTTP) that allows requests from a client to the server, whereas the server only responds to requests and cannot send own requests to clients.

However, collaborative web applications require bidirectional communication to allow responsiveness [18]. For instance, bidirectional communication enables chat application servers to propagate chat messages instantly to clients. Developers rely on polling mechanisms that allow event notifications but no responses from client to server.

Moreover, the communication via HTTP requests and responses is not on the abstraction level of application programming. Requests address services by arbitrarily chosen URLs and transfer information as name-value-pairs. In particular, the URLs and name-value-pairs can be selected without respect to existing programming interfaces. Furthermore, developers apply functionality that serializes objects for transmission explicitly. For these reasons, implementation details are not hidden by well-defined interfaces but openly exposed [19]. In fact, name-value-pairs, serialization formats, and URL addresses define interfaces on top of HTTP that are independent of object interfaces. The implementation against such interfaces crosscuts the implementation of actual application logic and collaboration is not transparent to developers.

Various web frameworks and applications address these issues by conforming to the Representational State Transfer (REST) [9] architecture or applying protocols like the Simple Object Access Protocol (SOAP) [13] for remote invocations over HTTP. The RESTful architecture emphasizes uniform URL interfaces analogous to standard database operations (*CRUD*), but does not incorporate conventions for other actions or transmission formats. SOAP provides arbitrary invocation mechanisms in heterogenous web environments through standardized types and message formats. However, SOAP is only a message encoding format and does not define invocations.

To bridge the abstraction gap between object-oriented programming and HTTP communication, we implemented transparent message passing as in Distributed Smalltalk [1] in our Orca framework [26]. Orca allows to develop client- and server-resident parts of web applications inside the Squeak environment [15] by generating JavaScript [7] sources from Smalltalk [12] code. The message passing allows to send Smalltalk messages across distribution boundaries. It builds upon a bidirectional communication layer that abstracts from HTTP. Client and server can send synchronous messages that pass control and return answers. Alternatively, oneway message sends do not block sending processes.

Remote message sends are not restricted to specific interfaces or types, but transmit arguments and return values either directly by value or indirectly by reference. Therefore, Orca serializes objects automatically or creates local proxies that encapsulate references to remote objects. Remote messaging happens transparently to developers. We argue that this approach increases understandability of distributed web applications compared to using requests and responses directly.

We evaluated our approach in a case study that compares Orca to a solution that applies HTTP directly. The comparison shows that Orca allows to implement the complete application on the level of abstraction of object-oriented programming. With Orca, the implementation is less scattered, less tangled and control flow can be followed easier.

The remainder of this thesis is organized as follows. Section 2 gives a short overview of the architecture of web applications, explains the browser's need to communicate to a server and demonstrates the challenges of using HTTP directly. Section 3 introduces our message passing approach, while Section 4 describes its implementation. Section 5 evaluates this approach and compares it to JavaScript's direct usage of HTTP. Section 6 presents related work and Section 7 concludes this thesis.

2 The Architecture of Web Applications

A web application is an application that is displayed by a browser. Browsers request documents or programs from servers. Documents are either static or dynamically generated by the server and expressed in markup. Browsers can execute JavaScript programs directly and may use proprietary plug-ins for other languages. Since JavaScript programs are executed from source and browsers use markup as graphic primitives, web applications can be independent of a user's computing environment. They are accessible without installation to millions of users. Also, since reloading pages effectively redeploys web applications, it is possible to ship small releases immediately.

Browsers are necessary for responsive user interfaces, but are restricted to security policies. Thus, web applications require a web server for certain tasks and, therefore, are divided into at least two parts. When partitioned into client- and server-resident parts, web applications rely on HTTP for communication.

2.1 Distribution of Web Applications

Partitioning of web applications between browsers and servers is necessary. A server is required for tasks that need access to local hardware, other programs or foreign servers, whereas programs on the client-side have access to the browser's user interface.

Browser execute JavaScript in a restricted environment since JavaScript sources are loaded from potentially untrusted hosts [10]. Programs are only allowed to request information from the original server (*Same-origin policy*). Thus, a server is needed to request information from other servers and to propagate data among clients. Additionally, programs are not allowed to access the local file system or to invoke other programs (*Sandbox policy*). Therefore, the server needs to store information in a database for later retrieval. Although client-side storage inside a browser becomes increasingly available [16], it is not a sufficient replacement since users potentially use web applications with multiple devices. Also, centrally stored data alleviates sharing among clients.

All major browser execute JavaScript single-threaded at the time of writing. Servers, however, can run programs in parallel on many cores and optimized for the actual hardware.

Browsers execute JavaScript from source code and, therefore, these sources are accessible. Since code obfuscation does not guarantee concealment, servers are used to protect intellectual property.

Without installation of additional plug-ins, browsers only execute JavaScript programs. However, different programming tasks might benefit from different programming paradigms and languages. The server can execute programs written in any programming language.

While administrators control the environment of the server, a user controls the environment on the client-side. Therefore, programs that rely on the correctness of certain environment values are best executed on the server-side.

We see that several tasks must or should be implemented on the server-side, but browsers are at least necessary as display of web applications. They render documents according to style declarations. The output can either be adapted by exchanging complete documents and style sheets or programmatically with JavaScript. Interface manipulation with JavaScript allows more responsive user interfaces compared to reloading. Therefore, JavaScript programs should be used to implement interface manipulation.

Furthermore, there is a trend towards "Fat Clients" [16,18] for several reasons including responsiveness and robustness. Client-side applications can potentially finish execution of tasks in less than the network's round-trip time and do not need to consider the reliability of network connections.

Browsers should run most operations, but certain tasks are best executed by servers. Thus, client and server need to execute web applications cooperatively.

2.2 Communication in Web Applications

Distributed parts of web applications exchange information using HTTP requests. Such requests can transfer arbitrary information including binary data, but usually requests transfer data between heterogenous environments that need different object representations. Thus, application data stored in objects is serialized for transmission. Such serialization translates objects typically into Extensible Markup Language (XML), Hypertext Markup Language (HTML) or JavaScript Object Notation (JSON). Libraries for serialization are available for most programming languages.

Usually, developers implement the following steps when interaction between client-resident programs and server-resident programs is necessary:

1. On the client: Serialization of JavaScript objects and creation of a HTTP request.
2. On the server: Dispatch of the HTTP request according to its target URL and structure.
3. On the server: Deserialization of the HTTP request content and reaction on the request.
4. On the server: Serialization of the result of an operation and preparation of the HTTP response.
5. On the client: Deserialization of the HTTP response content.

Client-resident parts of a web application serialize data, create a request, wait for the response, and deserialize the response content. The server performs all other steps. This renders understanding modules on either side without knowing the other side difficult.

The following example shows a JavaScript program (Listing 1.1) from a tutorial for jQuery[1], a popular JavaScript library that aims on productivity and provides an alternative interface to the request object of browsers. In the example, users rate content and this triggers a request to the server.

```
// send request
$.post("rate.php", {rating: $(this).html()}, function(xml) {
  // format and output result
  $("#rating").html(
    "Thanks for rating, current average: " +
    $("average", xml).text() +
    ", number of votes: " +
    $("count", xml).text()
  );
});
```

Listing 1.1: A user's click triggers a request.

