

University of Edinburgh

School of Informatics

MAKING DIZZY SHINE WITH AJAX

4th Year Project Report
Computer Science and Management Science

Mike Arthur
mike@mikearthur.co.uk

February 27, 2007

Abstract “*Ajax applications can provide a more effective user interface than those of GUI desktop or classic Web applications with fewer drawbacks than either individual approach.*”

In this report, the preceding hypothesis was evaluated by developing an Ajax Web application for the Dizzy chemical kinetics stochastic simulator [2] and analysing the benefits and drawbacks compared with the existing Dizzy classic Web application and Dizzy GUI desktop application. Among the benefits were those of a more responsive interface, less bandwidth usage and allowing logic to be offput to the client. The drawbacks included the challenge of ensuring cross-browser compatibility, unfamiliarity to the user, JavaScript debugging issues and the time-consuming nature of re-engineering an existing Web application.

Acknowledgements

Thanks go to:

- Project supervisor *Stephen Gilmore* and group supervisor *Julian Bradfield* for guidance and help.
- *John Oberlander* for clarification and help with HCI issues.
- *Gareth Saunders* and *Patrick Thomson* for greatly aiding my proof-reading of this report.
- The developers of FIREBUG, SUBVERSION and ECLIPSE for invaluable debugging and development tools.
- TEAM KLINGON: *Chris Paton, Charles Harley, Benjamin Rollinson, Colin Silcock, Iain Hendry, Raj Khokhar, Alan Campbell, John Mark Ritchie* and *Alistair Strachan* for the development of the classic web application and *Alistair Strachan's* help with the existing code base.
- The developers of BATIK, APACHE COMMONS and JFREECHART for the used libraries.
- *Steve Moitozo* for the PASSWORD QUALITY METER.
- *Lindsay McQuaid* for support.

Contents

1	Introduction	1
1.1	What is Ajax?	1
1.2	When is Ajax used?	2
1.3	What are the problems with Ajax?	3
2	Literature Survey	5
2.1	Ajax: A New Approach to Web Applications	5
2.2	Navigating the Applet-Browser Divide	6
2.3	Ajax: How to Handle Bookmarks and Back Button	7
2.4	DHTML accessibility: solving the JavaScript accessibility problem	7
2.5	Beyond Ajax	7
2.6	Web 2.0 Next Big Thing or Next Big Internet Bubble?	8
2.7	Beyond the Desktop Metaphor in Seven Dimensions	9
2.8	An Architectural Style for Ajax	9
2.9	Usability in Web Design	9
2.10	Emotionally centred design	10
2.11	Literature Evaluation	10
3	Background	11
4	Theory	13
4.1	Learnability	13
4.2	Flexibility	13
4.3	Robustness	13
4.4	Time Affordances	13
5	Specification	15
5.1	Lost functionality	15
5.1.1	Run progress	15
5.1.2	Simulation parameter validation	16
5.1.3	Changing chart axes	19
5.2	Online problems	21
5.2.1	Saving files	21
5.2.2	Upload progress	21
5.2.3	Password update	22
5.3	Solutions	22

6	Implementation	25
6.1	Feedback for long-running server-side tasks	25
6.1.1	Graceful Fallback	27
6.2	Input validation from server-side parameters	27
6.2.1	Graceful Fallback	29
6.3	Quickly modifying the output dataset	29
6.3.1	Graceful Fallback	30
6.4	Save status without redirection	31
6.4.1	Graceful Fallback	31
6.5	Feedback for lengthy client/server interaction	31
6.5.1	Graceful Fallback	32
6.6	Input validation based on client-side data	32
6.6.1	Graceful Fallback	33
6.7	Problems Encountered	33
6.7.1	Design of previous system	33
6.7.2	JavaScript	34
6.7.3	SVG	36
7	Evaluation	37
7.1	Learnability	37
7.2	Flexibility	38
7.3	Robustness	39
7.4	Time Affordances	40
8	Further Work	41
8.1	SVG	41
8.2	More vigorous usability testing	41
8.3	Testing under poor network conditions	42
8.4	Security	42
8.5	Comet	42
9	Conclusion	45
9.1	Hypothesis	45
9.2	Benefits	45
9.3	Drawbacks	45
9.4	Contribution	46
	Bibliography	47
10	Appendix	49
10.1	Glossary	49

