Listen

Simplifying web-based applications through actor-oriented programming and publish/subscribe messaging

Imperial College London
Department of Computing

15th June 2010

Samir Talwar
samir.talwar06@imperial.ac.uk

Supervisor
Prof. Alexander L. Wolf
a.wolf@imperial.ac.uk

Abstract

The idea behind *Listen* was originally to create a web framework that supported user-created extensions and plugins. More than that, it was intended to make these extensions not just easy to develop, but also easy to support, and reward developers who provide hooks that allow others to work seamlessly within their websites.

As web applications become part of our everyday lives, it becomes increasingly difficult to predict the necessary functionality to keep users happy. Simultaneously, integrating all the possible use cases into one application tends to bloat it, making it very difficult to present a clean interface to the user. The concept of extensible interfaces in desktop application development has existed for a long time, but is not possible in the majority of web applications.

Listen changes this by using the actor model to aid developers in splitting software up into discrete, independent processes which communicate over a publish/subscribe messaging architecture. Because any process, including an externally-developed one, can subscribe to a specific message topic, extensions that interface with existing functionality to seamlessly integrate themselves into the main application can be easily created, providing more features without the clutter and delivering a better user experience.

Acknowledgements

I would like to state my appreciation toward my supervisor, Professor Alexander Wolf, and my second marker, Dr. Cristian Cadar, for providing me with their most welcome support, as well as listening and offering constructive feedback to my ramblings in their respective offices. Their well-placed comments amidst my deluge of random thoughts helped keep me on the right path, and for that, I am incredibly grateful.

I must also express thanks to my partners in crime, the men and women of Imperial College's Department of Computing that slaved over their projects as I slaved over mine, keeping me sane throughout the process. I wish you all luck in the years ahead.

Contents

Chapter 1

Background

Listen is a web application architecture based on the actor model: each application consists of lightweight, independent processes that communicate through message passing. These processes run concurrently on the server as well as each individual client—the web browsers used to access the application websites—and transmit information through a standardised protocol which can send messages not only between the client and the server, but also to other servers. By splitting applications in this way and implementing a very loosely coupled communication system, Listen tackles a number of areas, including:

- concurrent programming
- multi-tier broadcast communication
- reusable, task-oriented code
- community-driven extensions

These features culminate in a framework which allows anyone to construct a web application quickly and easily. In addition, any user with a development background can modify the resulting site to better fit their specific needs, creating an open structure that leverages the power of communities to build a better web.

1.1 Actors

The actor model was initially created by Carl Hewitt, Peter Bishop and Richard Steiger as an architecture for artificial intelligence programming with the understanding that all forms of control and data flow can be represented

as messages. Actors were influenced by languages such as Smalltalk and Lisp, which lend themselves well to building compositions from small chunks. Similarly, actors, in their most pure form, are composed entirely of messages to other actors:

> Data structures, functions, semaphores, monitors, ports, descriptions, Quillian nets, logical formulae, numbers, identifiers, demons, processes, contexts, and data bases can all be shown to be special cases of actors. All of the above are objects with certain useful modes of behavior. Our formalism shows how all of the modes of behavior can be defined in terms of one kind of behavior: *sending messages* to *actors*.
>
> *Hewitt, Bishop and Steiger*, A Universal Modular ACTOR Formalism for Artificial Intelligence [1]

This sort of methodology lends itself to the development of new domain-specific languages which treat actors as part of the language. One such example is ActorScript, a theoretical language developed recently by Hewitt, in which every entity is an actor [2].

Most, however, take a more practical approach. Erlang, for example, is a functional language developed by Joe Armstrong for Ericsson that uses actors to implement concurrent programming. Listing 1.1 is a very simple program that demonstrates a few of the features actors have to offer.

Even this relatively simple program shows evidence of a number of benefits of the actor model. Our two actors, or "processes", as Erlang terms them, are `ping` and `pong`, which are spawned by the `run` function and, using the `!` operator, send messages to each other containing a reference to the sender. Both processes are idle until they receive a message, but when one does, it sends a message to the other, which fires a message back, and so on, until `Count` messages have been sent. During this, they print a line with each message, and so the result of calling `run(3).` would be six lines of alternating "Ping!" and "Pong!".

As each actor maintains its own state, they are completely insulated from the global program, allowing the script interpreter to optimise the threading to suit the current execution. Locks are also avoided—explicit synchronisation is often not necessary, as processes do not share data, and waiting for a result can usually, if not always, be modelled as a message receipt function.

Fortunately for the majority of web developers working in object-oriented and imperative paradigms, the actor model is not limited to the domain of functional languages. Scala, for example, is a multi-paradigm language that

```
1  -module(pingpong).
2
3  % This makes our three functions public.
4  -export([run/1, ping/1, pong/1]).
5
6  run(Count) ->
7      % Create two actor objects called `Ping` and `Pong`,
8      % instances of the `ping` and `pong` .
9      Ping = spawn(pingpong, ping, [Count]),
10     Pong = spawn(pingpong, pong, [Count]),
11     % Send `Ping` a message containing a reference to `Pong`.
12     Ping ! {pong, Pong},
13     % End on a high note.
14     true.
15
16 % If we're on the last one, we're done: return true.
17 ping(0) -> true;
18 % Otherwise, let's go.
19 ping(Count) ->
20     receive
21         % If we get a message referring to a `pong` instance:
22         {pong, Pong} ->
23             % Write "Ping!" to the console.
24             io:fwrite("Ping!\n"),
25             % Send that instance a message containing a
26             % reference to this `ping` instance.
27             Pong ! {ping, self()},
28             % Recurse - start listening for messages again.
29             % Decrement the counter - when we hit 0, we stop.
30             ping(Count - 1)
31     end.
32
33 % Works the same way `ping` does.
34 pong(0) -> true;
35 pong(Count) ->
36     receive
37         {ping, Ping} ->
38             io:fwrite("Pong!\n"),
39             Ping ! {pong, self()},
40             pong(Count - 1)
41     end.
```

Listing 1.1: A simple Erlang program that creates two actors.

provides the actor model as a library. Due to the simplicity of the language and its ability to define new operators, utilising it makes Scala feel as if the entire language were actor-based. Extensions have also been written for a number of other languages, including Stage [3], a framework that implements actors on top of Python. In both cases, such a fusion allows developers to switch between imperative, object-oriented, functional and actor-based programming as and when they need, allowing for maximum expressibility.

In all cases, the threading model is one of the key strengths of an architecture based upon message passing. By idling processes when they aren't executing code, a single thread can handle a number of them. By creating a thread pool which can grow and shrink as necessary, Erlang can handle several million processes without issue [4].

Implicit concurrency is not the only benefit of actors. As shown above, processes in Erlang can send messages containing references to themselves or other processes, allowing for very loose coupling. Processes can spawn other processes, send messages to them and receive messages in turn, even if the process is not known to exist at compile time. In this respect, they behave in similar ways to languages that use an object-oriented paradigm, where due to inheritance, it is possible to extend a program such that classes may handle other classes that did not exist at the time of writing the program. This also allows for easy unit testing, as each actor is an island which can be loaded on its own and probed from all sides without ever having to invoke others.

Finally, actors are generally intended to be very lightweight. By limiting an actor to one small aspect of the overall program, the entire program becomes very modular, allowing developers to easily change one part without breaking the rest. In addition, due to the dynamic nature of messages, even compiled languages only need to replace the changed segments, with the rest remaining untouched—Erlang even provides support for hot-swapping modules, so developers can install new functionality without ever stopping the program. Adding features to a program is also fairly simple—simply create a new actor and connect it to the rest of the system via message hooks.

Any system can be represented as a set of actors. Throughout this report, we will use a basic email software as an example to demonstrate functionality. One can think of a web-based email application as being made up of a number of discrete pieces of functionality. The server side consists of the processes that transfer email across domains, as well as storage and user management. Meanwhile, the client handles user interaction—accepting new emails to send, displaying existing emails, displaying folders (or labels, in the case of Google's *Gmail*) and contacts and providing the user with the ability

to manipulate them. Each one of these could be represented as one or more actors, depending on the aim of the developer, and as they are completely separate, be distributed amongst any number of servers.

With all these features in mind, actors could be the perfect base unit for modular, extensible websites.

1.2 Messaging

A large part of the actor model consists of passing messages between processes. With this in mind, it is important to decide on a correct paradigm such that messages are effectively delivered across a multi-tier system.

1.2.1 Message Queues

One approach taken by component-oriented systems is that of the message queue, in which a queue is created for each thread or process. Messages are sent asynchronously from one process to another, where it is added to the queue to be processed at a later time. This approach is common in event-based applications, where messages trigger event handlers. Using a queue means that messages will be processed in the same order they arrived, causing a deterministic result, and will not be missed (assuming infinite buffer length and perfect communication avenues). The Windows event loop, used in all Win32 applications, uses such a queue to notify applications of all system- and user-driven events [5]. Asynchronous event messages such as these are not based on a request-response mechanism, despite being fairly analogous to the traditional models of "user acts, program responds", but instead follow a model that whilst more complex to design, allows for message handlers that work concurrently with the rest of the system.

Typically, such message queuing systems are implemented as independent software, creating queues for all compliant applications. The queue manager will sort and deliver messages to the correct queues, and may even store messages in the case of a disconnected application. More advanced systems will accept metadata as part of the message that defines how it is handled, ranging from security policies to the exact delivery mechanism to be used.

There are some well-known message queues that operate synchronously. For example, the web itself is based on a blocking request-response mechanism in which the browser makes a request—sends a message—asking for a specific web page, and receives a reply a short period of time later. During the interval, no other communications are being made. However, this is only

true of single requests: in all other situations, asynchronous message passing results in a smoother, albeit more complicated experience.

Message queues can easily be applied to actors, and are indeed the method by which Erlang functions. Processes are idle by default, and "wake up" when messages are received. If another message is sent while the first is still being processed, it is added to the message queue, and the process will remove each message in turn and handle it. When there are none left, it returns to an idle state until the next message is received.

Perhaps the most obvious example of a message in the actor-based email system explained earlier is the email itself. For the user to send an email, he would open the composer, an actor in itself, write the email and send it. On sending, the composition actor would send the contents of the email in a message to an actor on the server which handles email sending.

1.2.2 Publish/subscribe

Message queues provide a reliable, non-blocking mechanism to convey information between processes, but they do not by themselves allow broadcasting to more than one receiver. Messages are passed by the middleware between any number of processes, but each message is addressed directly to the intended recipient. This is useful in many message-based applications, but in a framework where the eventual application is unknown, can be somewhat of a limiting factor. Workarounds involve sending several messages—one to each intended recipient—but this can place an unnecessary burden upon the sender, forcing it to keep a list of recipients for each type of message. Because of this tight coupling, any new processes will require changes in many existing processes, hugely increasing the amount of work necessary to introduce new features to an application.

One of the most popular and well-understood methods of creating a "one-to-many" messaging architecture is the *publish/subscribe* (*pub/sub* or simply *pubsub*) paradigm. In this, the roles are reversed: the sender no longer decides who the recipient of a message is. Instead, it simply publishes messages to a "topic". Any process can subscribe to the same topic and will automatically receive any messages published to it. A middle-man—a broker—handles the messages, maintaining lists of subscribers for each topic and forwarding messages to the correct recipients. This results in a very loose coupling between sender and receiver, contrary to the direct message passing system detailed above, which allows for easy development of new features that can easily integrate themselves with the rest of the system, simply by subscribing and publishing to existing topics.

> The publish/subscribe interaction paradigm provides subscribers with the ability to express their interest in an event or a pattern of events, in order to be notified subsequently of any event, generated by a publisher, that matches their registered interest. In other terms, producers publish information on a software bus (an event manager) and consumers subscribe to the information they want to receive from that bus.
>
> *Eugster, Felber, Guerraoui and Kermarrec*, The many faces of publish/subscribe [6]

There is no language-integrated facility for publish/subscribe messaging in any programming language. Such functionality must be integrated as a library or part of a framework. This is because when sending messages, the situation is somewhat more complicated. In our email example, the email-sending service would subscribe to a topic—preferably something descriptive such as "send-emails". The composer would then publish the message to "send-emails" and the broker would deliver it to all subscribers.