The example JavaScript code creates a request to the server through calling a function. The first argument to this function is the request's address. In this case the request is addressed to a PHP script. The second argument is the request's content. The called `html()` function returns HTML markup that is used in a name-value-pair. The third argument is a callback, which will be asynchronously triggered as soon as the server responded.

This example does not include the server's request handling, but request processing and response creation must happen. Often, several different requests

[1] `http://docs.jquery.com/Tutorials:Getting_Started_with_jQuery`, accessed June, 15, 2011

are addressed to a single script of the server. Such a script dispatches requests to methods. It can be difficult to understand client-server communication without knowing the dispatch mechanism.

When the request has been dispatched, the server will retrieve the received markup from the request by name. Since the server calculates an average value, the markup must be deserialized and converted to a number.

The callback function receives an XML document, which was created by the server to transfer average and count. The client-side function retrieves values from two specific XML nodes and displays them through document manipulation.

Both communication partners use different serialization formats and different name-value-pairs. Thus, interaction basically happens through an ad-hoc protocol on top of HTTP (*"rating"*-markup to the server; XML document with *"average"* and *"count"* to the client) that is completely independent of already existing object interfaces. The protocol must be reflected on both sides and adaptions on one side must be reproduced on the other side.

Besides programming against such an interface, developers are restricted to the protocol's request-response-cycle. Each interaction starts on the client-side. Notifications from the server to a client are not part of the protocol. However, certain applications benefit from the possibility to update clients instantly. For this reason, developers apply various polling mechanisms, which are generally referred to as Comet[2]. Implementations of Comet allow the server to push data spontaneously to the client.

A common implementation strategy is to use an *iFrame* of infinite length that keeps loading and, hence, can receive events at any time. Another implementation is called *long polling*. The client always keeps a asynchronous request open to allow the server to respond at any time.

While Comet implementations are available, they usually implement only event notification. Such events can trigger further requests that may contain answers to the server. However, such answering is not provided by available implementations.

Additionally, JavaScript execution is single-threaded in all major browsers at the time of writing. Thus, when a client requested information synchronously, the thread is blocked and the server cannot receive answers to sends through the asynchronous request.

When client-server communication is implemented as presented in this section, developers have to define ad-hoc interfaces that are independent of ob-

[2] `http://alex.dojotoolkit.org/?p=545`, accessed June, 29, 2011

ject interfaces, need to handle serialization manually, and are restricted to the request-response-cycle. In contrast, Orca provides transparent message sending that handles serialization automatically, works bidirectionally and lets developers program against existing object interfaces.

3 Object Collaboration in Orca

The HTTP protocol and serialization allow information exchange between heterogenous hosts. Applications serialize their data to intermediate representations and store them in name-value-pairs of packages. The serialization and names of such pairs represent ad-hoc abstractions and crosscut implementations. This abstraction mismatch is especially disconcerting when client and server run similar programming environments.

This is the case with our Orca Web framework. Orca allows to develop client- and server-resident parts of a web application in the interactive Squeak environment and the object-oriented Smalltalk language. The server runs Smalltalk code in Squeak's virtual machine and generates necessary client-side JavaScript code for execution in a user's browser. It compiles application code and specified classes of the standard library [28]. Generated JavaScript code maintains Smalltalk semantics [27] and Orca provides a complete Smalltalk environment in JavaScript: a Smalltalk-like class system, implementations for necessary virtual machine primitives, global singletons, and central language features, such as non-local returns from block closures. As a result, browser and server are not only expressed in the same language but effectively run in equal environments.

Inside each of those environments, objects interact through message sends. Messages pass arguments and invoke methods. We want client-server communication to be expressed on the same level of abstraction. Thus, we implemented transparent remote messaging and, as a consequence, messages invoke local and remote methods. When messages address remote objects, Orca forwards them transparently. The framework handles the necessary serialization and request generation automatically.

However, messages require a local receiver. Thus, Orca provides stand-ins for objects that are not part of the local address space. Such proxies can be created explicitly by certain factory methods. On the client, sending the `asRemote` message to a class returns a proxy of that class. Since there are potentially multiple clients, the server-side equivalence requires a session object to create a proxy for a class of a client. Client-resident methods can create proxies for all classes, whereas proxy generation on the server is restricted to classes available on clients. Each message send to such a proxy is forwarded to the remote class.

Smalltalk messages pass control between methods. A process stops execution of one method to execute another method. When a process finishes execution of a method, it resumes the calling context. Thus, message sends in Smalltalk are synchronous and block the execution of a method. In Orca, transparent remote message sends are synchronous as well. Remote messages block the sending process until an answer arrives. That is, besides looking like local messages, remote messages behave as if they were local, too.

Listing 1.2 shows a client-side example. A proxy of a class is obtained by sending the `asRemote` message and assigned to a temporary variable. While the variable is local, it holds a proxy that references the remote class. All messages sent to the proxy are forwarded and return a value. Thus, the instantiation happens on the server and returns a proxy for the newly created instance to the client.

```
Client>>#createRemoteInstance

    | remoteClass remoteInstance |
    remoteClass := MyClass asRemote.
    remoteInstance := remoteClass new.
    ↑ remoteInstance
```

Listing 1.2: A client-side method creates a proxy.

Smalltalk messages carry the arguments that are supplied as parameters to methods. So, remote messages need to transfer arguments to invoke certain methods. Orca's remote messaging transfers numbers, strings, characters, booleans and the pseudo-variable `nil` by value. The default option for instances of other classes is to transfer a reference instead of the actual object. That is, a proxy is generated transparently on such message sends and supplied as parameter. These rules apply for arguments and return values. The example method in Listing 1.2 sends a synchronous remote message to a proxy of a class. The class returns an instance on the server-side and Orca creates a proxy for it on the client-side.

Although Orca generates proxies for general collections, it treats arrays differently. Arrays are send as copies with each contained element passed either as proxy or copy. Developers need to be aware of this feature and can decide whether a proxy or a copy would fit their needs best. Transferring a proxy for a collection instead of an array of proxies can impact performance.

Creating proxies for remote objects alleviates sharing of particular functionality and state. Since the object is not replicated but in a single place, it is not necessary to synchronize the state of replicas. The types we transfer directly are either immutable or, in the case of strings and arrays, Orca considers them to be immutable. Nevertheless, communication happens in a distributed system and potentially in parallel. For example, when multiple clients manipulate a single object through proxies, one client can manipulate instance state while another client is receiving the old value. Therefore, developers need to be aware of concurrency issues.