Chapter 1. Introduction

1.1 What is Ajax?

Ajax is shorthand for “Asynchronous JavaScript and XML” and is a recent technique for developing more interactive Web applications, allowing faster input and better usability, striving for similar interfaces to those provided by GUI desktop applications.

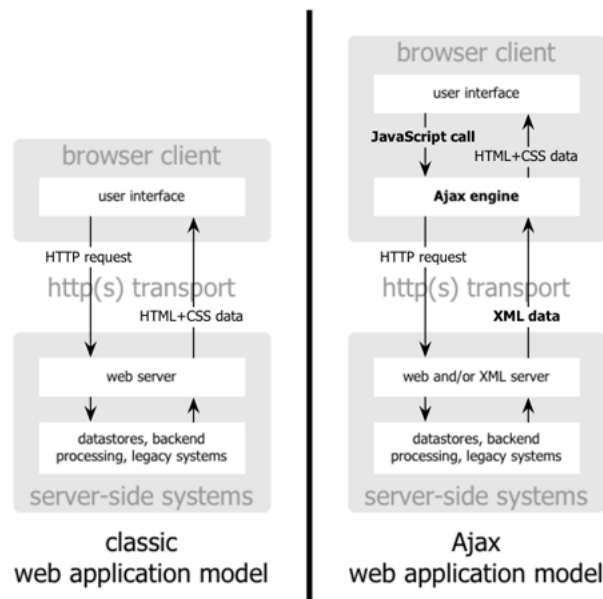


Figure 1.1: Comparing Ajax and classic Web application models

Copyright Adaptive Path, LLC 2005

Figure 1.1 shows the main differences between a classic Web application and an Ajax Web application.

A classic Web application sends a request to a server which generates the page, perhaps from a database, returning a page which is then displayed in the client’s browser. On each new page request or the retrieval or submission of new data, the whole page must be re-requested from the server.

An Ajax Web application follows the same pattern on the initial retrieval of a page, however, when new data needs to be sent or received from the server, instead of re-requesting the whole page, the browser can call a method written in JavaScript. JavaScript is a browser scripting language and in this case can be used to retrieve the data asynchronously. This data is normally returned as XML, which is then parsed by the JavaScript and used to modify the DOM and CSS of the page.

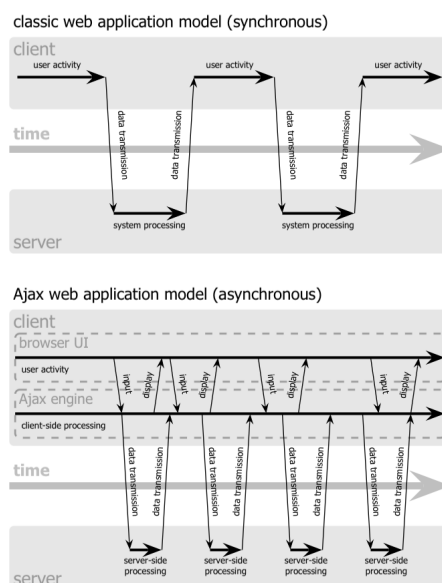


Figure 1.2: Comparing Ajax and classic Web application client/server interaction
Copyright Adaptive Path, LLC 2005

Figure 1.2 compares the typical interaction model between a classic Web application and an Ajax Web application.

The classic Web application requires user activity to take place then be submitted to the server and the data must be processed and returned successfully to the client before the user can commence any more interaction.