1.2.3 Existing Solutions

More traditional enterprise messaging solutions include WebSphere MQ [7], from IBM, and ActiveMQ [8], an Apache Foundation project. These fall under the umbrella of *Message-Oriented Middleware* (*MOM*), which provide various messaging frameworks to abstract away the complexity inherent in distributed, heterogenous applications. MOM software tends to be language-independent, yielding an API which can be accessed regardless of platform or programming language, and almost always implementing both point-to-point and publish/subscribe messaging paradigms at the very least. However, they are also based upon a server-to-server model rather than spanning both servers and clients, which tend to be situated behind many network layers and cause connectivity problems for anything that does not expect it. In addition, clients—standard desktop and laptop computers—are also geared towards general-purpose computing and multi-tasking, and tend to be far less powerful than the server counterparts on the other side of the connection.

Interestingly enough, middleware such as ActiveMQ tends to lend itself well to service-oriented architectures (*SOA*). There is no one definition for SOA, but the majority lean toward a set of services that are loosely connected through an underlying network architecture. Yvonne Balzer outlines some of the guiding principles of SOA projects as "reuse, granularity, modularity, composability, and componentization" [9], and also states the need

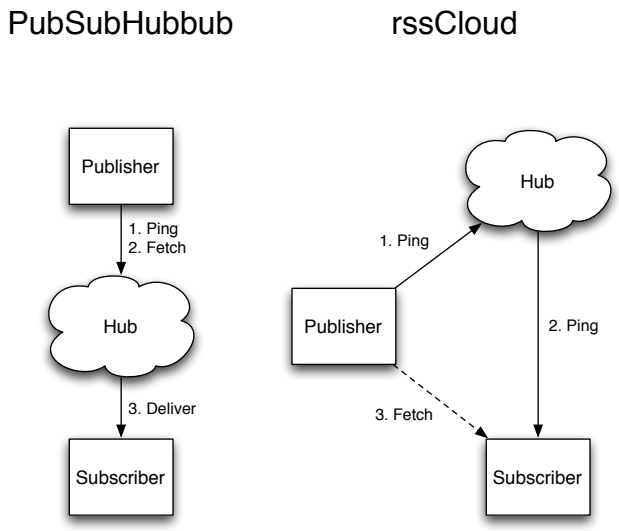

Figure 1.1: The *PubSubHubbub* and *rssCloud* publishing models

for services to be orthogonal and non-repetitive—some of the qualities also implicit in the actor model. Where they differ is in strictness: services, as defined by SOA, communicate through well-understood public interfaces, whereas actors may or may not implement event handlers for any conceivable message. More than this, SOA also guides the project itself, defining particular methods of progressing the architecture, dealing with third-party services and optimising relations between IT and the core business. In this sense, it is targeted in a similar way to the middleware it often uses, aiming to be an enterprise-level tool that helps build concrete requirements before implementation commences.

Publish/subscribe adheres well to Internet-based technologies, and as such, a number of protocols have also been designed to implement it across the web. Two of the most recent are *PubSubHubbub* and *rssCloud*, which define the architecture in terms of a publisher, a subscriber and a "cloud" or "hub". The difference between the two protocols, shown in figure 1.1, embodies the key debate among pub/sub creators: PubSubHubbub delivers content as it updates, whereas rssCloud simply delivers a link, leaving synchronisation to the discretion of the subscriber.

Both PubSubHubbub and rssCloud are designed in order to push Atom and RSS feeds, respectively, directly to a consumer, rather than having the consumer poll a producer on a regular basis as is currently the norm. As such, their choice of distribution mechanism is more fitting for an author-to-reader

relationship—The XML-based models are too heavy for fast, lightweight applications, and demonstrate a desire to publish news, blog posts and the like, rather than fuel inter-process communication.

The solution to a tiered messaging model seems to be somewhere in between all these different approaches. Traversing the client-server connection reliably and effectively is a tricky problem that may require a new paradigm, drawing on elements of existing protocols and tools in order to abstract away cross-tier communication in favour of an interface that behaves the same, whether running in a client-side script in a browser or an executable on a web server.

1.3 Model-View-Controller

Over the years, web development has matured to encompass a broad variety of design methodologies and patterns, allowing developers and designers to devise sites that reach realms of complexity never dreamed by the pioneers of the World Wide Web. By developing architectural patterns that abstract away the intricacies of HTTP and web servers in favour of an altogether simpler and more powerful set of tools, time is no longer being spent on making sites work, and is instead focused on the formulation of entirely new ideas. Stemming from this, frameworks have been designed to allow even the most novice programmer to not only create a website, but one that is modular, extensible and scales well.

One of the most prominent architectures used in web development today is *Model-View-Controller* (*MVC*), originally developed as a class library for Smalltalk–80 in order "to bridge the gap between the human user's mental model and the digital model that exists in the computer." [10] While at Xerox PARC, Trygve Reenskaug developed a paradigm that allowed for many-to-many relationships between the model, which is responsible for data handling and manipulation, and the view, which deals with the user interface and data representation to the user, giving developers and designers the ability to show information to the user in many different ways without a lot of duplicated code. In addition, a single controller can display any number of views at a time to the user, providing a composite UI design that can be reorganised as new features emerge without much effort at all.

The primary concern of the controller is to relay information between models and views. In this regard, it can be seen as superfluous—theoretically, it should be possible for models and views to communicate directly. Indeed, many websites use a concept derived from MVC named *Model-View-ViewModel* (*MVVM*), which they use with the Windows Presentation Frame-

work. In MVVM, the "ViewModel" replaces the controller. The actual architecture does not change, but the new naming convention better portrays the aim—the ViewModel simply acts as a bridge between the view and the model, exposing no additional functionality but simply allowing view developers to write user interfaces entirely in XAML, an XML-based presentation markup language. This extra layer encompasses the communication logic through data binding, separating the UI and business logic code out so designers can create views without having to know how to deal with the models, and programmers can do the same in reverse [11].

However, the controller also plays the role of the entrance point to the application, dynamically querying for data through the models and deciding which views to show upon request. Pushing all data through the controller also allows for input validation to be handled separately from the actual business logic in the model, which can aid in keeping the code base legible. Removing the controller from the equation could potentially cripple the architecture; directly transferring information between views and models should not be complicated, but in order to completely remove it, a new entry point must be decided upon and shown to be just as useful and simple to understand.

We can draw lines fairly easily between the different subsystems in our fictional email software to demonstrate the MVC divide. Everything running on the client falls under the domain of the View, and the storage aspects of the server are clearly part of the Model. The Controller, then, would be the aspects that allow the user to manipulate information: sending and receiving email and manipulating those that are already on the server, as well as auxiliary data such as contacts. On a publish/subscribe system, the broker acts as a ViewModel, directing information between views and models.

1.4 Client-Side Programming

In order to build processes that run on the client, part of the framework must be developed in a way that is compatible with the client-side environment. Essentially, this means that the code must run in a browser—it must be written in JavaScript. Generating a GUI on the server would defeat the point of having client-side processes: in order to fully exploit the actor model, the entire user interface must be generated programmatically on the client.

While methods familiar to JavaScript programmers may be viewed as unorthodox by those unfamiliar with the language, especially those coming from the object-oriented world, it is actually very powerful, allowing for a very expressive yet minimalist coding style. A forced resemblance to Java

often throws new developers off:

> JavaScript's C-like syntax, including curly braces and the clunky `for` statement, makes it appear to be an ordinary procedural language. This is misleading because JavaScript has more in common with functional languages like Lisp or Scheme than with C or Java.
>
> *Douglas Crockford*, JavaScript: The World's Most Misunderstood Programming Language [12]

Developer aversion aside, client-side scripting still has some problems. The first occurs when attempting to implement a messaging system: once a web page has loaded, the connection is closed, and the client no longer has an active connection to the server. Further connections can be opened through an `XMLHttpRequest`, but even those will only remain open for a short period of time. Almost since the inception of the web as a mechanism for communication, rather than static delivery, people have sought out solutions, resulting in plenty being developed over the years. For decent client-server messaging, a bidirectional communication layer is necessary.

1.4.1 Comet

One answer to this comes in the form of *Comet*, an umbrella term created by Alex Russell, president of the Dojo Foundation, for a grouping of several existing technologies which all attempt to reject polling in favour of an emulated *push* mechanism. It is also known by other terms, including "slow loading", "HTTP server push" and "Reverse Ajax", which lend some insight into the mechanisms behind it.

Comet works by establishing either an *HTTP stream*—a long-lived HTTP connection—or a long polling mechanism. In both cases, the client connects to the server, as the reverse is not possible over HTTP, and the connection is maintained for the lifetime of the application. This can be considered the reverse of Ajax calls (hence the name "Reverse Ajax"), which are executed once for each data request, and can only go from client to server and back: because of the long-lived connection, either side can send information to the other without difficulty.

Figure 1.2 demonstrates the benefits of a Comet-based model—it enables the server to push messages out as necessary, rather than requiring the client to poll repeatedly as in a traditional AJAX-based application. Because of the benefits brought by this at very little cost, the Comet style of delivering server messages is now prevalent among web applications.

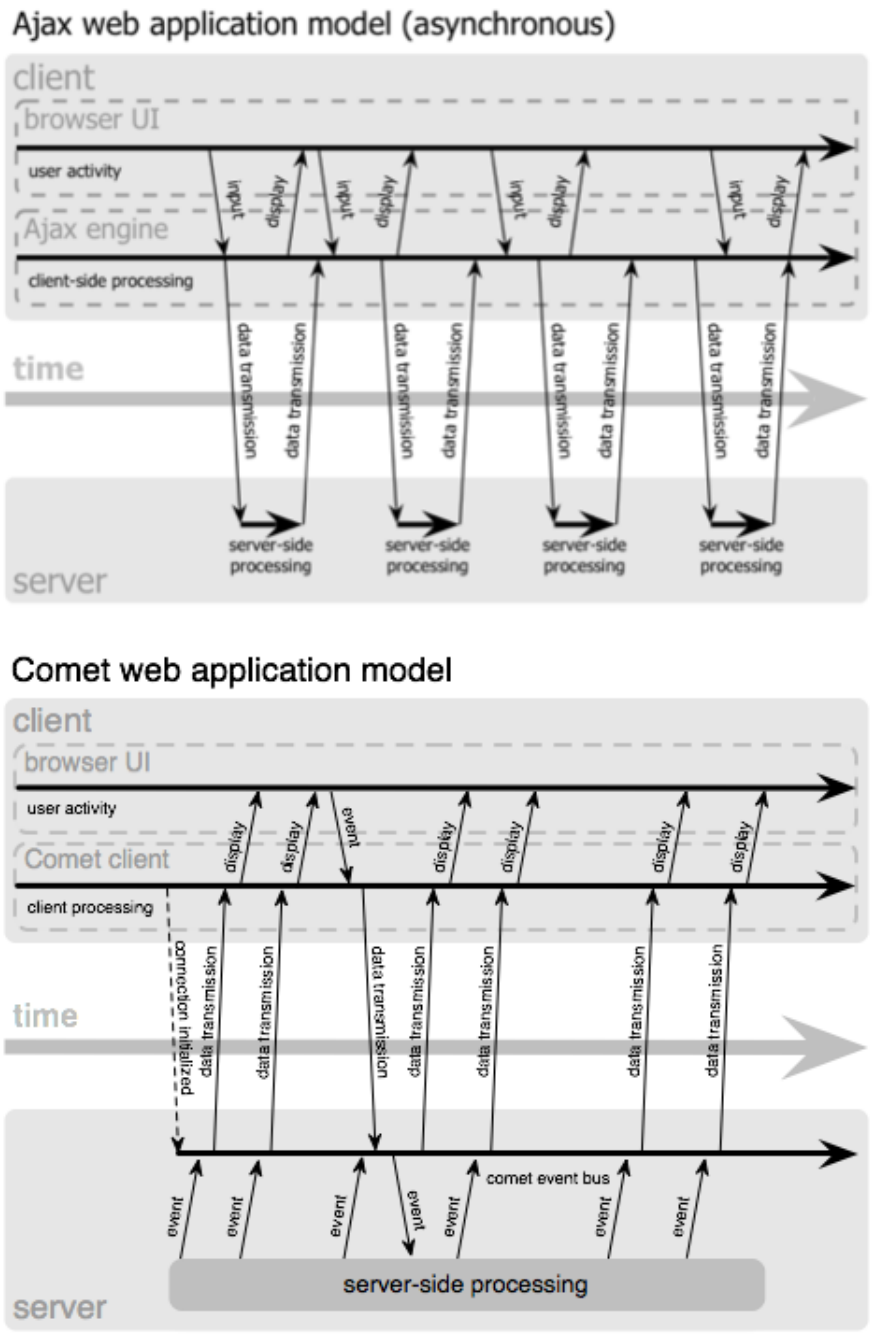

Figure 1.2: Alex Russell's illustration of the differences between Ajax and Comet [13]

HTTP streaming One of the oldest forms of push via HTML and JavaScript is HTTP streaming, mentioned above, which comes in one of two main forms. The original is page streaming via a hidden inline frame (`<iframe>`) element, embedding a second HTML document inside the first. This inner document consists of *chunked* data, and as such is not closed until the server chooses to end the connection. It requires no scripting to initialise, but the server-to-client payload consists of either client-side code or a link to a server-hosted script file inside `<script>` tags, which obviously require scripting to be available in order to execute. Simple though hidden `iframe`s might be, they also have a few problems, one of which is simply that in all web browsers, the page appears never to finish loading, as the inner document is never ended. This can cause confusion, potentially stopping mainstream users from interacting with the website and therefore making the technique unusable.