Certain situations benefit from supplying copies instead proxies. For instance, when an object is created by a client, but should be accessible permanently on the server, a proxy would be insufficient since clients may leave the web application's page. Additionally, when a client sends an object by reference to the server and the server propagates this object to other clients, clients would receive a proxy of a proxy. When clients use such proxy proxies, two remote sends are necessary for each access to the object's state. Since remote sends are significantly slower than local sends, it improves performance to copy certain objects. Orca lets developers define which objects should be send by value and which should result in proxies. Instances of classes that return true on `copyOnSend` messages are always passed as values. When an object is passed as copy, the object's class has to be available in the target environment. In Orca, the client has access to all classes that have been translated and transferred to the client. This is usually a subset of all classes available on the server. Besides, as mentioned in Section 2, certain operations can only be performed in particular environments. Furthermore, the instance is recreated in the other environment and, thus, bound to the local class object. That is, when the instance accesses class variables the look-up directs to the local class.

As we have shown, arguments and return values of remote message sends are not restricted to specific types. Remote sends appear similar to local message sends. However, local message sends move a single process from executing one method to executing another one, whereas remote message sends involve two processes. A process that sends a remote message is suspended until it receives an answer. Since the two involved processes are potentially executed on different hosts, remote message sends are orders of magnitude slower than local invocations. The sending process waits for at least the roundtrip time of the network.

The impact of network latency can be reduced by not waiting for return values. Orca provides *oneway message sends* that do not suspend the sending process. Such non-blocking sends return directly, but without meaningful result. The interface to send oneway messages is similar to Smalltalk's interface to explicitly perform messages. An example method is shown in Listing 1.3. The client-side method creates a proxy for the server's database singleton and sends a oneway message to trigger a backup.

```
Client>>#backupTheDatabase

    | remoteDatabase |
    remoteDatabase := Database asRemote uniqueInstance.
    remoteDatabase performForked: #backup.
```

Listing 1.3: A client-side method sends a oneway message to the server's database.

Another example shows how oneway messages and proxies can be combined to provide asynchronous messaging with callbacks. A client sends a oneway message containing a query and a callback block to the remote server (Listing 1.4). Smalltalk blocks are anonymous functions and closures. That is, a block is stored with referenced environment and can be executed later. The server-side method gets invoked with a string value and a proxy for the block (Listing 1.5). When the server's database system finishes the query, it sends the result to the remote block without suspending the process.

```
Client>>#executeQueryAndShowResultFor: aQuery

    | callBack |
    callBack := [ :result | self display: result ].
    self server
        performForked: #executeQuery:thenCall:
        with: aQuery
        with: callBack.
```

Listing 1.4: A client-side method invokes a remote method and supplies a query and a callback block.

```
Server>>#executeQuery: aQuery thenCall: aRemoteBlock

    | result |
    result := self executeQuery: aQuery.
    aRemoteBlock performForked: #value: with: result.
```

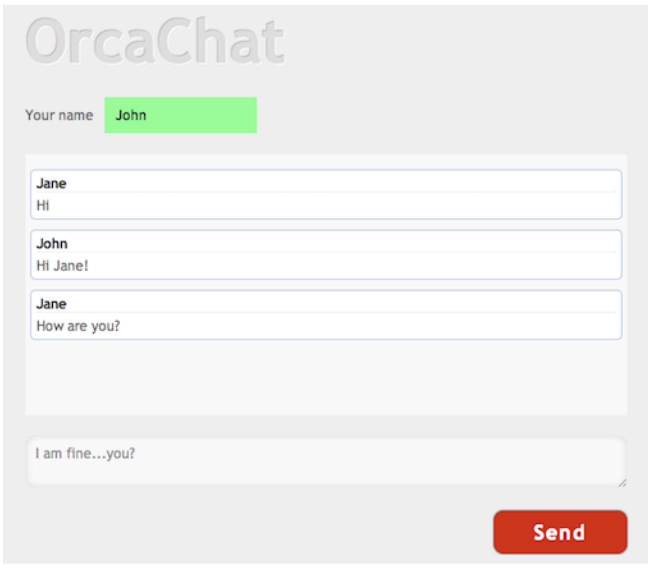
Listing 1.5: A server-side method executes a database query and then invokes a callback block.

Effectively, developers partition the application between client and server by instantiation. Objects can either be created locally or remotely and, thereby, developers decide where objects reside and execute their methods. That is, applications are partitioned on the level of abstraction of object-oriented programming.

Remote message sends may result in network-related exceptions. For example, when a user leaves the web application's page while a server sends a remote message, a `ClientTimedOut` exception is thrown and must be handled in server-side code.

A chat application, as depicted in Figure 1, shows proxies and remote messaging in a meaningful context. On the client-side, a `ChatWindow` displays all chat notes and provides input fields for a name and a note. On the server-side, a `ChatHub` propagates chat notes between registered clients. All `ChatNode` instances are transferred as copies between client and server (Listing 1.6). On initialization, the client creates a proxy for the server's singleton and registers itself through a oneway message that adds a proxy to the server's collection of chat participants (Listing 1.7). A button click triggers another oneway message that transmits a chat note to the server (Listing 1.8). This message invokes a spreading method of the server that distributes the chat note among all registered clients (Listing 1.9). Since the server iterates over all proxies without knowing which clients are still reachable, we catch the mentioned timeout exception. That is, if a client is no longer using the web application, the exception is caught without breaking the iteration.

In this example, synchronous remote message sends are used to obtain proxies, whereas oneway messages are used to spread copies of objects.



John

Jane
Hi

John
Hi Jane!

Jane
How are you?

I am fine...you?

Fig. 1: Users share messages with other users in a chat application.

```
ChatNote class>>#copyOnSend

    ↑ true
```

Listing 1.6: Chat notes should be passed by value.

```
ChatWindow>>#initialize

    self remoteHub: ChatHub asRemote uniqueInstance.
    self remoteHub performForked: #registerWindow: with: self.
```

Listing 1.7: Client-side initialization connects client and server.

```
ChatWindow>>#sendChatNote

    | chatNote|
    chatNote := ChatNote
        text: messageInput text
        author: nameInput text.
    self remoteHub
        performForked: #spread:
        with: chatNote
```

Listing 1.8: The client sends chat messages to the server.

```
ChatHub>>#spread: aChatNote

    self registeredChatWindows do:
        [ :each |
            [each
                performForked: #displayChatNote:
                with: aChatNote]
            on: ClientTimedOut do: ["nothing"] ].
```

Listing 1.9: The server spreads chat messages among clients.

The chat application is an example for situations in which the server sends messages to clients. To allow this, we implemented Orca's remote messaging on top of a bidirectional HTTP abstraction.

4 Implementation

The Orca framework provides message-based communication between distributed application parts as in Distributed Smalltalk. The approach is implemented on top of a communication layer that abstracts from HTTP and provides bidirectional communication between client and server. Moreover, this section explains in detail how messages are forwarded and return values answered.

The implementation is limited by the implementation of Squeak's virtual machine. Additionally, Orca's remote messaging does not provide distributed garbage collection, yet. These limitations are presented in more detail at the end of this section.

4.1 Layered Communication Infrastructure

Orca's communication infrastructure consists of layers as shown in Figure 2. Client- and server-resident objects communicate through messages, which have to be forwarded if addressed to remote objects. The message forwarding builds upon a bidirectional communication layer that abstracts HTTP's unidirectional request-response mechanism. Of course, message sends map down to requests and responses, as HTTP still is the underlying communication protocol.