The Ajax Web application allows continuous user activity. On certain events JavaScript functions will be run by the browser, allowing client-side processing of data, such as input validation. If necessary, this data can then be sent to the server and/or the server return more data. This usage pattern is closely similar to that of a GUI desktop application where the user does not typically have to wait for the completion of tasks and can work without worrying about manually initiating interaction with the “server”.

1.2 When is Ajax used?

With increased bandwidth now available to large proportions of the first-world population, Web applications are becoming feasible replacements for GUI desktop applications. This movement is known by some as “Web 2.0”. One of ideas behind “Web 2.0” is the Web as an application platform, replacing GUI desktop applications, allowing cross-platform applications and allowing data to be easily shared and manipulated between machines in different geographical localities. The only prerequisites for these applications are an Internet connection and a relatively modern browser.

The first wave of these applications to move online was email. There has been Web-based email for over 10 years and it is many users' sole way of accessing their email. Google's email application, Gmail, was one of the first uses of Ajax technology on the Internet. For browsers that support Ajax, the user interface is similar to a GUI desktop application, with the ability to drag items and quickly open and close emails without the whole page reloading.

With Ajax the goal of moving GUI desktop applications online may finally be realised. As shown in Figure 1.2 on the facing page, Ajax finally allowed the user of an Ajax Web application to have an uninterrupted workflow. This can potentially greatly boost their productivity, bringing it more in-line with that from GUI desktop applications, whilst keeping the benefits of a Web application such as distributed processing, no need for local backups, high up-time and portability.

Now users and developers wishing to take advantage of Ajax have many options available to them. For developers, many Ajax libraries are now available, making Ajax user interface development as simple as importing any other library. Users can now find Ajax applications for word processing, spreadsheets, picture management, email, satellite mapping and more.

1.3 What are the problems with Ajax?

Ajax technologies, while being useful to both users and developers, are still in their infancy. Ajax applications frequently suffer from niggling bugs such as "breaking" the expected functionality of the "Back" button, causing other unexpected browser behaviour and may have security risks. It is also currently unclear whether the claimed benefits of Ajax actually result in better user interfaces. The role of this project is to try and discover any benefits and problems and see whether Ajax Web applications provide interface improvements over classic Web applications and existing GUI desktop applications, whilst having fewer disadvantages than either individual approach. A literature review was conducted in order to discover the need for this discussion. An existing classic Web application was modified into an Ajax application after critically evaluating the user interface failings of the existing classic Web application and GUI desktop application. Finally, the new Ajax Web application was evaluated and through this the strengths and weaknesses of Ajax were discussed.

Chapter 2. Literature Survey

2.1 Ajax: A New Approach to Web Applications

The term “Ajax” was first used in “AJAX: A NEW APPROACH TO WEB APPLICATIONS” [3], an article about interaction design. The two examples referenced as examples of Ajax were Google’s Suggest and Maps services.



Figure 2.1: Google Suggest

Figure 2.1 shows an example of Google Suggest in progress. As search queries are entered, suggestions for the search query are displayed, updating almost instantaneously with no other user interaction.

Figure 2.2 on the following page displays an example of Google Maps’ usage. This figure does not convey the use of the interface as well as a demonstration, but the map can be manipulated by clicking and dragging around the map, and the map is dynamically loaded and updated, with no need to press reload. Also, take note of the shadow overlay from the Address “bubble”, again, dynamically updated in real-time.

These applications are an example of what Adaptive Path called Ajax. Ajax makes use of the manipulation of XHTML and CSS, using the DOM, by JavaScript, and information retrieval and manipulation using XML, XSLT, XMLHttpRequest and JavaScript. The information retrieval is done by regular polling of XML data