With the advent of `XMLHttpRequest` and Ajax, the process of requesting additional information could be performed entirely in scripts, instead of altering the content of the page. By opening a long-lived Ajax connection, a client can receive data from the server in any form (for example, XML or JSON), parse them and act based on their contents. Because Ajax connections are notified through the `onreadystatechange` event every time the state of the response is updated, they can react instantly to new occurrences.

Long polling The alternative to streaming is more closely related to "normal" behaviour: a request is made and a single response is accepted and processed. However, upon receipt, a new request is made to the server which can receive more data. This practice of always maintaining a connection to the server, coupled with the fact that the server can delay responding if it so chooses, is known as *long polling*. Being closer to the traditional web model, it evidences less side effects and tends to be more reliable in older browsers than HTTP streaming. As such, it is more commonplace, and people speaking about Comet without specifying which method they are using are often referring to this one.

There are two methods of long polling. One uses `XMLHttpRequest`s to establish the connection, and the other requests a script via a `<script>` tag. Other than this, they work in the same way: the client requests updated information from the server via one of the two methods. When the server state changes, it replies with the update (which can be either data or code, in the case of the developer using an `XMLHttpRequest`, or only code, if he or she used a `<script>` tag) and the connection is closed. A new connection is created immediately, and another update is requested. Meanwhile, the data that has just been received is processed by the browser, either through

client-side code that parses and acts upon the data, or in the case of the data being executable code, by running it directly.

More information on Comet can be found in the technical report, "A Comparison of Push and Pull Techniques for Ajax" [14].

1.4.2 HTML5

Relatively recently, the web standards committees, including the W3C, started work on a new version of HTML: HTML5. The actual markup language itself bears no relevance to bidirectional communication, but the HTML5 specification does define a number of new scripting APIs. Two of these are especially relevant, as they provide a standard way to create long-lived connections between the client and the server.

Server-Sent Events

Part of the HTML5 specification defines an API for "server-sent events"—essentially, a standardised protocol for HTTP push. The implementation is fairly simple for web developers: simply create an `EventSource` object in JavaScript which connects to a page on the server. Following this, register an event listener—a JavaScript function which handles the data sent from the server as it arrives. The API takes care of the rest, leaving the developer with a very simple yet effective implementation:

```
1 var source = new EventSource('updates.cgi');
2 source.onmessage = function (event) {
3     alert(event.data);
4 };
```

The server output is also very simple. In this case, the text prefixed by `data:` is accessed through `event.data` in the code above.

```
1 data: This is the first message.
2
3 data: This is the second message, it
4 data: has two lines.
5
6 data: This is the third message.
```

These examples were sourced from the W3C specification [15].

Web Sockets

In the same vein as `EventSource` lies the `WebSocket` interface. However, whereas the former provides an easy mechanism for server-to-client communication, web sockets embody a full-duplex communications layer—both the client and the server can send information over the same connection. The client is notified of messages immediately through the `onmessage` callback, making real-time communications possible. While this is possible using Comet, having an API dedicated to the purpose should not only simplify the process greatly, but also standardise it, resulting in much greater browser penetration and eventually allowing developers to write client-side code without having to provide a fallback to what some would consider little better than a hack.

Due to the hugely different nature of web sockets when compared to other browser-based communications, data is not transferred over HTTP, but rather through a new protocol, conveniently termed the Web Sockets protocol. While it should make for more efficient traffic and simpler code in the long run, currently, implementations are scarce. The same is true of both the `WebSocket` and `EventSource` interfaces: as they are new additions to HTML, there are no browsers that support either in their latest stable versions, though progress is being made to implement them in future releases—a beta version of Google Chrome now supports Web Sockets [16], and a basic extension to the Apache HTTP Server entitled *mod_pywebsocket* has been created for testing purposes. Using these HTML5 features in any website is now possible, but hardly recommended, as less than a percentile of of web users will be able to use them. Consequently, if a developer chooses to design for server-sent events or web sockets, a fallback must be implemented in the form of a current Comet implementation.

1.4.3 Accessibility

As mentioned earlier, the entire GUI of an application with hot-swappable processes must be generated dynamically on the client, rather than running on the server. This can cause problems among certain users which have no easy solution. For example, the website *w3schools* recorded information that suggests that between five and ten percent of users do not have JavaScript enabled [17], which would leave the user staring at a blank page. An error message can be provided, but this is not very helpful for the average user, and even less so for those who deliberately disable JavaScript for security or performance reasons.

This reliance on JavaScript will also play havoc with users with accessi-

bility issues, especially those that have to use a screen reader or similar in order to read and interact with a page. In 2008, Aaron Cannon conducted a case study on how screen readers deal with JavaScript [18], and his results were troubling. Not only did JAWS or Window Eyes, two of the more prevalent screen readers, fail to process new information as it is triggered by JavaScript, but perhaps more importantly, they sometimes fail to trigger the required events upon certain user actions. Such behaviour could make a rich Internet application inoperable, alienating a not-inconsequential portion of the its user base.

Developing methods to minimise the risk of inaccessibility is beyond the scope of this project. However, it is worth noting that work is being done as part of the Web Accessibility Initiative in the form of the Accessible Rich Internet Applications project (*WAI-ARIA*). Its web page states that it provides authors with:

- Roles to describe the type of widget presented, such as "menu," "treeitem," "slider," and "progressmeter"
- Roles to describe the structure of the Web page, such as headings, regions, and tables (grids)
- Properties to describe the state widgets are in, such as "checked" for a check box, or "haspopup" for a menu.
- Properties to define live regions of a page that are likely to get updates (such as stock quotes), as well as an interruption policy for those updates—for example, critical updates may be presented in an alert dialog box, and incidental updates occur within the page
- Properties for drag-and-drop that describe drag sources and drop targets
- A way to provide keyboard navigation for the Web objects and events, such as those mentioned above

WAI-ARIA Overview [19]
Making Ajax and Related Technologies Accessible

1.5 Community-Driven Development

Open-source software has had a long history of community-driven development. Often, patches are introduced not by the developers, but by irritated

users who simply want them in the program now, rather than in a year's time. However, whether those patches are ever officially accepted and folded into the project or not is entirely at the discretion of its owner, who can sometimes be indifferent or at odds with the contributor in question. Forking the project is always a possibility, but requires a lot of maintenance, as changes to the original usually have to be made in the fork too.

An interesting solution to this problem of diverging interests is perhaps most evident in the Mozilla Firefox web browser. Instead of introducing features as the community demands them, the actual browser is kept quite lean. Missing or additional functionality is not introduced to the Firefox source at all, but rather developed as an extension which the browser loads at runtime. Not only does this mean that the experience can be customised on a per-user basis, but it has also fostered a huge community of developers who, between them, have created tens of thousands of add-ons for the browser, ranging from simple teaks to existing functionality to large-scale programs in their own right [20]. In this regard, the browser has become a form of operating system in itself. Some of these add-ons have inspired changes in Firefox itself, and still others have been merged directly into the source code, with only a few tweaks. In these cases, the software has been positively influenced by the community, with add-on developers fulfilling a need that was not taken care of or possibly even recognised by the project organisers.

One of the goals of this project is to create a way for users to contribute to web applications, fostering growth and using the power of the community to build new features, all without direct involvement of the original application creators. To this end, the ability to create and modify processes must be provided, potentially in the form of other "core" processes, deploying them on either the client or the server as necessary, and providing a means for any user to customise the application to their own desires, rather than bowing to the whims of a developer that may not have considered their use-case at all.

In the case of our email application, the users may find that the developers have not considered all the ways in which people use email. Some treat their inbox as their calendar, others as their to-do list, and still more use it to run their businesses. Some aim for "Inbox Zero", filing everything away in the correct folders, and others take the approach fostered by Gmail, letting the search functionality take care of finding a specific message. It would be very difficult for the application creators to cater for every use case, but by allowing the very same users to create plugins that take care of the specific piece of functionality they need, they can diversify their potential market while still focusing on their core competencies.

There are, unfortunately, security implications in letting users create server-side processes. Because they are not user-specific, they cannot be

turned off on a per-user basis—there is only one instance of the process. Consequently, malicious users could create or modify processes that interact with all users in order to diminish or destroy the user experience. As the processes are running on the server, they could even access the database in order to retrieve sensitive information and relay it back to the creator. However, this is not to say the feature is useless: this is a proof-of-concept, and while it might allow any user to edit processes, other frameworks based on the same ideas would be perfectly within their rights to be more discerning, only allowing trusted users to create server-side processes. In addition, there is an upside too: by providing functionality to build the processes in the first place, any one developer can introduce a feature to an application which directly improves it for a large subset of its users, potentially driving it forward faster than even its own development team could hope to imagine.

Chapter 2

Design

2.1 Introduction

The ultimate goal of this project is *Listen*, a software framework which allows a developer to easily create a web application. Each application is built by connecting independent processes on both the client and the server using a messaging infrastructure that abstracts away the separation between the layers—messages can be sent to any process, including those on different servers or even in entirely separate applications.

2.2 Messaging

As explained in the introduction, the messaging architecture is based upon the publish/subscribe paradigm. Each and every process can subscribe to a topic through the requisite message broker, and any message published to a topic will be delivered, barring communication errors such as a dropped network connection, to every process that has subscribed to it.

The architecture exposes an API that abstracts away the concept of clients and servers, such that any process, anywhere, can publish a message that will reach all subscribers of the message topic. The delivery mechanism should not be determined by the process, but by the broker, depending on the topic. Assuming an object-oriented paradigm, every process is initialised with a reference to its local broker, and can simply call the `publish` method, providing the destination topic and payload as parameters.

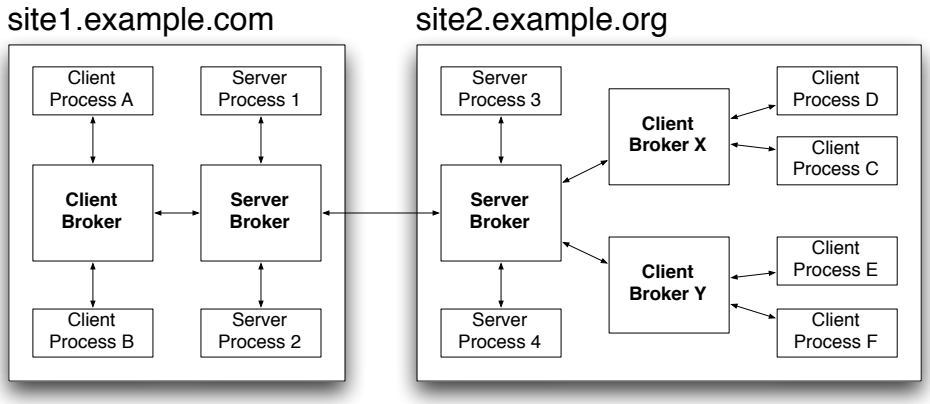

Figure 2.1: Example architecture diagram

2.2.1 Message Structure

A message is simply the combination of a destination topic and payload data. The data is encoded in JSON, a standard interchange format, rigorously defined [21] and natively produceable and parseable by both Python and JavaScript.

To avoid duplicate messages, a *Universally Unique Identifier* (*UUID*) [22] is attached to each message data payload under the key "__id". This is used to track messages and ensure that when a client delivers a message to the server, if it receives the same message back, it can be immediately dropped. When the server handles a message, it also attaches its own hostname and port (under "__host" and "__port" respectively) so clients on other servers can distinguish topics between servers.

2.2.2 Topics

Topics are hierarchical: they can be nested, and subscribing to a topic that is an ancestor of others will result in all messages addressed to any descendant being received. For example, `/messaging/text` and `/messaging/photo` are separate topics and can be subscribed to and published separately, but subscribing to `/messaging` will result in the process receiving all messages addressed to all three topics, as well as any other descendants of `/messaging`.