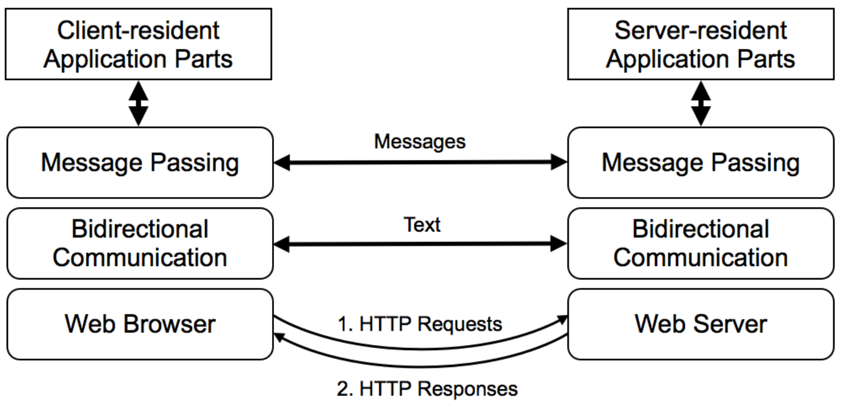

Client-resident Application Parts
Server-resident Application Parts
Message Passing
Messages
Message Passing
Bidirectional Communication
Text
Bidirectional Communication
Web Browser
1. HTTP Requests
Web Server
2. HTTP Responses

Fig. 2: Application parts communicate through messages, which transparently result in requests and responses.

4.2 Bidirectional HTTP Abstraction

The server propagates updates to clients through an implementation of Comet, *long polling*. The client issues an asynchronous request to the server and the server responds whenever there is a message for that client. When the server responds to this asynchronous request, the client issues another asynchronous request. However, for a short time, the server has no request to respond to, but might attempt to send further messages to client-resident objects. Therefore, each client is represented by a session object on the server and each session maintains a message queue.

Our long-polling implementation of Comet also allows answers from clients to the server. We use following requests to send return values. Message sends that return values are synchronous and, hence, block the execution of the sender until an answer is responded. Thus, since JavaScript execution is single-threaded in all major browsers, a synchronous message send blocks the execution of all client-resident application parts. When the server uses the asynchronous long-polling request to send a message to the client in this state, the client cannot respond. For example, a message send from a client can cause the server to send a message back to the client before answering the request. Such nested interactions must be possible for programs. For instance, the Double Dispatch idiom would result in nested communication. Therefore, the server uses the client's synchronous request instead of the asynchronous long-polling request to send its message. Figure 3 shows such a sequence of requests and responses.

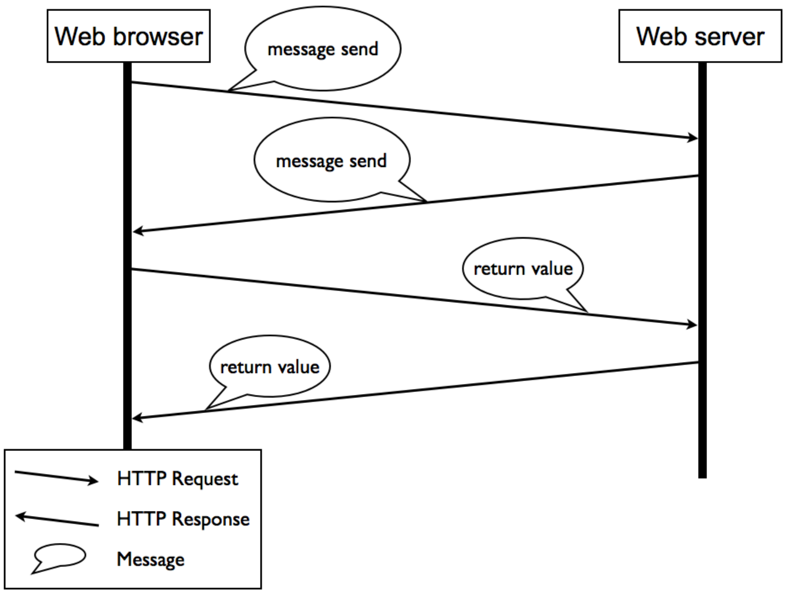


Fig. 3: The server uses a synchronous request to send a message, receives an answer and sends the answer to the original request.

Besides, this approach limits the number of open requests per client. At any time, a client waits for maximal two responses. This is important since some browsers are restricted to a certain maximum of open requests.

We use HTTP's request and response packages, but without regard to their original intent. Therefore, we mark packages as *synchronous sends*, *forked sends* and *answers*. A synchronous send is a HTTP package that contains a Smalltalk message and the sender waits on an answer. Synchronous sends must eventually be responded to with packages that contain return values. Such packages are marked as answers. A forked send is a HTTP package that contains a Smalltalk message that should only be performed. Forked sends can potentially be send through asynchronous requests.

When a client does a synchronous send, it waits for an answer and that answer needs to be returned into the correct execution context. In JavaScript, responses to synchronous requests are returned on top of the execution stack that issued the request, whereas responses to asynchronous requests create a new stack that starts with the supplied callback function. Although, the callback function is a closure and, hence, binds all referenced objects as environment for later execution, using an asynchronous request would not stop execution until an answer is received. Since there are no concurrency primitives in JavaScript [10], all synchronous sends must use synchronous requests to return into the correct execution context. Therefore, if the server has access to a synchronous and to the asynchronous long-polling request, it must always respond to the synchronous request.

As mentioned, the Double Dispatch idiom causes nested requesting. When such nested requesting was triggered on the client-side, the client eventually waits for multiple answers since the response to the first request from the client to the server contained a synchronous message instead of an answer. When the client receives this first request, it performs a message send on top of the context's execution stack. As a result, the execution stack of a client that waits on multiple answers contains multiple execution contexts that expect answers in *last-in-first-out* order. When the client finishes performing a message send, it transfers the return value through a synchronous send in order to maintain the remaining stack. Thus, when the answer package to the initial synchronous send is received, the answered value can be returned into the correct execution context.

On the server-side, the mentioned message queue stores synchronous and asynchronous message sends. Each synchronous message is associated with a *promise*, which is a semaphore that waits on a value [11]. Such a promise blocks the execution of a process until an answer is received. Whenever the server sends a message that is associated with a promise, it pushes the promise on a stack of the session. All promises on that stack represent processes that sent an synchronous message to the client and currently wait for an answer. Since JavaScript execution is single-threaded, an answer from this session's client always belongs to the top-most promise. Resolving a promise with an answer resumes the waiting server process.