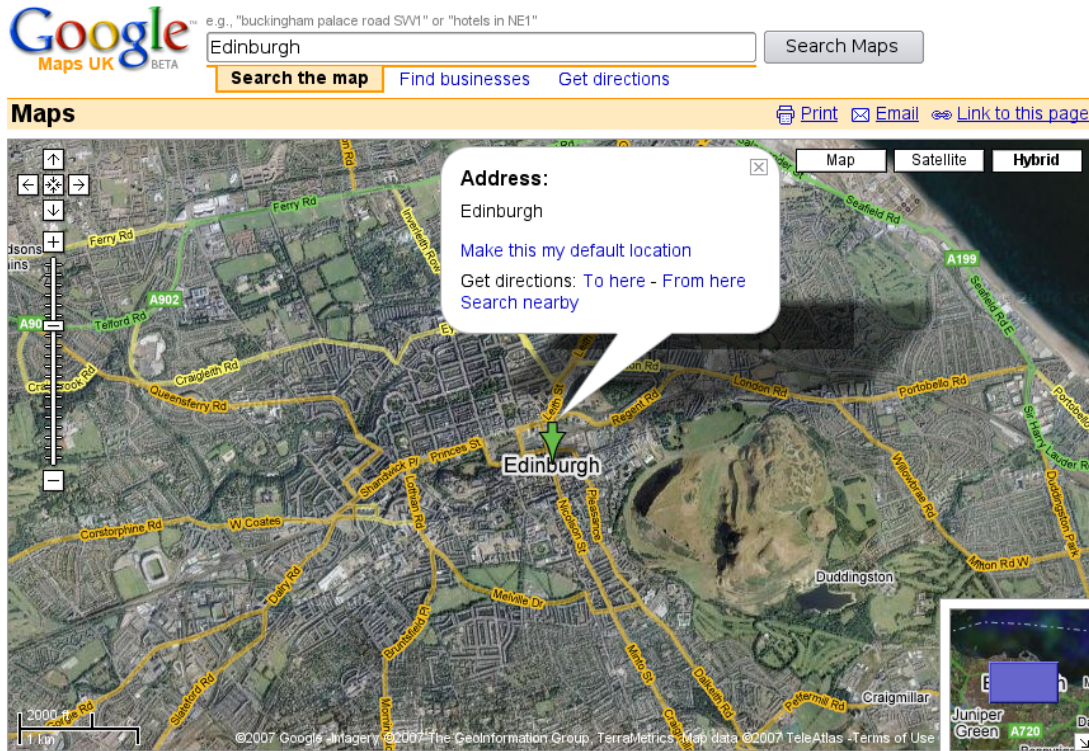


Figure 2.2: Google Maps

using JavaScript, providing the interactivity not existing in classic Web applications.

At the time of this article, the main users of Ajax were Google, making use of it on their Gmail, Groups, and the aforementioned Suggest and Maps services, but it also saw use in Flickr and Amazon's A9 search engine. The article provided a clear explanation of both the technical usages of Ajax, its implementation and merits, and paved the ground for future studies in using Ajax for rich-client applications. Although this article was the first mention of Ajax, analysis of the usability of related technologies have been seen before, albeit with different terminology, such as DHTML.

2.2 Navigating the Applet-Browser Divide

Comparable issues to those spawned by Ajax have been analysed before, such as in "NAVIGATING THE APPLLET-BROWSER DIVIDE" [10]. The feel of browser applets was very different to that of classic Web applications and raised similar usability problems to those encountered with Ajax applications. Users were found to instinctively press the browser's "Back" button when intending to return to a previous stage in the applet process, which, if the applet had not taken this

into consideration, wiped all progress they had undertaken so far in the applet, causing the user to have to start again. Again, applets brought claims of allowing the user to “dynamically interact with information”, similar to those claims made today with the rise of “Web 2.0” and Ajax. This article contained the results from a usability study which found that users tended to depend on the browser navigation buttons to navigate through the applet. When navigation buttons were provided inside the applet the users were found to have fewer problems with integration and navigation. This article also highlighted the now so-relevant issue of ensuring compatibility between browsers.