With a hierarchical nature, topics can easily be represented as URLs. Aside from helping people understand them, this also provides some useful qualities: primarily, the ability to reference topics on other domains. For example, in an application located at `site1.example.com`, the topic

`/messaging/text` is equivalent to `//site1.example.com/messaging/text` or even `http://site1.example.com/messaging/text`. However, on another server, using `/messaging/text` refers to the local topic. To communicate with `site1.example.com`, the process needs to qualify the topic path with the domain name, publishing or subscribing to `//site1.example.com/messaging/text`.

Topics are maintained in a tree structure in the brokers, with subtopics acting as branches of their parents. For example, the topic `/example` is treated as the parent of `/example/one` and `/example/two`. This makes it easy to deliver deeply-nested messages, as one simply needs to break the topic into segments and traverse the tree accordingly, delivering to each subscriber registered with every node of the tree on the way. If a node does not exist before a process attempts to subscribe to it, it is created.

Given a system laid out in the form portrayed in figure 2.1, the following gives an example of how such a system would convey messages. For a visual guide to the finished subscription structure, refer to figure 2.2 and figure 2.3, which show the topic hierarchy as a tree, with subscriptions in set notation beside each node.

1. **Server Process 1** subscribes to the topic `/test/one`
2. **Client Process A** subscribes to the topic `/test/one`
3. **Client Process B** subscribes to the topic `/test/two`
4. **Server Process 1** publishes a message to the topic `/test/one`
   It is delivered to **Server Process 1** and **Client Process A**
   It is *not* delivered to **Client Process B**
5. **Server Process 3** subscribes to the topic `/test/one`
6. **Client Process C** publishes a message to the topic `/test/one`
   It is delivered to **Server Process 3** only
   `/test/one` is separate on each server
7. **Server Process 2** subscribes to the topic `/notthetest/other`
8. **Client Process D** subscribes to the topic `//site1.example.com/test/two`
9. **Server Process 3** publishes a message to `//site1.example.com/test`
   It is delivered to all subscribers of both `/test/one` and `/test/two`: **Server Process 1**, **Client Process A**, **Client Process B** and **Client Process D**

10. **Client Process E** subscribes to the topic `//site1.example.com/test`

11. **Client Process F** subscribes to the topic `/test/two`

12. **Client Process B** publishes a message to the topic `/test/two`
    It is delivered to all subscribers of `/test` and `/test/two`: **Client Process B**, **Client Process D** and **Client Process E**
    It is not delivered to **Client Process F**, as **F** has subscribed to `/test/two` on `site2.example.org`

As URIs, topics should follow the syntax laid out in RFC 3986 [23]. In addition, they must be in one of the following formats:

- `http://example.com/path/to/topic`
- `//example.com/path/to/topic`
- `/path/to/topic`

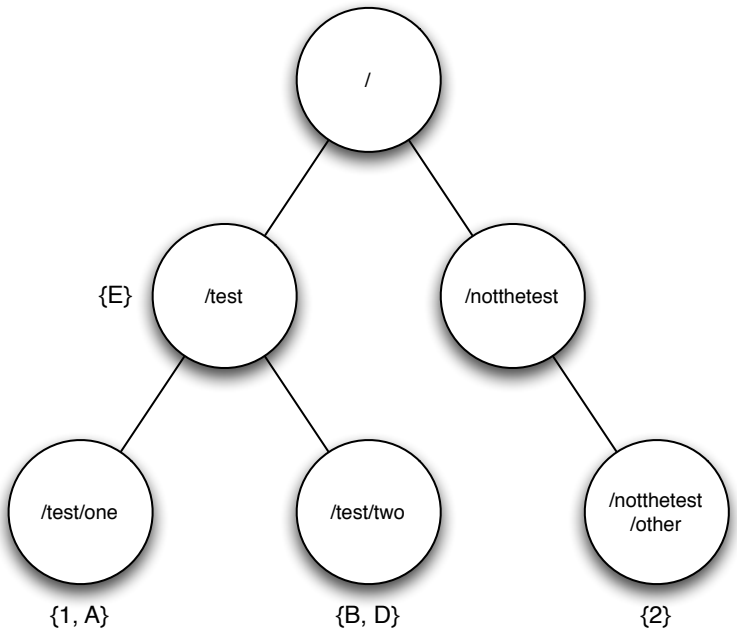

Figure 2.2: The final server broker subscription structure on `site1.example.com`.

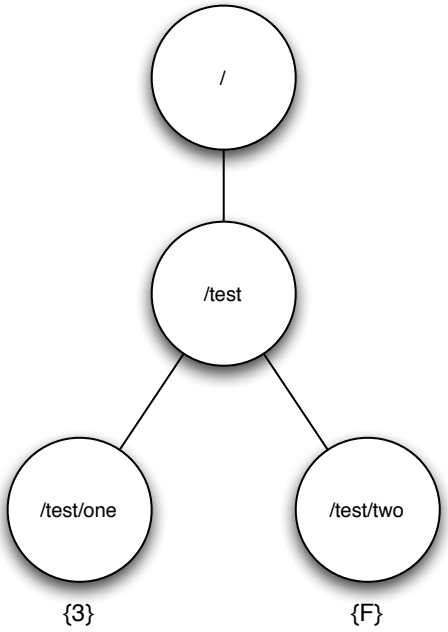

Figure 2.3: The final server broker subscription structure on `site2.example.org`.

2.2.3 Brokers

For each application, there are (at least) two brokers: one running on the server, and one running on each client, inside the web browser. Processes receive a reference to the broker local to their system on startup, and can use it to publish messages and subscribe to topics.

Each broker maintains a tree of topics as described in subsection 2.2.2. At each node of the tree, it stores a set of subscriptions to that particular topic, which is altered whenever a process subscribes or unsubscribes. Then, when a message is published, the tree is traversed according to the topic, and at each step, all subscribers are sent a copy of the message. On the server, these subscribers can be processes running on the server itself, or clients or other applications on behalf of their processes. However, each client only keeps a record of local subscribers. As all messages (aside from messages explicitly marked as local to the client) are delivered to the server for forwarding, recording subscribers from other systems is unnecessary, as this information will simply be a duplicate of that on the server.

The client must also notify the server when the subscription status of a topic changes. To avoid unnecessary messages, it only sends a subscription

request when a particular topic gains its first subscriber, and conversely, only sends an unsubscription request when it loses its last. Servers sending subscription and unsubscription requests to other servers behave in the same way.

2.2.4 Special Topics

All forms of specialised messaging can be implemented as a special form of topic. We have opted to use certain prefixes to denote specific actions to the broker. This allows us to transfer messages between brokers (especially between the client and server) and easily parse them. As the special prefixes are all symbols, they should not interfere with normal use.

The prefix being used should form the first part of the topic in question. For example, when subscribing to the topic `/example/topic`, the topic should be of the form `/+/example/topic` or `http://example.com/+/example/topic`.

Subscription (+) The + prefix represents a subscription. This is unlikely to be used by the processes themselves, as for clarity, there should be dedicated methods to handle this.

Unsubscription (-) Similar to +, the unsubscription prefix, -, represents a process unsubscribing from a topic.

External Subscription (++ and --) To allow brokers to easily differentiate subscriptions inside the same application from those to external sites, two + or - characters should be used when connecting to external sites. This is to avoid ambiguity: `/+//example.com/example/topic` and `/+/example.com/example/topic` look very similar, but only the first is an external subscription: the latter just uses "example.com" as the first segment of the topic. Using `/++//example.com/example/topic` makes this more obvious.

Server-only messaging ($) There may be occasions where messages only need to be delivered to the server and not the clients. In these cases, the $ prefix should be used.

Client-only messaging (^) Sent from a client, the messages should only be delivered to subscribers on that client. Sent from a server, the messages should be delivered to all clients of that server, but not subscribed processes on the server.

Direct client messaging (`~`) In the case where messages must be sent directly to one client, the `~` prefix can be used. Each client has a public-facing identifier (explained in section 2.3), which can be used to specify the intended recipient. For example, to send a message to a client with an ID of `f47ac10b-58cc-4372-a567-0e02b2c3d479`, the topic would be `/~f47ac10b-58cc-4372-a567-0e02b2c3d479/example/topic`.

2.2.5 Security

It was originally intended that the message broker itself did not currently limit either publishing or subscribing—any process, on any domain, should theoretically have been able to work with any topic. However, after writing applications for the framework, it became clear that limiting messages to specific recipients would be necessary at some point. With this in mind, the server- and client-only messaging prefixes, as well as direct client messaging, was introduced. Using these, messages with sensitive data can be restricted to only trusted recipients, and the brokers will ensure that no other process, benign or not, will ever see their contents.

2.3 Clients

While there is only one server instance per application, there can be an unlimited number of clients—one for each browser window open. Each client can be running any number of processes, and each one has a broker running.

Clients sending messages to the server simply establish an AJAX-style connection, using the topic as the URL, and transmit the message. Servers connecting to other servers work in the same way. The client opens an HTTP POST request to the topic in question. For example, to publish to the topic `/x/y/z`, a connection is opened to `http://example.com/x/y/z`. Special topics—subscriptions, etc.—receive no special treatment in this situation. The message data is encoded as JSON and passed in the request body. Only one message is sent at a time, as they are delivered by the client-side broker as soon as the message is published. The server-side broker, which runs within the web server, then distributes the message to every subscriber, including other clients via the mechanism described above.

In rare cases, messages may be sent at a rapid pace, making establishing an HTTP connection for each one very expensive. This is handled automatically by the broker—if the rate of messaging becomes too high, it will switch to another mechanism, connecting to the special topic `/_` and sending a group of messages all at once. The messages are formatted such that each

takes up two lines, the first being the topic and the second, the data. They are then concatenated together, separated by a new line, and sent. This proved to dramatically decrease the bandwidth and processing power needed to send a large amount of messages in a short space of time.

To aid the server and other clients in identifying clients and directing messages to the correct recipient, each client possesses a public ID, sent with every message, as well as a private ID, known only to itself and the server. They are allocated by the server using cookies. These identifiers are UUIDs [22], randomly generated according to the UUID 4 specification. As there are $2^{122} = 5.3 \times 10^{36}$ possibilities, given sufficient entropy and a uniform distribution in the random number generator, to say that clashes are extremely improbable would be an understatement.

Initially, our plan was to use a single ID for everything. However, when direct messaging was introduced, it became clear that there would need to be two. The private ID is used by the server to match up browser requests with its in-built representation of clients. If it were made public, one client could disguise itself as another very simply, and so another public ID was created, which processes can use to send direct messages.

2.4 The Web Server

In order to handle the interactions required by this new messaging architecture, we needed to develop a server capable of parsing and delivering them to the correct recipients. A number of possibilities were considered, but our chief need was for a quick development schedule that freed up time for experimenting as much as possible with the framework. We therefore decided to create a complete web server.

Python 3 was our final choice for this, as it has various classes and functions in its own standard library that make it very easy to develop a program handles HTTP requests. While not backward-compatible with Python 2, there is a helpful conversion tool that migrates Python 2 scripts to Python 3 fairly effectively. In addition, there are a number of improvements made in Python 3 which directly affect the creation of the web server: most notably, the revamping of the `http.server` and `urllib` modules, which provide excellent support for serving web pages and dealing with dynamic URLs.

Along with the server, we have also provided a command-line script that will execute it. The configuration file that accompanies execution can be specified on the command line, and the server can be closed simply by pressing `Ctrl+C`. As this particular server is a proof-of-concept, it cannot be run as a daemon.

2.4.1 Loading processes

While the developer can specify processes to be loaded by the server explicitly through the configuration file (covered in subsection 2.4.4), ideally, this should be automatic. To this end, the server will enumerate all Python files in the *application/server* directory and all JavaScript files in the *application/client* directory. The JavaScript files are easy to process: the list is simply sent as part of the index page and the browser requests each one individually. Because of the way JavaScript works, each process can be injected directly into Listen and run immediately.

Python works a little differently. First of all, each process exists as a class in a separate module, which must be loaded explicitly. Unlike JavaScript, which has no namespaces, Python maintains a strict separation between modules, and so next, the class name must be discovered. Unfortunately, due to Python's affinity for duck typing (if an object looks like an instance of a type, it should be treated as such—subclassing is not strictly necessary), iterating through the module's classes and finding one that subclasses the `ServerProcess` class is not an option. Instead, we look for a callable object (either a function or a class) with the `message`, `start` and `stop` functions—the basic definition of a process. This is still not perfect, as the aforementioned callable object could return a new instance of a process without actually *being* the process, but it is impossible to tell without calling the object, which may cause unwanted side effects. Once the process is found, we simply instantiate it.