HTTP does not provide connections. That is, no connection is closed when a user leaves the web application's page. The server still retains the asynchronous long-polling request and a session for that client. We use a timeout to discard the request and remove the reference to the session object eventually. When a client issues the asynchronous long-polling request to the server for the first time, the server creates a session object and starts a watchdog process for this session. The client's asynchronous long-polling request is responded with a timeout warning when the server has not used this request for sends in a certain timespan. The watchdog process checks periodically whether there has been a new request from the client for a specific longer timespan. If there was no activity and the client therefore did not issue another asynchronous request as reaction to the timeout warning, the server discards the session. If at that point, a process still waits on answers to message sends, that process is resumed and a `ClientTimedOut` exception is thrown. Such timeouts occur only when clients are no longer reachable. Actual remote execution is not restricted by a timeout. Nevertheless, developers using Orca's remote messaging have to expect timeout exceptions and must define appropriate behavior.

We would prefer to use explicit disconnect events to provide finalization hooks for developers. This problem is known and HTTP connection-events are discussed in research [16].

The interface to this bidirectional communication layer allows to send string messages either synchronously through a `send:` method or without blocking through a `sendForked:` method. Developers can setup named message handlers and string message sends must be addressed to a specific message handler.

For example, we implemented a remote execution interface based on Smalltalk blocks that uses the interface. Blocks that receive `onServer` or `inSession:` as messages are translated differently by Orca and generated code uses the bidirectional communication layer. This allows to execute arbitrary code on the server or on a certain client. Arguments and return values of such blocks are serialized transparently. Listing 1.10 shows an example method that uses an onServer-block to compile Smalltalk source code as method of a class. The method is part of a system editor that displays and changes the server's Smalltalk classes in a browser.

```
SaveButton>>#compileSource: source of: className

| block |
block := [ :cls :src | class |
    class := (Smalltalk classNamed: cls).
    class compile: src ].
block onServer
    value: className
    value: source.
```

Listing 1.10: A client-side method compiles source code on the server.

Even though such blocks allow to express client-server communication solely within Smalltalk, statements have to be wrapped in blocks and these blocks require arguments. In consequence, developers again define interfaces for the purpose of single interactions. Additionally, only globally accessible objects can be used in such remote code blocks.

To use object interfaces directly, we implemented transparent message forwarding on top of our bidirectional communication layer.

4.3 Transparent Message Passing

Even though a remote message is addressed to a remote object, it needs to be sent to a local receiver. This receiver is a proxy that forwards all messages. Since there is no hook for method invocation in Squeak, we forward messages through Smalltalk's `doesNotUnderstand:` feature. When a message is send in Smalltalk, the receiver's class looks up a method to handle the message. If this class does not provide an implementation for the received message, the lookup continues with its superclass. The lookup proceeds until a method is found or the inheritance chain was completely traversed. If no method is found, a `doesNotUnderstand:` message containing the original message is send to the receiver. Since we implemented this mechanism for Orca's Smalltalk environment in JavaScript [27], we can rely on this hook to forward messages. On the client and the server, our proxy classes directly extend basic class objects and therefore inherit nearly no methods. Additionally, we prefixed all methods of our proxy classes to maximize the probability that arbitrary messages sent to our proxies end up in `doesNotUnderstand:` . We can then pass the received message on to the client with respect to our serialization rules for arguments and return values, explained in Section 3.

On the server, proxies reference a session and an identifier for a remote object. On the client, proxies hold only an identifier since the associated remote objects reside always on the single server. On each client and for each session on the server, a map, which we call *Remote Object Map*, associates identifiers with receiver objects. Besides associating, these Remote Object Maps also protect remotely referenced objects from local garbage collection.

For proxy creation and the actual message forwarding we use our bidirectional communication layer. A message handler, which we call *Remote Object Handler*, conducts remote instantiations and forwards messages.

Client-side programs can create proxies for any Smalltalk class by sending `asRemote` to the class. When that method is invoked, the following steps are performed:

1. The class uses the bidirectional communication layer's `send:` interface to transmit an instruction for a new proxy. This instruction contains the name of the class.
2. The Remote Object Handler checks that a class of that name is available on the server.
3. (a) If such a class is available and not already included in the server's Remote Object Map, the class is inserted into the map and a new identifier returned. If the class is already inside the map, its existing identifier is returned.
   (b) If the class is not available, an error message is returned.

4. Either the remote identifier or the error message is sent to the client leading to either a new proxy with that identifier or an exception with that message.

This procedure is similar for proxy generation on the server-side, except that the method for explicit proxy creation is `asRemoteIn:` . A session object has to be supplied as argument to indicate which client is meant.

Synchronous message sends rely on the `send:` method, while oneway message sends use the `sendForked:` method of the layer's interface. These layer methods are not called directly, but are invoked automatically whenever a proxy object receives a message through usual sends or our explicit oneway sends.

When a proxy receives a usual message on the server, the framework performs the following steps:

1. The proxy receives the original message in its `doesNotUnderstand:` method.
2. The proxy forwards the message to its session.
3. The session sends the message name, arguments and the remote identifier to the Remote Object Handler of its client. This send relies on the bidirectional communication's `send:` interface.
4. The Remote Object Handler of the client retrieves the identified object and performs the received message. The handler catches all exceptions to forward errors to the server.
5. It transmits the result of this message to the server.
6. The session object on the server checks the answer for an error. If an error was received, the session throws it. Otherwise, it returns the answer to the sender.

Message forwarding from clients to the server is similar. Oneway messages are sent with the bidirectional communication's `sendForked:` method and return `nil` directly. Application exception are not returned.

4.4 Limitations

The implementation of Orca's remote messaging approach is limited by an optimization of the Squeak virtual machine. Additionally, at the time of writing, there is no garbage collection for objects that have been referenced remotely.

Orca's proxies rely on Smalltalk's `doesNotUnderstand:` hook that is called when no appropriate method is found on method lookup. However, this method is only called for messages that are send and Squeak's virtual machine optimizes certain message sends by direct execution without message send. Such *no-lookup-methods* cannot be intercepted and forwarded by our proxies. This may or may not pose a problem since only a few specific messages like `class` and `ifTrue:ifFalse:` are not send.

Orca provides no distributed garbage collection at the time of writing. When a proxy for an object is created, a map associates an identifier with that object. To allow the local garbage collectors to reclaim such associated objects, the association must be removed. For now, we remove these references when a client is no longer available since the map is referenced by a client's session.

However, these limitations do not prevent the development of collaborative web applications with Orca, which we will evaluate in the following section.

5 Case Study

Client and server collaborate in the execution of web applications. Either they send HTTP requests directly or they rely on higher abstractions. The Orca framework provides such an abstraction in form of message passing. We will compare both approaches in a case study.

All major browsers provide a JavaScript interface to the HTTP protocol. Developers can either use this interface or libraries that wrap it. Since those libraries only offer convenience functions, developers still need to create requests manually. We will compare Orca to a plain JavaScript solution. The JavaScript solution uses a JavaScript server to focus the comparison on communication mechanisms and not on single- versus multi-language development.

Performance is not part of the comparison since both approaches can theoretically transfer data with an equal number of requests. Also, the primary goal of Orca's remote messaging is not to increase performance, but to increase transparency.

To restrict the scope of the comparison, we only show implementation parts that provide client-server communication. The examples are not part of existing applications, but highlight key differences between both approaches.