2.3 Ajax: How to Handle Bookmarks and Back Button

The issue with the “Back” button was further explored in “AJAX: HOW TO HANDLE BOOKMARKS AND BACK BUTTON” [9]. The problem was raised by using Gmail as a case study. Once a user enters Gmail, at the time of publishing, the URL remained the same throughout the session, so if a user entered an email message, and then wished to leave the message and go back to the inbox by clicking the browser’s “Back” button, to their surprise they would find themselves out of Gmail.

2.4 DHTML accessibility: solving the JavaScript accessibility problem

Before the term “Ajax” became popular, some of the technologies used were referred to as DHTML. “DHTML ACCESSIBILITY: SOLVING THE JAVASCRIPT ACCESSIBILITY PROBLEM” [4] discussed some of the other issues arising from the usage of these technologies, focusing on the difficulty in providing content accessible to those with disabilities when the application relies on JavaScript. The main problems highlighted were dealing with focus on dynamic content and providing semantic data for GUIs, to allow the use of purely text-based input and display. It also called for a paradigm shift to ensure that interfaces are all usable through a keyboard interface and other assisting technologies, hopefully providing interfaces that can be better than the existing alternatives for disabled users.

2.5 Beyond Ajax

“BEYOND AJAX” [5] discussed both the potential of Ajax Web applications and their limits. The first limit shown is that Ajax applications cannot allow realtime



Figure 2.3: Adobe Flash’s multimedia privacy settings

event notification, instead requiring a polling loop; the more regular the poll, the more bandwidth and processing used, both on the client and server. The second limit raised is, that although most processing-intensive tasks can and have been made into Web applications, the GUI desktop equivalents provide certain advantages. An example of this is direct hardware access, which is not possible without browser plugins, such as Adobe’s Reader or Flash, the latter providing video and audio recording, with user-controllable security levels (see Figure 2.3). This paper showed some of the possible advantages of Ajax such as dramatic bandwidth savings in leveraging the transfer of XML using JavaScript rather than reloading whole pages. However, the benefits and costs of migrating GUI desktop applications online must be weighed up, with respect to security, bandwidth, response and interactivity issues.

2.6 Web 2.0 Next Big Thing or Next Big Internet Bubble?

The term “Web 2.0” was first used publicly by O’Reilly Media as a name for a conference in 2005. It has become rather popular in technology circles, describing the rise in the use of Ajax and other XML technologies and also the growth of Web-based communities around these technologies. “WEB 2.0 NEXT BIG THING OR NEXT BIG INTERNET BUBBLE?” [1] emphasised the large number of technology start-ups based on these concepts and made comparisons made to the “Dot Com Bubble”. This article highlighted the use of a “Rich User Experience”, which refers to the use of Web applications attempting to behave more like their GUI desktop counterparts. Also, the importance of dynamic content, user participation, meta-data and markup were raised, allowing a way for users to both create and find information more effectively. This article reiterated the previously mentioned issues accompanying Ajax technologies, such as lack of JavaScript availability and

“breaking” the “Back” button, also the difficulty of bookmarking dynamically generated pages with varying states, meaning that storing the state of pages for a bookmark may be very difficult.

2.7 Beyond the Desktop Metaphor in Seven Dimensions

The issue of creating more rich-Web applications was discussed in “BEYOND THE DESKTOP METAPHOR IN SEVEN DIMENSIONS” [8]. It mentioned the use of Ajax technologies to create far more GUI-like Web interfaces, allowing them to be more familiar to users and updated without user intervention. It also pointed to the recent growth of Ajax applications being used, forming the previously mentioned “Web 2.0”, bringing a shift from applications (such as those on the desktop) to online services, frequently using Ajax, storing personal information and documents outside of an office machine.