2.4.2 Static files

The web server firstly handles static files such as the process code, as well as HTML and CSS required for a web browser to render the application. These files are delivered upon request: when the user directs his browser to the root address of the application (`http://example.com` or similar), an HTML page is served. This page is customisable by the developer, but by default it has no body. The head contains the page title (the name of the application), and links to the application JavaScript and CSS files that contain the framework code and the application processes and styles. The paths of the files are not the actual locations, but virtual paths that the web server recognises as being "special". When the browser requests to download each one of these, the path is interpreted and the correct file is delivered.

2.4.3 Messaging

Because of the nature of HTTP, transmitting messages from the server to the client is more complicated than moving them in the other direction. The client first establishes a long polling request over HTTP GET (described in subsection 1.4.1) to the special topic `/_`. This connection is kept alive by the server for a maximum of 60 seconds, during which it can be used to convey a single message from the server to the client. After the message is sent, the connection is immediately closed. After closing the connection, the client establishes a new connection and the same process is repeated.

When transferring a message in this manner, the topic is unknown by the client, and so cannot be represented in the HTTP headers. The response body therefore starts with the topic, followed by a newline, and then the data encoded as JSON.

As it is possible for messages to be sent in the intervening time between the closing of one connection and the opening of the next, the server keeps a record of clients that have recently held a connection. Any messages sent to clients with recently-terminated connections will be held in a message queue for a maximum of 60 seconds, and upon reconnection all messages will be delivered in a single batch (in the same format as the client when delivering more than one message at the same time), after which the connection will be closed and the cycle will begin again.

2.4.4 Configuration

Because of the nature of applications, the server is, to an extent, configurable. A sample *application.config* file (shown in listing 2.1) is provided which demonstrates to the developer the potential options, but they are also listed here.

Application name The title of the application, delivered to the web browser

Version The application version, currently unused

Host and port The application network location, so it knows how to reference itself

Virtual paths Alterations in paths, so the directory structure of the application can be reorganised

Processes A list of processes to either disable or, if processes are disabled by default, enable

Libraries Client-side libraries to include so processes can benefit from them (jQuery is provided as an example)

Stylesheets Application stylesheets, so clients do not have to use JavaScript to style elements

```
1  <?xml version='1.0' encoding='utf-8'?>
2  <config>
3      <name>Example</name>
4      <version>1.0</version>
5
6      <host>localhost</host>
7      <port>80</port>
8
9      <paths>
10         <path virtual="application" physical="application" />
11         <path virtual="system" physical="system" />
12     </paths>
13
14     <processes>
15         <server enabled-by-default="true">
16             <process name="example-disabled" enabled="false" />
17         </server>
18         <client enabled-by-default="true">
19         </client>
20     </processes>
21
22     <libs>
23         <client>
24             <lib name="jQuery" url="http://ajax.googleapis.com/
                   ajax/libs/jquery/1/jquery.min.js" />
25         </client>
26     </libs>
27
28     <stylesheets>
29         <stylesheet name="Stylesheet" url="/application.css" />
30     </stylesheets>
31 </config>
```

Listing 2.1: A sample application configuration file.

2.5 Processes

As detailed in the background, processes are *actors*: independent entities that maintain separate state from every other and communicate via messages.

Processes embody a task in the application: they are designed to handle one part, and allow the others to deal with the rest.

On both the server and the client, an interface is provided. Application developers can implement these interfaces in order to create processes which can be immediately deployed to an application server. On the server, the interface has been written in Python. For compatibility reasons, the client interface has been written in JavaScript.

All server processes are launched by the application on startup, and run until disabled or for the lifetime of the application. Similarly, all default client processes (as opposed to community-created ones) are launched by the client-side portion of the application as it is loaded by the browser. Every process is initialised with a reference to the local broker, which they can use to publish and subscribe to topics.

Client-side processes are responsible for generating the user environment. Rather than have the server generating HTML, as is commonplace among traditional web servers, the browser is served an HTML page with an empty body, and processes can act in tandem in order to create a user interface through DOM manipulation.

2.5.1 Dependencies

Sometimes, processes will depend upon other processes before they can start. This can be specified on both the client and the server by assigning the static `dependencies` field to a list of process names. If this is the case, both the server and the client will attempt to load the processes in the correct order. If a dependency that does not exist is specified or a dependency cannot be loaded, all dependents will also fail to load.

2.5.2 Community-Driven Processes

Processes cannot only be created by the application developers; users of the application can create *external* processes for the client, uploading them as source code to the server to be stored and run. These client processes are simply added to a list of optional modules which users can turn on and off.

With users able to enable and disable processes at will, the application developer must have some control over his or her own *internal* client processes. By simply omitting the functionality that stops the process, he can signify to the application that the process is critical and should not be turned off.

To allow developers to start working with the framework straight away, the *Switcher*, *Editor* and *Logger* processes are distributed with the frame-

work. These processes should provide basic functionality that will be useful in almost all applications. They are optional and replaceable, but easily enabled, and work straight "out of the box." Together, they provide the user with complete control over processes, allow them to monitor the messages that are being sent and received, and allow them to create and run new ones.

Chapter 3

Using the Framework

While we have attempted to make the Listen framework as customisable as possible in the short period of time we have had to create it, more important was to make it as *easy* to use as possible. To this effect, developers can jump right in and start making applications with only minimal configuration.

The only required pieces of information are the application title and version, which can be changed as often as is necessary, and the web server hostname and port. These are required so the client can contact the server, as well as to aid in inter-server routing of messages. After setting this information in the *application.config* file, which is provided complete with example information, the developer is free to create processes on both the server and the client.

Creating processes on the server is somewhat different from creating them on the client, and so we will look at both individually.

3.1 Server Processes

Processes on the server are persistent: they launch as part of the server initialisation and run until the server is terminated. Server processes are created in their own thread. Listing 3.1 shows a basic server process, which we will use to demonstrate the possible functionality.

The aptly-named `begin` method is the first we shall examine. In this method, we initialise all variables that we will need throughout process lifecycle, and subscribe to any topics necessary for the process to operate correctly. In this case, we subscribe to just one: `/primes/request`. We will use this topic to request the next prime number from the service. To do this, we use the `subscribe` method. Alternatively, we can use the `broker.subscribe` method, but the shortcut is provided for convenience. Subscription takes a

```
1  import system.server.process
2
3  class Primes(system.server.process.ServerProcess):
4      NEXT_PRIME_TOPIC = '/primes/next'
5      REQUEST_TOPIC = '/primes/request'
6
7      def begin(self):
8          self.next = 2
9
10         self.subscribe(Primes.REQUEST_TOPIC)
11
12     def message(self, topic, data):
13         if topic.startswith(Primes.REQUEST_TOPIC):
14             self.publish(Primes.NEXT_PRIME_TOPIC, {
15                 'next_prime': self.next
16             })
17
18             self.next += 1
19             while not Primes.is_prime(self.next):
20                 self.next += 1
21
22     @staticmethod
23     def is_prime(n):
24         if n <= 1:
25             return False
26
27         for i in range(2, n):
28             if n % i == 0:
29                 return False
30
31         return True
```

Listing 3.1: A basic server process that publishes the next prime number on demand.

topic as a string—breaking it down, adding it to the correct list and notifying any servers is all handled by the broker. Unsubscription works the same way, using the `unsubscribe` method, which also takes the topic as its only argument.

It is worth noting that `begin` has a corresponding `end` method, which allows us to perform cleanup operations when the process is ended. This will prove especially useful for processes that are connection-bound, such as those that deal with databases or other network services, as it gives them room to close connections and clean up any loose ends.

Now we have subscribed to a topic, the most important method, and the only one that the developer is required to override, becomes useful. The `message` method is called, in the process thread context, every time a message is received on a subscribed topic, and passed the topic and data as a string and dictionary object respectively. It can then act upon the data, potentially sending more messages if it deems them to be necessary.

As we can see, the `message` method first checks the topic in order to decide its execution. In this case, it is fairly pointless, as we are only subscribing to the one topic, but in the event of multiple subscriptions, which are far more common in larger applications, it becomes necessary. Due to the tree-like nature of topics, it is also recommended that we check the start of the topic and not the whole string, as messages targeted at subtopics should also be handled in the same way.

Once the topic has been determined, this particular process publishes a message. This is handled by the `publish` method, which takes two parameters—the topic and the data, or message payload, which should be in the form of a dictionary. If it is not, it will be encapsulated inside one, with the special key "__value" holding the original data. The *Primes* process publishes a single message to `/primes/next` with a dictionary holding a single value: the next prime number. After doing this, we calculate the next prime number using a very simple (and slow) algorithm. After all messages have been processed, the thread is put into a waiting state until more messages are received, at which point it will be woken up by the broker.

Developers may wish to perform additional tasks without waiting for a message. Fortunately, it is fairly easy to spawn a new thread in Python and have it execute a function. By placing the operation necessary in a loop that stops when the process is ended, and sleeping for a certain period between each iteration, it can be run in the background to perform maintenance, send messages on a schedule or fire an event based on a countdown.

A more traditional mechanism might be a PHP page that, when requested, responds with the next prime number. Aside from requiring some sort of database to save state between requests, which our example would

also need if it wanted to persist after the server is shut down, it would also require some way of recording the last number each client had requested. Conversely, because we are broadcasting the number, we can assume that all clients receive all numbers without any additional work. As can be seen from the code sample, our processes can perform a range of very complicated tasks in a way that is easy to understand, making it much easier for developers to create message-based applications in very few lines of code.

3.2 Client Processes

Client processes function in a similar way, though the code used to create them is somewhat different due to the different nature of JavaScript. Listing 3.2 shows a client counterpart to the server example explained above, which requests the next prime number every second and waits for a response, then prints it at the top of the web page.

Like the server process, the client provides `begin` and `end` methods to allow the process to start up. Notice that unlike the server, we have produced an `end` method that completely cleans up everything we may have produced. This is because unlike the server, users can activate and deactivate processes on the client whenever they choose, without reloading the page in their browser. They therefore need to be able to switch themselves off at any given point.

The `subscribe`, `unsubscribe` and `publish` methods work just as they do on the server. As an aside, we define `self` as `this` at the top of the process definition to get around some of JavaScript's quirks—because objects are not class instantiations, we can get some unexpected (but rational) behaviour if we use `this` inside nested classes. We should therefore use `self` to access the three broker methods, just as we do on the server.

Our process starts by subscribing to `/primes/next` so it can receive prime numbers as they are published. It then creates a container element to which it can add numbers without disturbing any other processes. Finally, it sets the `requestNextPrime` method to execute every second. This will publish an empty message to `/primes/request`, causing the server process to push a new prime number down the pipe, which is received by the client's `message` method. After validating the topic and data, it adds the number to the top of the container. Assuming there are no other processes, this will also be the top of the web page.

One thing that seems missing from the client is the ability to spawn worker threads. If the developer wishes for a function that executes on a timer, we can use the native `setInterval` function, as we see in the example, to register

```
1  Listen.Processes.Primes = function() {
2      var self = this;
3
4      var NEXT_PRIME_TOPIC = '/primes/next';
5      var REQUEST_PRIME_TOPIC = '/primes/request';
6
7      var container;
8      var requestIntervalID;
9
10     this.begin = function() {
11         self.subscribe(NEXT_PRIME_TOPIC);
12         container = document.createElement('ul');
13         container.id = 'primes';
14         document.body.appendChild(container);
15         requestIntervalID = setInterval(requestNextPrime, 1000);
16     };
17
18     this.end = function() {
19         self.unsubscribe(NEXT_PRIME_TOPIC);
20         document.body.remove(container);
21         clearInterval(requestIntervalID);
22     };
23
24     this.message = function(topic, data) {
25         if (topic.substring(0, NEXT_PRIME_TOPIC.length)
26             == NEXT_PRIME_TOPIC) {
27             if (data.next_prime) {
28                 printPrime(data.next_prime);
29             }
30         }
31     };
32
33     var requestNextPrime = function() {
34         self.publish(REQUEST_PRIME_TOPIC, null);
35     };
36
37     var printPrime = function(prime) {
38         var element = document.createElement('li');
39         element.textContent = prime;
40         container.insertBefore(element, container.firstChild);
41     };
42 };
```

Listing 3.2: A basic client process that retrieves the next prime number from the server once per second and displays it to the user.

a function for execution on a fixed period. However, if we need a function to execute on each message, we can place the functionality (or a call to a separate function) directly inside the call to `message`.