A collaborative painting tool serves as example application. Users manipulate a canvas with a brush tool to add strokes or an eraser tool to remove strokes. Clients send all changes to the server, which spreads brushing and erasing to other clients. The involved objects are shown in Figure 4. A canvas object displays and erases brush strokes on the client-side, while a hub object stores and propagates strokes on the server-side.

Implementation with Orca: On initialization, a canvas creates a proxy for the server's hub singleton and sends a synchronous remote message for all available strokes (Listing 1.12). Since the stroke class defined that instances should be copied on remote message sends (Listing 1.11), this synchronous remote message returns a local array of local strokes and the canvas draws them without further remote message sends. After drawing all strokes, the canvas registers for updates

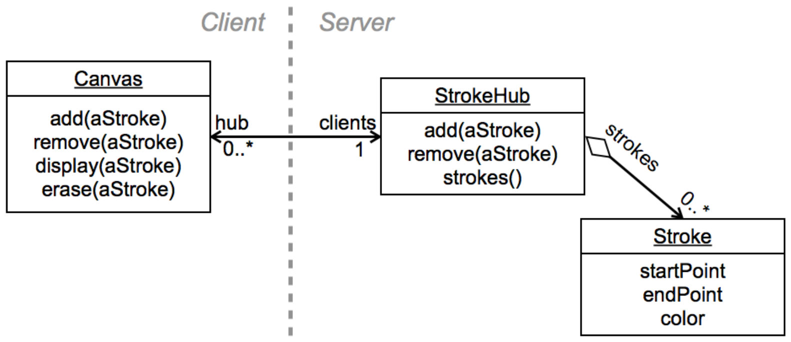


Fig. 4: Objects of a collaborative painting tool.

at the server by sending itself. On the server-side, the hub receives a proxy of the canvas.

For new strokes the client performs a oneway message send that carries the stroke object (Listing 1.13). The server receives a copy of the stroke instance, adds it to its collection and performs a one oneway message send for each registered canvas (Listing 1.14).

The eraser functionality is implemented similarly, except that messages are send to invoke the `remove:` and `erase:` methods.

```
CanvasStroke class>>#copyOnSend

    ↑ true
```

Listing 1.11: Canvas strokes should be transferred as copies.

```
Canvas>>#initialize

    | strokes |

    self remoteStrokeHub: StrokeHub asRemote uniqueInstance.
    strokes := self remoteStrokeHub strokes.
    strokes do: [ :each | self display: each ].
    self remoteStrokeHub addCanvas: self.
```

Listing 1.12: A client initializes its canvas.

Implementation with JavaScript: When a client connects to the server, the initialize function opens an asynchronous Comet request to the server to allow instant canvas updates (Listing 1.15). Furthermore, it issues a synchronous request addressed to the server to obtain all strokes. When the server responds

```
Canvas>>#add: aStroke

    self remoteStrokeHub
        performForked: #add:
        with: aStroke.
```

Listing 1.13: A client sends a new stroke to the server.

```
StrokeHub>>#add: aStroke

    self strokes add: aStroke.
    self registeredCanvases do:
        [ :each |
        each
            performForked: #display:
            with: aStroke].
```

Listing 1.14: The server stores and distributes strokes.

the strokes, the client deserializes the JSON data to create stroke objects and displays them.

Whenever a user draws a stroke, the client sends a request to the server with a name-value-pair (Listing 1.16). The key is *"newStroke"* and it holds a JSON representation of the stroke. The server dispatches the request by its URL address to a method. This method accesses the value of the request's name-value-pair by name and deserializes the JSON representation to store the stroke object in the server's collection (Listing 1.17). Moreover, this method propagates the name-value-pair representation of the stroke to all clients using the Comet request.

Similar functions handle events from the eraser tool.

```
function /*Canvas*/ initialize() {
    this.comet().open();
    var request = new XMLHttpRequest();
    request.open("GET", this.strokesURL(), false);
    request.send(dataString);
    var strokes = JSON.parse(request.responseText);
    strokes.forEach(function (each) {
        this.display(each);
    })
}
```

Listing 1.15: A client initializes its canvas.

```
function /*Canvas*/ add(aStroke) {
    var dataString = "newStroke=" + JSON.stringify(aStroke);
    var request = new XMLHttpRequest();
    request.open("POST", this.addStrokeUrl(), true);
    request.send(dataString);
}
```

Listing 1.16: A client sends a new stroke to the server.

```
function /*StrokeHub*/ add(request, response) {
    var stroke = JSON.parse(request.bodyAt("newStroke"));
    this.strokes.push(stroke);
    this.allRegisteredClients.forEach(function(each) {
        this.comet().pushData(each, request.body);
    })
    response.writeHead(200);
    return response;
}
```

Listing 1.17: The server stores and distributes strokes.

Of course, the applications could be implemented in various ways. For example, developers might decompose the application differently and might shift responsibilities between objects. This would change the application's architecture and collaboration between parts. However, certain pieces of the implementations are characteristic to the approaches.

Transmission entities: All parts of the distributed web application communicate through HTTP requests. The JavaScript functions create new requests and edit responses explicitly. Developers need to know the HTTP protocol and a JavaScript interface to it. In contrast, Orca abstracts from HTTP. Objects send Smalltalk messages that automatically result in HTTP request. Developers neither need to know the protocol nor the specific JavaScript interface to it.

Object serialization: Arguments for invocation need to be transferred in serialized form. The JavaScript solution uses the JSON format for serialization. All functions call the browser's JSON object explicitly to serialize data. In the process, data is always passed by value. With Orca, there is no manual serialization. The framework transfers arguments and return values of message sends through automatic serialization of objects and creation of proxies. Developers decide whether objects should be passed by value or by references.

Remote addressing: In the JavaScript example, requests are explicitly send to a specific URL. The server dispatches requests to functions. If the same address is used for different requests, the dispatch takes a request's name-value-pairs into account. The URLs and name-value-pairs effectively form an interface between client and server. With Orca, objects send messages to other objects. The framework automatically converts message sends to proxies into necessary HTTP requests. As a result, developers can program against already defined object interfaces.

So, both presented implementations differ in regards to conceptual transmission entities, visibility of serialization and how remote methods are addressed. These differences affect three objectives for object collaboration in distributed systems. Object collaboration should be apparent, transparent and separated.

Apparent collaboration: Communication across computer networks is not as reliable as computing on a single machine [2]. Developers have to be aware of the reduced reliability and have to expect network-related exceptions. Additionally, remote invocations are potentially orders of magnitude slower then local invocations. For these reasons, developers should employ object collaboration across distribution boundaries consciously and all interactions should be apparent. When HTTP is used directly, requests are created explicitly and addressed to URLs. Hence, collaboration between remote objects is directly visible to developers. Orca uses message sends for local and remote invocations depending on the receiver. When the receiver is a proxy for a remote object, it forwards the message transparently. Object collaboration can be less apparent in Orca applications. By convention, we reflect the execution location in variable names in our applications.