2.8 An Architectural Style for Ajax

“AN ARCHITECTURAL STYLE FOR AJAX” [7] discussed the use of various frameworks for the creation of Ajax-enabled pages, allowing the Web developer to worry more about content than writing the JavaScript manually. The main focus of the paper was the styles for architecture of Ajax-based solutions. It introduced the idea that Ajax applications can be seen as a hybrid, combining features from GUI desktop and Web applications. This brought the conclusion that an architectural style must be created, rather than simply reusing existing desktop or Web styles. The style described in this paper is known as “SPIAR”, and highlights the factors of intractability, latency, network performance, simplicity, scalability, portability and visibility in Ajax application design. The rest of the paper focused more on specific application development techniques.

2.9 Usability in Web Design

“USABILITY IN WEB DESIGN” [6] revealed Ajax as a technique for increasing the speed of page loading, especially when only a small change is made to the page content. It also emphasised the need for accessibility considerations when using client-side technologies, such as Ajax, to continue to ensure the separation of style and content, and ensuring the page falls back correctly to browsers without JavaScript.

2.10 Emotionally centred design

The concept of Rich Internet Applications was discussed in “EMOTIONALLY CENTRED DESIGN” [11]. Rich Internet Applications are Web applications that have similar functionality and features as a GUI desktop application, but run in a Web browser. Ajax technologies are normally used in the creation of RIA. The paper posed a question, examining a different dimension of Ajax technologies, “why are these [*Ajax sites*] so compelling?”. The reasons returned were those of fluid movement and immediate responses to user input, which create “engaging interfaces”. However, the issue of Ajax becoming a development trend was raised. A call is made for detailed studies into user interaction with RIA, rather than developers to creating RIA for the sake of personal exploration and jumping on the latest technological bandwagon, instead trying to use RIA interfaces to improve user effectiveness and engagement.

2.11 Literature Evaluation

The previously analysed literature provides a detailed examination of both the advantages and drawbacks arising from using Ajax in Web application development. However, multiple articles call for a detailed usability study into the efficiency of user interaction with a Ajax application, and a direct study of this does not appear to have been done. This is a major problem as most of the articles are speculative on the benefits of using Ajax. This means that there is a real difficulty for developers currently debating whether to re-engineer a classic Web or GUI desktop application using Ajax or create a new Ajax application due to the lack of “hard” data to support the conclusions found in the current available literature.

Chapter 3. Background

Dizzy is a chemical kinetics stochastic simulator written in Java, available as a GUI desktop application. It provides a model definition environment and an implementation of the Gillespie, Gibson-Bruck, and Tau-Leap stochastic algorithms [2]. This application was modified to use Enterprise Java technologies and run on a J2EE Web application server (such as Tomcat) by a group of students in 2005, known as “*Team Klingon*”. This means there is currently an application that is feature-complete, with two implementations, Web-based and desktop-based.

This report will consist of creating a third implementation using Ajax. This will be performed by modification of the Web-based implementation, using the desktop-based implementation as a reference, to try to create a more usable interface, from evaluation of the drawbacks of the previous two interfaces. This new interface will receive a detailed usability study aiming to fill the gaps in papers highlighted in the Literature Survey: the need for a usability study of Ajax technologies, compared with both GUI desktop and classic Web applications.

From here onwards, for ease of explanation, the GUI Dizzy implementation will simply be known as DIZZY, the non-Ajax Web-application version as KLINGON (as it was designed by “*Team Klingon*”) and the Ajax version as SHINY (referencing the title of this report).

Chapter 4. Theory

The aim of this report is to test the following hypothesis:

“Ajax applications can provide a more effective interface user interface than those of GUI desktop or classic Web applications with fewer drawbacks than either individual approach.”

This hypothesis will be evaluated by examination of the following criteria [12]:

4.1 Learnability

Ajax applications allow the user to have an interface that is more predictable and allows greater response feedback than either of the alternative applications.

4.2 Flexibility

Ajax applications match the user’s interface expectations better than the alternative applications, allowing them to take better control of dialogue flow, and allowing support for more threads of simultaneous operation. Different forms of input are better facilitated and the interface can be more customisable.

4.3 Robustness

Ajax applications allow for a more honest interface, better indicating the user’s action history and current state in the application. Errors are more rapidly repaired and prevented. The application is more responsive, providing better feedback to user input.