With only a few lines of code, we have embodied a request and response. A large part of the code base on the client is, as we can see, concerned with displaying the data it receives to the user. Coupled with a library such as *jQuery* for presentation work, issues with client-side programming should become far easier to solve, and with an easy way to convey information back and forth between the server and the client, without worrying about AJAX requests and server-to-client messaging (see subsection 1.4.1 for more information), developers can quickly create useful applications. Rather than focusing on the mechanisms of transporting information, they can more easily see the bigger picture.

Chapter 4

Evaluation

Listen has three main goals. The first, and most important, is to provide a simple framework that anyone can use. Secondly, it must be extensible, allowing contributors to create extensions to applications the original developers would never have dreamed. Finally, it must be efficient and reliable: as the underpinnings of an application, the framework must continue to operate come hell or high water.

4.1 Usability

When designing the framework, a lot of care was put into making it as frictionless as possible. While a basic understanding of the concepts is required, there is no need to comprehend the inner workings of the broker, nor is it necessary to spend hours configuring the server to achieve optimum results. In this section, we will look at establishing a basic yet useful application. Usability is difficult to quantify, but we will attempt to analyse the final result in a manner that helps the reader decide upon the practicality of learning a new framework such as this.

4.1.1 The Application

The application we are going to demonstrate is well-suited to a messaging-based framework: it will be the simplest possible chat software. By the end, a user will be able to send a message and have it appear, along with his or her name, on the screens of every other user. In addition, new users will instantly receive a log of the last few messages so they can jump straight into the conversation.

4.1.2 Configuration

The first thing to do is configure the server. Unfortunately, it is not capable of running without any information whatsoever, but only a few, simple things are necessary.

```
1 <?xml version='1.0' encoding='utf-8'?>
2 <config>
3     <name>Shout</name>
4     <version>1.0</version>
5
6     <host>localhost</host>
7     <port>80</port>
8 </config>
```

Listing 4.1: Our `application.config` file.

The title and version are entirely up to the developer, and the host and port constitute the socket that users will use to connect to the application. This is necessary so we can communicate a return address when subscribing to another application. This may not be necessary for our purposes, but extension creators will need it.

4.1.3 User Interface

Constructing a user interface in Listen works in the same way as in any other JavaScript application, with one or two exceptions. Because we do not know how actors will behave until runtime, we cannot serve static HTML, but must generate it dynamically. This can be difficult and time-consuming when dealing with vanilla JavaScript, but fortunately, the framework provides full support for libraries which make the process much easier. In this example, we will be using *jQuery*, with which the reader should be familiar. The jQuery file should be placed in the `application/client/lib` directory, or a reference to an online version should be placed in the configuration file, as can be seen in listing 2.1.

The user interface can be divided into two parts: receiving messages, and sending them. We will deal with the former first, creating an interface for incoming messages.

Incoming Messages

```
1 Listen.Processes.Incoming = function() {
2     // Necessary to avoid JavaScript quirks.
3     var self = this;
4
```

```
5       // This is our HTML element.
6       // We will be placing incoming messages in here.
7       self.incomingBox = $('<div>').attr('id', 'incoming');
8
9       // This is called when the process is started.
10      self.begin = function() {
11          // Add the box to the document.
12          $(document.body).append(self.incomingBox);
13      };
14
15      // This is called when the process is stopped.
16      self.end = function() {
17          // Empty the box and remove it from the document.
18          self.incomingBox.children().remove();
19          self.incomingBox.remove();
20      };
21
22      // This function will be called by our message method later.
23      function addMessage(name, contents, datetime) {
24          // Use the current time if one isn't provided.
25          if (!datetime) {
26              datetime = new Date();
27          }
28
29          // Create the elements that will make up the message.
30          var timeElement = $('<span>').addClass('time').
31              text('[' + datetime.hour + ':' +
32                       datetime.minute + ']␣');
33          var nameElement = $('<strong>').
34              addClass('name').
35              text(name + ':␣');
36          var contentsElement = $('<span>').
37              addClass('contents').
38              text(contents);
39
40          // Add the message to the box and scroll to it.
41          self.incomingBox.append(
42              $('<p>').addClass('message').
43                  append(timeElement).
44                  append(nameElement).
45                  append(contentsElement)
46          ).scrollTop(self.incomingBox.height());
47      }
48  };
```

Listing 4.2: The first process, `incoming.js`.

This will create a box into which we can feed messages, and give those messages CSS classes so we can style them later. Next, we need to manufac-

ture a way of sending messages out.

Outgoing Messages

```
1  Listen.Process.Outgoing = function() {
2      var self = this;
3
4      // A text input element for the user's name.
5      self.nameBox = $('<div>').attr('id', 'name');
6      self.nameInput = $('<input>').attr({
7          'id': 'name-input',
8          'type': 'text',
9          'placeholder': 'Your name'
10         // "placeholder" only works in HTML5-compatible browsers
11     });
12     self.nameInput.append(outgoingInput);
13
14     // A text input element for the outgoing message.
15     self.outgoingBox = $('<div>').attr('id', 'outgoing');
16     self.outgoingInput = $('<input>').attr({
17         'id': 'outgoing-input',
18         'type': 'text',
19         'placeholder': 'Your message'
20     });
21     self.outgoingBox.append(outgoingInput);
22
23     self.begin = function() {
24         // Insert the name box before the incoming box.
25         $('#incoming').before(nameBox);
26
27         // Insert the outgoing box after the incoming box.
28         $('#incoming').after(outgoingBox);
29     };
30
31     self.end = function() {
32         // Clear any user input.
33         self.nameInput.val('');
34         self.outgoingInput.val('');
35
36         // Remove the boxes.
37         self.nameBox.remove();
38         self.outgoingBox.remove();
39     };
40 };
41
42 Listen.Process.Outgoing.dependencies = ['Incoming'];
```

Listing 4.3: The second process, `outgoing.js`.

This works similarly to the previous listing—it simply creates two more

`<div>` elements and places text input elements inside them. It then positions these boxes before and after the box created by our `Incoming` process. Interesting to note is the last line, which states that this process depends on the other. This ensures that it only loads if `Incoming` does, and that it will be loaded afterwards, so we can be sure that the incoming messages box exists before inserting our elements around it.

Stylesheets

Now we have our HTML set up, there is the small matter of styling it. This requires a change to the configuration file: we must add the `<stylesheets>` element, which can be seen in listing 2.1. Once added and altered to suit the application, stylesheets can be placed in `application/static` ad nauseam.

With that, we have a user interface which can be seen by running the server and browsing to the URL specified in the configuration file—in the case of our example above, `http://localhost/`. In a straight comparison with a traditional web application, the only real difference is the existence of the `end` method, a tradeoff we made in order to drastically improve the extensibility of the framework. The rest of the code would be very similar, demonstrating the freedom Listen provides to simply get things done without creating artificial barriers to progress.

4.1.4 Communication

Now we have a UI, we can start dealing with messages. The first thing to do is establish a mechanism for sending them. This is where the framework starts to become useful, encapsulating the complicated process of sending messages to the server via AJAX in a convenient `publish` method, which ensures the message is delivered not only to subscribed processes on the server, but all other clients. Because the broker handles this, we don't need to write any server-side code to distribute the messages—we just need to add one function to the outgoing process.

```
1     // Grab any user keypresses inside the input element.
2     self.outgoingInput.keypress(function(event) {
3         // If the key pressed is Return or Enter, send the
              message.
4         if (event.keyCode == 0x0D || event.keyCode == 0x0A) {
5             // Send the message to all subscribers.
6             self.publish('/messages', {
7                 name: self.nameInput.val(),
8                 contents: self.outgoingInput.val()
9             });
10        }
```

```
11      });
```

Listing 4.4: Sending messages (goes inside the `Outgoing` process).

In eight lines of code, we add an event to the input element that fires when a key is hit, and if the key is an Enter or Return, we publish the message to everyone listening.

Next, we need to display this message to all users. To receive it, we just need to add one line to the `Incoming` process' `begin` method: `self.subscribe('/messages);`. Once this is done, we can create a `message` method which handles them simply by calling the `addMessage` function we defined earlier.

```
1       self.message = function(topic, data) {
2           // Always check the topic to make sure it's the right
3           // one. Remember not to check it directly, as it could
4           // be a subtopic of the one we are interested in.
5           if (topic.substring(0, 8) == '/message')
6               if (data.contents) {
7                   addMessage(
8                       data.name || '<unknown>', // default name
9                       data.contents,
10                      // We will use the `datetime` attribute
11                      // later. It is simply the Unix timestamp
12                      // of the message.
13                      data.datetime ?
14                          new Date(data.datetime / 1000) :
15                          null
16                  );
17              }
18              break;
19          }
20      };
```

Listing 4.5: Receiving messages (goes inside the `Incoming` process).

The message brokers will take care of delivering the message—all we need to do is display it. Note that we should also create an empty `message` method in the `Outgoing` process, as Listen requires every process to have a message handler.

4.1.5 Logging

So far, we've been able to do everything on the client. However, some things are not possible or work much better as a server process. These include most request/response-based services and information delivery and storage services, such as a logging service, email delivery as discussed in chapter 1 or

even serving rich media such as images, sound and video. We will therefore create our logging service on the server as a Python script.

The service should work on a request/response basis, sending the last 10 messages out to anyone who asks. In order to do this, we will listen on the `/history` topic for clients and send a list of messages straight back. Simultaneously, we will be listening to messages and adding them to our log.

```
1  import time
2  import system.server.process
3
4  class Logger(system.server.process.ServerProcess):
5      MAX_LOG_SIZE = 10
6
7      def begin(self):
8          self.log = []
9
10         self.subscribe('/messages')
11         self.subscribe('/history')
12
13     def message(topic, data):
14         if topic.startswith('/messages'):
15             # Store the message arrival time so it can be
16             # accurately represented when re-sent.
17             data['datetime'] = int(time.time())
18
19             # Add the message to the log.
20             self.log.append(data)
21             if len(self.log) > MAX_LOG_SIZE:
22                 self.log.remove(0)
23
24         elif topic.startswith('/history'):
25             # Get the client's ID so we can send it directly.
26             # This saves every other client from having to deal
27             # with the messages.
28             client_id = data['__client_id']
29             if client_id:
30                 for message in self.log:
31                     # '/^' is a special topic that directs
32                     # messages to one specific client.
33                     self.publish(
34                         '/^/' + client_id + '/messages',
35                         message)
```

Listing 4.6: The `Logger` process.

We then put the request in the `begin` method of the `Incoming` process by publishing to `/history` after subscribing to `/messages`: `self.publish ('/history', null);` is all that is needed to make the request.

Listen is designed to make it easy as possible for services to send and

receive messages. With this functionality in place, it becomes very simple to quickly add a new feature to a web appplication that seamlessly slots in, enabling developers to quickly respond to their users' requests with minimum effort.

4.2 Extensibility

While responsive development is, from the users' perspective, always a good thing, a framework can only make it possible, not force it. To this end, Listen sports the capability for community developers to create their own client-side processes and add them to the application. The process is simple: in any application in which the *Editor* is provided and enabled, a user can click it at any point and bring up a text editor, which he or she can use to create new JavaScript "files", edit existing ones or delete them.

4.2.1 Basic Functionality

Practically, there is no difference between a client-side script created by the original developers of the application and one created by the community. All have access to the exact same APIs, are treated no differently by the broker and can send and receive messages in the exact same way.

To create an entirely new process, all the user has to do is open the Editor, create a new script, enter the code and save. It will be uploaded to a managing server process, which stores it in an SQLite database on the server, and then included in the HTML document header, which instructs the web browser to download and execute the script. It is not added directly to the document, as there may be a difference in execution if a script attempts to evaluate the code.

Unlike the main client scripts, which are loaded immediately, all user scripts are loaded on demand by the Switcher, which requests a list from the server as soon as it is initialised. When the requisite box is checked, the script is downloaded, the process is started and, assuming the code is correct, the user will experience new, community-created features after just two clicks. They can also disable scripts the same way. All enabled scripts are remembered using cookies, so that if the page is refreshed or the browser session ended, reloading the application will result in an environment that is optimal for the user.

4.2.2 Security

While any user is able to create a new process from scratch, they are also free to edit any existing community processes in the same way. Because the framework does not authenticate users in any way, it cannot discern the developer of an extension from any other user. As a result, all users are free to edit and delete any script.

This has profound security implications. Because scripts are run automatically by the client of anyone who has had them switched on in the past, a malicious user could alter a process trusted by others to steal confidential information or destroy data. All users of that process would then be vulnerable, with little to no recourse once impacted. This dangerous possibility means that the Editor, as it stands, must be taken as purely a proof of concept.