Transparent collaboration: Developers often need to understand program behavior by reading code. Therefore, collaboration between remote objects should be transparent to allow correct predictions of program flow. In the JavaScript solution, requests are mapped to functions according to their URL address. Often, such dispatch also takes name-value-paris into account. Developers need to be aware of the mapping between requests and functions and need to implement against the additional interface. In contrast, since Orca forwards Smalltalk messages to remote objects, no additional abstraction is necessary. Developers can employ their usual development tools to search for implementations and senders by message name. We argue that Orca's approach increases transparency of object collaboration in web applications.

Separated collaboration: A concern is any consideration that can impact the implementation of a program [21]. Such a concern is either separated in its own module or crosscuts the implementation of other concerns [17]. A crosscutting concern is scattered across entities of a specific abstraction level and possibly tangled with the source code related to other concerns. Crosscutting concerns reduce software quality and impede maintenance [6]. In the JavaScript solution, collaboration is scattered since each remote invocation is apparent in two functions. One function serializes and stores data in a HTTP package, while the other function retrieves and deserializes the data. Functions are tangled since they implement collaboration and application logic. Thus, collaboration between remote objects is a crosscutting concern in the JavaScript implementation. In the Orca solution, the methods only implement application logic. Interaction between remote objects is implemented directly, not scattered across the implementation and, therefore, no crosscutting concern. According to statistical results [6], this means that adaptions to Orca applications are less likely to create defects through inconsistent adaptions.

While remote invocations are less apparent in Orca applications, the collaboration between remote objects is more transparent and implementations less scattered and less tangled.

6 Related Work

Our message passing approach relates to three categories of preceding research and alternative approaches. There are different approaches to remote invocations, other technologies provide bidirectional communication as alternative to HTTP, and related web application frameworks apply remote invocation mechanisms.

6.1 Remote Invocations in Distributed Systems

Remote invocation facilities have been developed to alleviate the implementation of distributed systems. The following presents the first remote procedure call facility, the original work on transparent message passing in a Smalltalk system, a widely accepted standard for remote procedure calls in heterogenous environments, and a standard for remote invocations between web servers.

Remote Procedure Calls Originally, remote procedure calls (RPCs) [3] have been implemented as a novel approach to distributed systems development at Xerox PARC. RPCs provide transfer of control and data across a network by procedure invocation. An RPC suspends the local process and transmits the call, which invokes execution of a remote procedure. The call contains all arguments in serialized form. After execution the result is passed back and resumes the calling process. RPCs allow to call remote procedures as if they are local.

Developers need to expose interfaces for these remote procedure calls explicitly in an interface description language. The RPC facility developed at PARC provided a tool to translate such descriptions into bindings for several programming languages.

Distributed Smalltalk Smalltalk's dynamic type system and its message-based invocations allow to pass messages to arbitrary objects instead of calls to explicitly exposed procedures. User-transparent remote messaging across Smalltalk-based systems was first implemented for Distributed Smalltalk [1]. The implementation only passes certain basic types as copies and generates proxies for all other types. All messages sent to proxies are forwarded transparently.

Orca's approach and implementation builds upon the concepts developed for Distributed Smalltalk. However, in contrast to Orca, the implementation of Distributed Smalltalk provides support for access control, object mobility and an incremental distributed garbage collector. Additionally, even though proxies are also implemented with Smalltalk's `doesNotUnderstand:` hook, Distributed Smalltalk implements a utility to rename no-lookup selectors to looked up aliases.

Distributed Smalltalk only allows to send remote messages synchronously, whereas Orca offers oneway messages that do not block the sending process.

CORBA While numerous RPC solutions were proposed, few were standardized to the extend of the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) [20,25]. CORBA abstracts heterogenous collaboration infrastructures by its Interface Definition Language (OMG IDL) that includes standardized types and exceptions. Numerous programming languages provide compilers for the OMG IDL that generate client-side stubs and server-side *skeletons* and offer mappings for the OMG IDL types.

Besides standardization, potential heterogeneity of infrastructures and necessity to generate stubs, Orca differs in regards to proxy creation. References to remote objects can be created by sending the `asRemote` message or the `asRemoteIn:` message to a class with Orca, whereas in CORBA clients have no special operation for remote handle creation. Remote references are either obtained through *Directory services*, or by using so called *factory objects*. While directory services provide handles to already existing objects, factory objects return proxies of newly objects. CORBA provides a dynamic invocation interface that can be used without compile-time knowledge of objects interfaces.

Web Services Web Services [5] promise universal access to heterogenous applications available on the Internet. The standard aims to establish web application development as composition of Web Services. Applications can request information from Web Services through the Simple Object Access Protocol (SOAP), which encodes messages or RPCs in XML.

Several protocols including HTTP can be used to transfer SOAP packages. The interface description language of Web Services is called Web Service Description Language (WSDL) [4] and allows dynamic binding and generation of SOAP-based interfaces.

While the Web Service standard offers interoperability between heterogenous web applications based on XML-encoded messages, Orca forwards messages between homogenous environments. Also, since the browser's "native language" is JavaScript, we chose JSON as representation of messages. And since we pass messages instead of calling procedures, interface descriptions through languages like WSDL are not necessary with Orca.

6.2 Bidirectional Communication for Web Applications

In the future, web frameworks might rely on alternative technologies that incorporate bidirectional communication. The following presents two alternatives that are discussed at the moment of writing.

SPDY Google's SPDY[3] is an experimental application-layer protocol for transporting content. SPDY builds upon the Transmission Control Protocol (TCP). It adds a layer between SSL and HTTP, to allow multiple concurrent, bidirectional streams over a single secure TCP connection with minimal deployment effort. Applications still communicate through HTTP requests and responses, but the SPDY layer allows server-side and concurrent sends without a request limit. SPDY implements request prioritization and header compression to address limited bandwidths. Although Google build a first SPDY-enabled browser, the protocol is neither a standard nor implemented in any other browsers, yet.

WebSockets WebSockets [14] are bidirectional connections. For example, developers can setup an event handler that is called whenever a connection closes instead of implementing timeouts for HTTP. However, WebSockets only provide an interface for asynchronous sends. Also, at the moment of writing, the WebSockets specification is in *W3C Editor's Draft* status and not all major browsers have implemented the interface.

6.3 Remote Invocations in Web Applications

The following relates Orca to frameworks that provide remote invocation facilities and allow to express client- and server functionality in a single language.

[3] `http://www.chromium.org/spdy/spdy-whitepaper`, accessed June, 21, 2011

Google Web Toolkit Google Web Toolkit (GWT) [24] provides a Java-to-JavaScript compiler and an emulation of core Java constructs and classes in JavaScript. Developers write the complete application in Java. Besides basic requesting, GWT provides an RPC facility that transfers method invocations and serializes all arguments automatically. Only certain types are allowed as parameters, but developers can declare their own classes serializable by implementing a particular Java Interface. In contrast to Orca, GWT does not create proxies deliberately for arguments and return values that are not of certain types.

GWT's RPCS are called *Server Calls*. Developers can call to client-side stubs to invoke server-side methods. Servers cannot push data to clients spontaneously. All Server Calls are asynchronous and require a callback object as parameter. The callback defines methods that are called on success and failure. The failure handler must handle checked exceptions, while RPC invocations may happen at runtime as well. GWT's server calls present themselves as remote calls, while Orca's message sends appear as local calls.

Proxies can be created for all classes that extend a certain Java class and implement a specific interface.

HOP The HOP [22,23] programming language specifies all aspects of web applications in a single programming language. The server executes a Scheme dialect, while a compiler generates JavaScript code for client-side execution. HOP programs contain syntactic annotations that define the execution environment of statements. Developers can escape from client-side to server-side execution and vice versa. Since HOP renders applications to markup on the server, escaping to client-side in HOP programs effectively defers execution.

HOP provides calls to remote functions that accept a callback function. Remote function calls do not support automatic serialization.

The language implements a long-polling event loop similar to Comet. Clients register explicitly for specific events, whereas in Orca the server can send arbitrary messages to all clients.

Moreover, both approaches imply different ways of application partitioning. HOP allows inline escaping on a per statement base, while the location of execution is defined per object with Orca.

7 Conclusion

With Orca, distributed objects interact through Smalltalk messages, while the framework abstracts the underlying HTTP protocol. Messages invoke methods transparently according to object interfaces. Objects can be created locally or remotely. This decision defines where objects reside and partitions applications into client and server components. All aspects of web applications are expressed in a single object-oriented programming language. Orca unifies client- and server-resident parts and necessary interaction between parts.

This frees developers from using HTTP directly, from manual serialization and the definition of own ad-hoc protocols on top of HTTP. Developers define only object-oriented interfaces and can search for implementations and senders with the message's name. Object collaboration is on the abstraction of usual method invocation and transparent to developers. Implementations are less crosscutting and, for this reason, less error-prone, since modifications affect fewer areas and inconsistent adaptations, therefore, less likely.

In the future, we will address distributed garbage collection. Furthermore, remote message sends carry arguments either as copies or create proxies transparently. We would like to provide an alternative mobility operation that moves objects instead of copying. Such an operation would need to exchange references to the real object with references to a proxy and vice versa. Finally, we need to address security concerns at some point.

Despite this, the Orca framework already permits the construction of complete web applications within Smalltalk and distributed objects interact transparently.

References

1. Bennett, J.K.: The Design and Implementation of Distributed Smalltalk. In: Proceedings of the ACM SIGPLAN Conference on Object-oriented programming systems, languages and applications. pp. 318–330. OOPSLA '87, ACM (January 1987)
2. Birman, K.P.: Reliable Distributed Systems: Technologies, Web Services, and Applications. Springer-Verlag (March 2005)
3. Birrell, A.D., Nelson, B.J.: Implementing Remote Procedure Calls. In: Proceedings of the ninth ACM Symposium on Operating Systems Principles. pp. 3–24. SOSP '83, ACM (December 1983)
4. Chinnici, R., Moreau, J.J., Ryman, A., Weerawarana, S.: Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. World Wide Web Consortium, Recommendation (June 2007)
5. Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., Weerawarana, S.: Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. IEEE Internet Computing 6, 86–93 (March 2002)
6. Eaddy, M., Zimmermann, T., Sherwood, K.D., Garg, V., Murphy, G.C., Nagappan, N., Aho, A.V.: Do Crosscutting Concerns Cause Defects? IEEE Transactions on Software Engineering 34, 497–515 (July 2008)
7. European Association for Standardizing Information and Communication Systems: ECMA-262: ECMAScript Language Specification (December 1999), `http://www.ecma-international.org/publications/standards/Ecma-327.htm`
8. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. World Wide Web Consortium,

Request For Comments (June 1999), http://www.w3.org/Protocols/rfc2616/rfc2616.html

9. Fielding, R.T., Taylor, R.N.: Principled Design of the Modern Web Architecture. In: Proceedings of the 22nd international conference on Software engineering. pp. 407–416. ICSE '00, ACM (June 2000)
10. Flanagan, D.: JavaScript: The Definitive Guide. O'Reilly Media (August 2006)
11. Friedman, D.P., Wise, D.S.: The Impact of Applicative Programming on Multiprocessing. In: Proceedings of the International Conference on Parallel Processing. pp. 263–272. ICPP '76, IEEE (July 1976)
12. Goldberg, A., Robson, D.: Smalltalk-80: the language and its implementation. Addison-Wesley (January 1983)
13. Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.J., Nielsen, H.F.: SOAP Version 1.2 Part 1: Messaging Framework. World Wide Web Consortium, Recommendation (June 2003), http://www.w3.org/TR/soap12-part1/
14. Hickson, I.: The WebSocket API. World Wide Web Consortium, Editor's Draft (June 2011), http://dev.w3.org/html5/websockets/
15. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future: the story of Squeak, a practical Smalltalk written in itself. In: Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. pp. 318–326. OOPSLA '97, ACM (October 1997)
16. Jazayeri, M.: Some Trends in Web Application Development. In: Proceedings of the 2007 Future of Software Engineering. pp. 199–213. FOSE '07, IEEE (May 2007)
17. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Proceedings of the European Conference on Object-Oriented Programming. p. 220–242. ECOOP '97, ACM (December 1997)
18. Kuuskeri, J., Mikkonen, T.: Partitioning Web Applications Between the Server and the Client. In: Proceedings of the 2009 ACM Symposium on Applied Computing. pp. 647–652. SAC '09, ACM (March 2009)
19. Mikkonen, T., Taivalsaari, A.: Web Applications – Spaghetti Code for the 21st Century. In: Proceedings of the 2008 Sixth International Conference on Software Engineering Research, Management and Applications. pp. 319–328. SERA '08, IEEE (August 2008)
20. Object Management Group: The Common Object Request Broker Architecture: Core Specification, V. 3.0.3. (March 2004), http://www.omg.org/spec/CORBA/3.0.3/
21. Robillard, M.P., Murphy, G.C.: Representing Concerns in Source Code. ACM Transactions on Software Engineering and Methodology 16 (February 2007)
22. Serrano, M.: Programming Web Multimedia Applications with Hop. In: Proceedings of the 15th international Conference on Multimedia. pp. 1001–1004. MULTIMEDIA '07, ACM (September 2007)
23. Serrano, M., Gallesio, E., Loitsch, F.: Hop: A Language for Programming the Web 2.0. In: Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications. pp. 975–985. OOPSLA '06, ACM (October 2006)

24. Smeets, B., Boness, U., Bankras, R.: Beginning Google Web Toolkit: From Novice to Professional. Apress (September 2008)
25. Vinoski, S.: CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. IEEE Communications 35, 46–55 (October 1997)

Bachelor Theses

26. Eckhardt, S.: An Architecture Overview of the Orca Web Framework. Bachelor's thesis, Hasso Plattner Institute for Software Systems Engineering (June 2011)
27. Strobl, R.: Mapping Smalltalk Language Features to JavaScript. Bachelor's thesis, Hasso Plattner Institute for Software Systems Engineering (June 2011)
28. Wassermann, L.: Translating Smalltalk to JavaScript. Bachelor's thesis, Hasso Plattner Institute for Software Systems Engineering (June 2011)