Certain features are necessary before the Editor should be allowed on a public-facing website. First of all, while there is no problem with allowing a user to view the source code of any community process—indeed, as JavaScript is an interpreted language, any user who downloaded the script would be able to read it—only the original developer should be able to change or delete it. Secondly, there must be an versioning mechanism, in which the server stores all versions of a process and serves the version last used by the user unless they choose to update. This is necessary, as a manevolent user would otherwise be able to create a useful process and then alter it to perform functions against a user's wishes.

Also useful, though not necessary, would be a comment or review system, so that users can explain to others any possible issues stemming from the lack of security in a specific process. Similarly, extension developers should be able to provide a changelog with each version to explain why it has been updated.

All of these features are built into the addon system in Mozilla Firefox, which pioneered the use of user-installed extensions on the World Wide Web.

4.2.3 Server Processes

Listen does not currently have any capability for a user to add or change a server process. This is partially due to the architecture of the server, which makes it very difficult to add new processes or terminate existing ones during execution. Because handles to the process are kept by the broker for messaging purposes, fully removing a process from the system would be very difficult. Reloading updated processes is also quite a complex task, due to the way Python handles classes.

Community-created server processes would also be a large security risk, even with all of the above in place. Due to the very nature of a server process, it has access to all the data on the server. One could therefore be used to misappropriate or manipulate information about all users, or even take down the entire server by performing an internal denial of service or attacking the file system. As there is only one instance running globally, users would not have the ability to enable or disable said processes, and so could not trust the application. These processes would be a huge liability, and while they would be interesting from an academic standpoint, allowing a user to run code on a live, public server would be disastrous.

4.3 Performance

In order to fully exercise the framework, a stress test was built. This simply consisted of a pair of processes which could be instructed to publish any number of messages, as well as subscribing to the same topic in order to receive them all. The purpose was not to test the fine control over messaging that the framework affords the developer, but simply to overload and potentially crash it.

To aid us in measuring the performance of the framework, the processes are fully configurable. We created a client-side controller that presented us with a form, allowing us to specify the number of messages per second and the test length in seconds. Additionally, the processes both had the ability to spawn any number of child processes, all of which send messages at the rate specified, and receive every message sent from every process. This results in exponential growth in the number of messages being received, hopefully putting an enormous amount of stress on the brokers in order to fully test their stability and performance.

To determine how the number of messages impacts performance, we ran the stress tests a large number of times, individually varying the number of server processes, client processes and messages per second. Each test was run for 60 seconds, during which we monitored the CPU usage on the server and the client nodes. We used five computers with Intel Core 2 Duo processors clocked at 3.0GHz and 4 GiB of RAM to run the server and four clients, all connected with Gigabit Ethernet to the same subnet. All clients were running a fresh copy of Google Chrome, version 5, which, at the time of writing, utilises the *V8* JavaScript engine, currently one of the fastest [24]. This is not an accurate representation of a standard web application, and should be far less intensive in the general case, but by increasing the number of messages being sent proportionally, we came away with excellent results.

Given that α is the number of processes on the server, β is the number of processes on the client, ϕ is the number of clients on the network and μ is the number of messages sent each second, we can calculate the number of messages sent and received across the network. The total number of messages being sent in one second is expressed as $\mu(\alpha + \beta\phi)$. However, because each message is broadcast to every process, the number of messages received in the same time, assuming no latency, is $\mu(\alpha + \beta\phi)^2$. A small increase in the number of processes can therefore place a large increase in load upon the brokers.

4.3.1 Varying the Server

Our first test measured CPU usage as we varied the number of processes on the server, keeping the number of client processes and messages per second constant. We can see from the results in figures 4.1 and 4.2 that increasing the number of server processes dramatically increases the amount of server processing, but does not affect the client at all. This implies that the actual messaging overheads are fairly small, and that the majority of processing time is taken up by the background threads actually sending the messages.

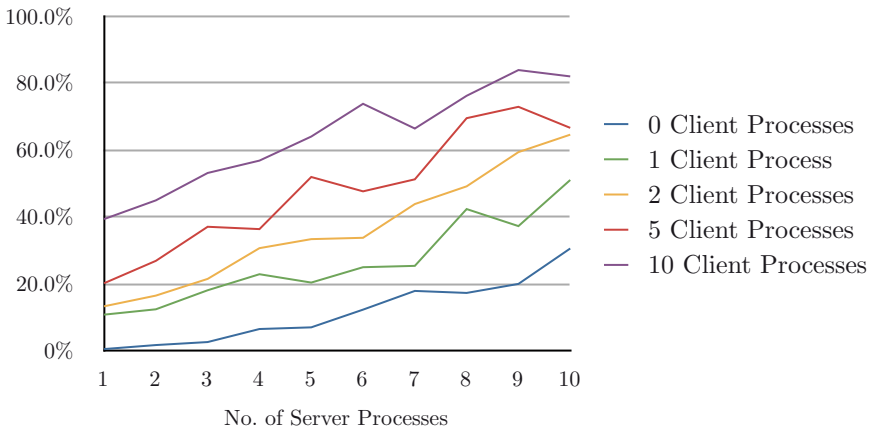

Figure 4.1: Server CPU Usage

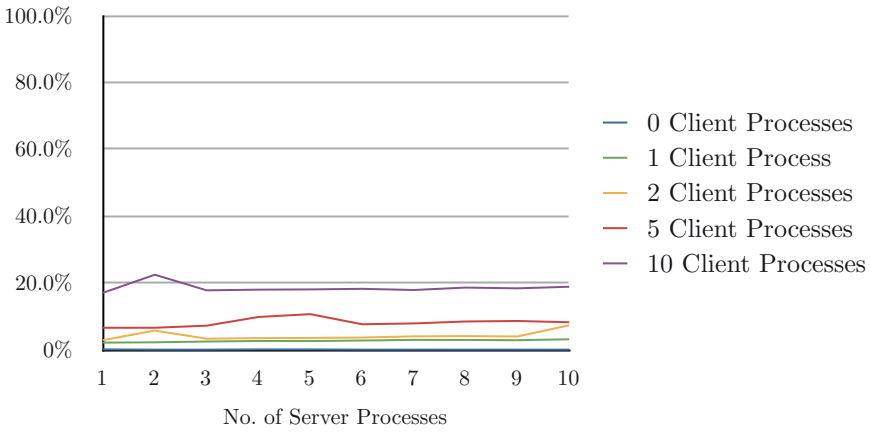

Figure 4.2: Average Client CPU Usage

4.3.2 Varying the Client

Measuring the CPU usage when varying the client further justifies our conclusions drawn from the above tests. From figures 4.3 and 4.4, we can see that there is only a very slight increase in usage as we increase the number of nodes, showing that the load is distributed fairly well across the network. Even at the peak, with 400 messages being sent every second, the server broker is coping fairly well, using only 30% of one CPU core to deliver all the messages.

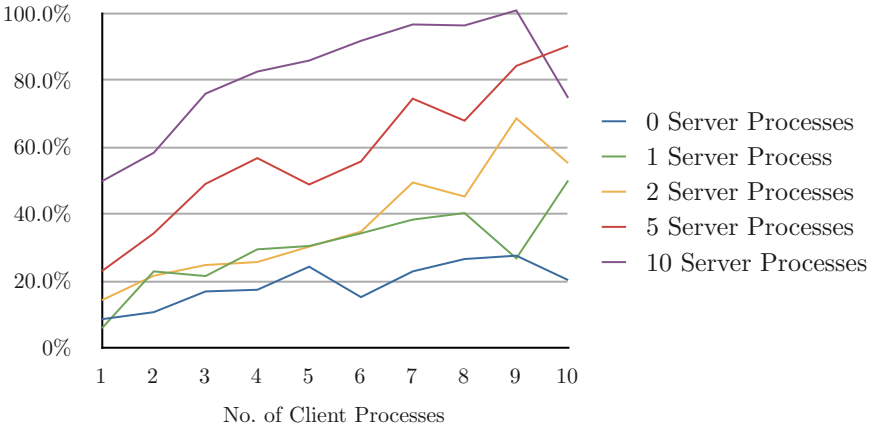

Figure 4.3: Server CPU Usage

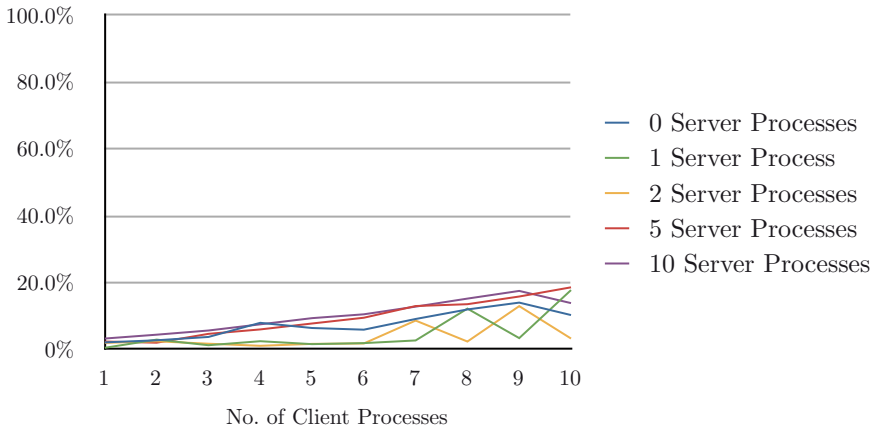

Figure 4.4: Average Client CPU Usage

4.3.3 Varying the Message Frequency

For the final test, the number of processes on the server and client were kept constant, and only one client was used. To achieve a useful result despite the linear, rather than polynomial, proportionality between messages sent and messages received, we increased the number of messages exponentially. Both the server and the client behaved as expected: when the server was running the test, the client was idle and the server's CPU usage increased steadily with the number of messages. However, when the tests were run on the client, both the server and the client experienced increase processing as the number of messages climbed. This is due to the differing behaviour of server and client: while the server will only forward messages to subscribers, the client will forward all messages to the server.

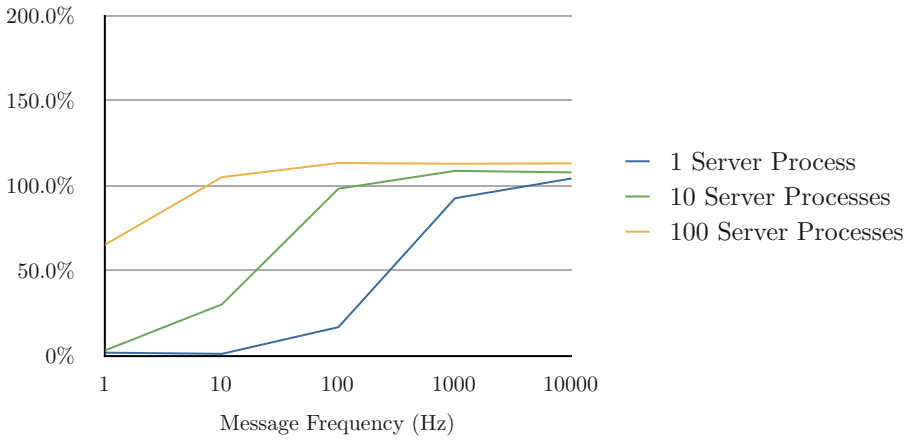

Figure 4.5: Server Only: Server CPU Usage

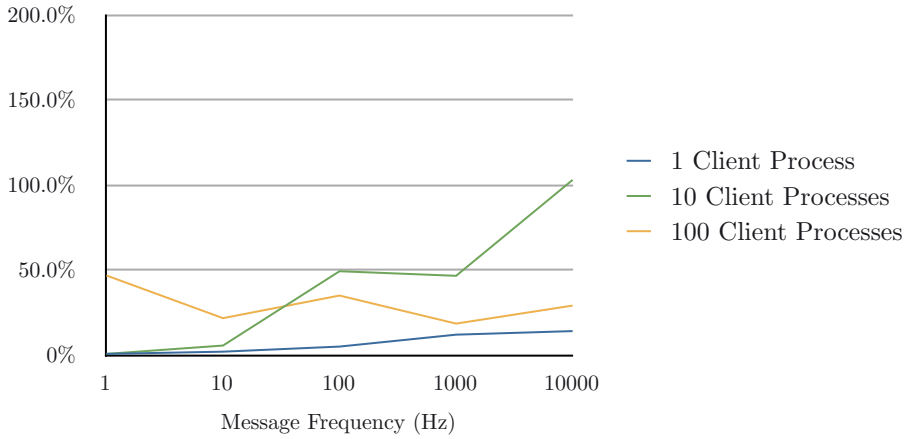

Figure 4.6: Client Only: Server CPU Usage

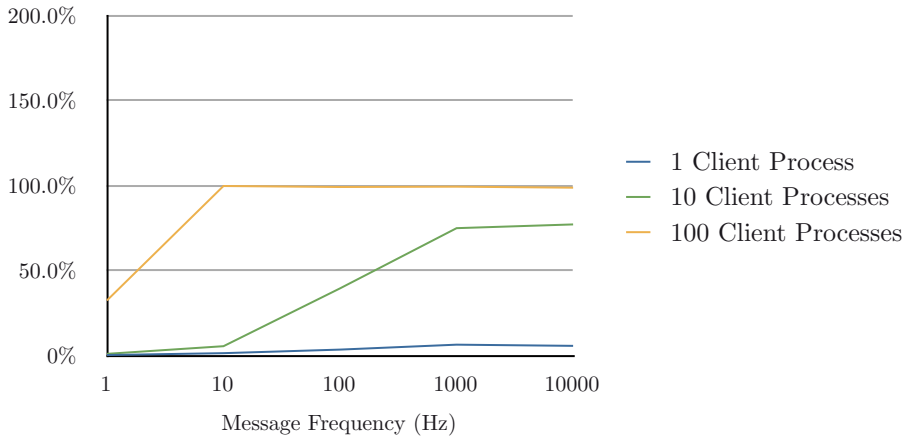

Figure 4.7: Client Only: Client CPU Usage

We also measured message delay: the number of messages being delivered within a second of being sent as a percentage of the total number of messages. In this case, the client was perfect up to approximately 100 messages per second, after which it took a slight decline. However, it hit a hard limit at approximately 238 messages per second when running one process, after which it could do no more. Increasing the number of processes also increased the limit, probably due to threading optimisations performed by the Google Chrome V8 engine. This hard limit seems to be the maximum that V8 can handle on the test hardware, perhaps due to minimum sleep times. While further optimisations to the code could possibly increase this limit, 238 messages per second should be far more than is necessary for any application.

Interestingly enough, on the server, increasing the number of processes actually decreases the total number of messages the server is able to send. This is probably due to the large number of threads necessary to handle the processes, resulting in a lot of processor time being dedicated to thread management and context switching rather than actual execution.

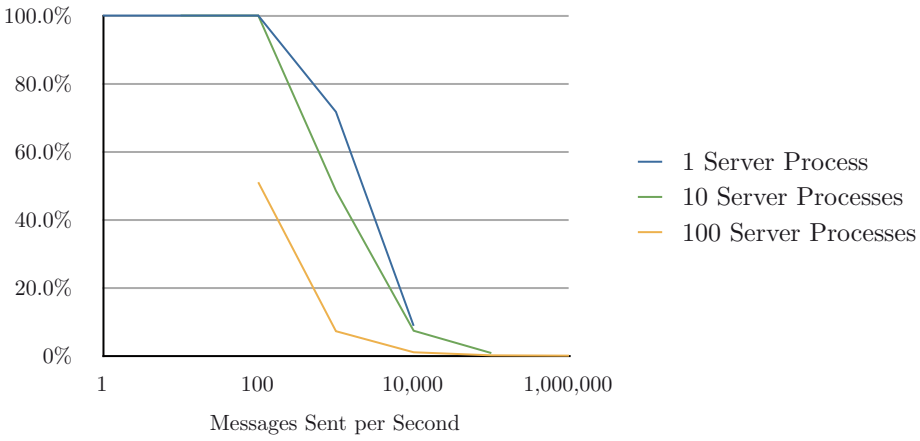

Figure 4.8: Server Only: Messages Delivered on Time

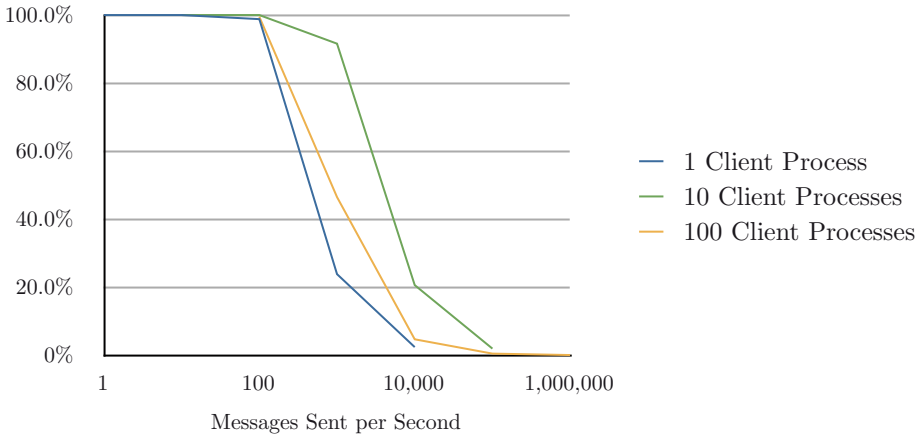

Figure 4.9: Client Only: Messages Delivered on Time

4.4 Reliability

A fortunate side effect of these stress tests, originally intended to measure performance, was that they initially broke the server. By driving it far beyond what could be expected of it, we found some interesting bugs that would have otherwise gone unnoticed. The majority were deadlocks, but there was also some code that relied on a there being a relatively small number of messages sent per second.

Originally, the client would deliver messages to the server as they were published, creating a new HTTP request and sending the data instantly. With large numbers of messages, this proved to be quite troublesome, as the act of establishing HTTP requests is both processor-intensive and time-consuming from a network perspective. This was therefore replaced with a new mechanism that waits a minimum of 200 milliseconds between each delivery, queueing up messages and sending them all at once if necessary. With this in place, the client is now able to handle far more messages than it could previously.

When attempting to send messages at a very high frequency, certain threads within the server would occasionally find themselves hanging. Processes would succumb to this due to a lock implementation which allowed the process creator to replace certain locks with a sleep mechanism so that a new thread would not have to be created in order to schedule functionality on a timer. However, this resulted in odd behaviour at certain points, and so it was removed in favour of a much simpler technique.

The broker was also hanging after prolonged periods of high stress. After much investigation, it was determined to be caused by multiple locks in both the broker, certain processes and the web server request handler all acting in tandem. The behaviour was erratic, with no way to guarantee reproducing it, and so could not be measured, but occured frequently enough at the higher ends of the scale that testing became difficult. The problem was somewhat ratified by replacing all the fine-grained broker subscription locks with one coarse lock, which greatly improved the situation and had no real effect on performance. However, there was still a small possibility of it freezing when sending upwards of a few hundred messages per second across the network—unlikely, but it would bring the entire system to a halt. The error was traced to the broker blocking when responding to client requests, which can potentially fail forever and never reply. This was fixed by dedicating a thread to each client, resulting in a more stable and more efficient messaging system.

Chapter 5

Conclusion

In our evaluation, we identified the key benefits in using a messaging framework such as Listen, as well as any potential costs it might incur. In addition, we delivered an analysis of the framework's stability and efficiency. Finally, we provided a example showing some of its capabilities, as well as demonstrating its simplicity.

Listen is clearly suited to message-based applications, such as the basic chat program outlined in section 4.1. However, it can be used in a huge variety of situations, as the request/response paradigm the web is based upon can be represented within the publish/subscribe model. With that in mind, the fact that it defaults to broadcasting information, rather than keeping it private, lends itself well to applications of a more social nature, encouraging communication between users as opposed to separating them. If the increasingly large number of social web applications is any clue to the direction the web is heading, this is an excellent position in which to be.

5.1 Future Work

There are many ways in which Listen could be extended to better serve the needs of developers and users. Perhaps the most obvious is accessibility: so reliant is the framework on client-side scripting, users with it disabled will just see a blank web page. Listen could greatly benefit the capability to provide static HTML to the browser, and trade AJAX requests for the simple practice of serving web pages.

To this end, Listen would need the capability to simulate JavaScript actions. However, this can be avoided if the JavaScript were automatically generated from a server-side script. If this script were written in Python, code could be shared between the client and the server, massively decreasing

the learning curve of the framework and significantly cutting maintenance of particular items, such as the broker and the stress test, which run on both. This would require a UI-independent feedback mechanism—one that could manipulate items on a web page through the DOM, alter HTML before sending it to the browser or send messages to the client to do so, all dependent on where the process is being executed and the type of request being made.

The core libraries provided—the Logger, Switcher and Editor—are fairly basic, and lack the features needed to make them useful for more than demonstration purposes. The Editor, in particular, needs a large overhaul for the reasons mentioned in subsection 4.2.2. This would almost certainly require another library related to user authentication and management, bringing even more functionality to the framework.

Due to time constraints, some decisions had to be made that favoured a quick implementation over the best possible. One example was the web server itself: written in Python because of its stellar library support and our prior experience with the language, a lot of work has gone into making everything work correctly and tracking down bugs. Given the time to learn and implement a more complicated development process, the framework would benefit from being integrated into a successful web server such as Apache HTTPd, which has been proven to be reliable and efficient. In addition, it would mean that the framework could be integrated with existing websites, increasing the size of its potential userbase.

Finally, there is the choice of the language itself. Python is known by many, is very popular and has libraries for any functionality a programmer might possibly want. However, its imperative nature makes it ill-suited for actor-based development. Porting Listen to an actor-based language such as Erlang, or one with native support alongside more traditional features such as Scala, could significantly improve the development process. However, Erlang has little in common with JavaScript, and the disconnect between them would increase the complexity of the framework. Simultaneously, while Scala works fairly similarly to JavaScript in some cases, it is a new language with its own teething problems. Every language brings both pros and cons to the table, and while Python was not perfect, it also has a large number of useful features not collectively found anywhere else.

Bibliography

[1] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *IJCAI*, pp. 235–245, 1973.

[2] C. Hewitt, "ActorScript(TM): Industrial strength integration of local and nonlocal concurrency for Client-cloud Computing," *CoRR*, vol. abs/0907.3330, 2009.

[3] J. Ayres and S. Eisenbach, "Stage: Python with Actors," in *IWMSE '09: Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, (Washington, DC, USA), pp. 25–32, IEEE Computer Society, 2009.

[4] U. Wiger, "stress-testing erlang." `http://groups.google.com/group/comp.lang.functional/msg/33b7a62afb727a4f`, 2005.

[5] "About Messages and Message Queues." `http://msdn.microsoft.com/en-us/library/ms644927.aspx`.

[6] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.

[7] "WebSphere MQ." `http://www-01.ibm.com/software/integration/wmq/`.

[8] "Apache ActiveMQ." `http://activemq.apache.org/`.

[9] Y. Balzer, "Improve your SOA project plans." `http://www.ibm.com/developerworks/webservices/library/ws-improvesoa/`, 2004.

[10] T. M. H. Reenskaug, "MVC." `http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html`.

[11] J. Gossman, "Introduction to Model/View/ViewModel pattern for building WPF apps." http://blogs.msdn.com/johngossman/archive/2005/10/08/478683.aspx, 2005.

[12] D. Crockford, "Javascript: The world's most misunderstood programming language." http://javascript.crockford.com/javascript.html, 2001.

[13] A. Russell, "Comet: Low Latency Data for the Browser." http://alex.dojotoolkit.org/2006/03/comet-low-latency-data-for-the-browser/, 2006.

[14] E. Bozdag, A. Mesbah, and A. van Deursen, "A comparison of push and pull techniques for ajax," *CoRR*, vol. abs/0706.3984, 2007.

[15] I. Hickson, "Server-Sent Events." http://dev.w3.org/html5/eventsource/, January 2010.

[16] Y. Fujishima, F. Ukai, and T. Yoshino, "Web Sockets Now Available In Google Chrome." http://blog.chromium.org/2009/12/web-sockets-now-available-in-google.html, 2009.

[17] "Browser Statistics." http://www.w3schools.com/browsers/browsers_stats.asp.

[18] A. Cannon, "JavaScript and screen readers." http://northtemple.com/2008/10/07/javascript-and-screen-readers, October 2008.

[19] "WAI-ARIA Overview." http://www.w3.org/WAI/intro/aria.

[20] "Statistics Dashboard." https://addons.mozilla.org/en-US/statistics.

[21] D. Crockford, "Introducing JSON." http://json.org/.

[22] P. Leach, M. Mealling, and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace." RFC 4122 (Proposed Standard), July 2005.

[23] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax." RFC 3986 (Standard), January 2005.

[24] P. Gralla, "Safari 5 in depth: Has it sped past Chrome?." http://www.computerworld.com/s/article/9177990/Safari_5_in_depth_Has_it_sped_past_Chrome_, June 2010.