4.4 Time Affordances

Ajax applications allow fewer or no more unpredictable delays than the alternative applications. The Ajax application allows for greater reassurance to unavoidable delay than alternative applications.

Chapter 5. Specification

In comparing DIZZY's implementation with KLINGON's, the benefits provided by the Web application (e.g. distributed processing, client-server architecture, portability etc.) are ignored and instead the focus of this report is on the functional user interface differences between the two applications. These differences can be split into two areas: features lost from the original application by moving to a static Web interface and new problems introduced by the online application.

5.1 Lost functionality

5.1.1 Run progress

One of the main problems encountered when using KLINGON, compared with DIZZY, was informing the user of the progress of a run. The application performs runs which can vary greatly in time consumed, ranging from milliseconds to hours, depending on the complexity, machine load and other factors.

5.1.1.1 DIZZY

The screenshot displays the DIZZY web application interface during a simulation run. The interface is organized into several panels:

- model name:** [model]
- controller:** Includes buttons for 'start', 'cancel', 'pause', and 'resume'.
- simulators:** A list of simulation engines: ODE-RK5-adaptive (selected), ODE-RK5-fixed, ODEtoJava-dopr54-adaptive, ODEtoJava-imex443-stiff, gibson-bruck, gillespie-direct, tauleap-complex, and tauleap-simple.
- Simulation Parameters:** Fields for 'start' (0.0), 'stop' (450), 'number of results points' (100), 'stochastic ensemble size' (empty), 'step size (fractional)' (0.0010), 'max allowed relative error' (1.0E-4), 'max allowed absolute error' (0.01), and 'number of history bins' (400).
- view symbols:** A list of symbols including DNA3, DNA4, DNA80, G3D_G80D, G3D_free, G3_RNA, G3_protein, G4D_DNA3, G4D_DNA4, G4D_DNA80, G4D_free, and G4_RNA. A 'select all' button is at the bottom.
- Output Type -- specify what do do with the simulation results:** Radio buttons for 'plot' (selected), 'table', and 'store'. Below 'store' is an 'append' checkbox and a 'format' dropdown set to 'CSV-excel'.
- simulation results list:** A table showing three entries: [10/02/07 22:48] [model] [ODE-RK5-adaptive], [10/02/07 22:48] [model] [ODE-RK5-adaptive], and [10/02/07 22:49] [model] [ODE-RK5-adaptive] (highlighted). A 'reprocess results' button is below.
- Progress Bar:** A horizontal bar at the bottom left.
- secs remaining:** 144.9985, displayed at the bottom right.

Figure 5.1: Running a simulation with DIZZY

Figure 5.1 displays the interface's appearance while a run is in progress. The main elements to change and update are the progress-bar in the bottom left, and the estimated time remaining in the bottom right. There is also the facility to

cancel or pause a run if it seems to be taking too long or system resources are temporarily needed for another task.

5.1.1.2 KLINGON



Figure 5.2: Running a simulation with KLINGON

In Figure 5.2 there is no such progress indication. The user of the system is informed that the simulation “may take some time”, without indicating how long this may be, and the user is expected to wait at the page, with no progress indication, until the run is completed. Furthermore, there is no functionality provided to pause or cancel a run, so if the run takes an excessive amount of time it is unclear to the user how to terminate this, meaning server time may be wasted simply because the user is unable to stop an unnecessary run.

5.1.2 Simulation parameter validation

When performing a run of a simulation the user must enter some parameters for the run itself and for the calculation of results from the run’s conclusion. These parameters are subject to various constraints, e.g. the “stop time” cannot be before the “start time” and the “step size” must be fractional. These constraints must be checked at some stage, either at the user interface or simulator level, or the run will fail.

5.1.2.1 DIZZY

Figure 5.3 on the facing page shows the results from when the user presses the “start” button to run a simulation with an invalid “stop time”. The dialogue prints

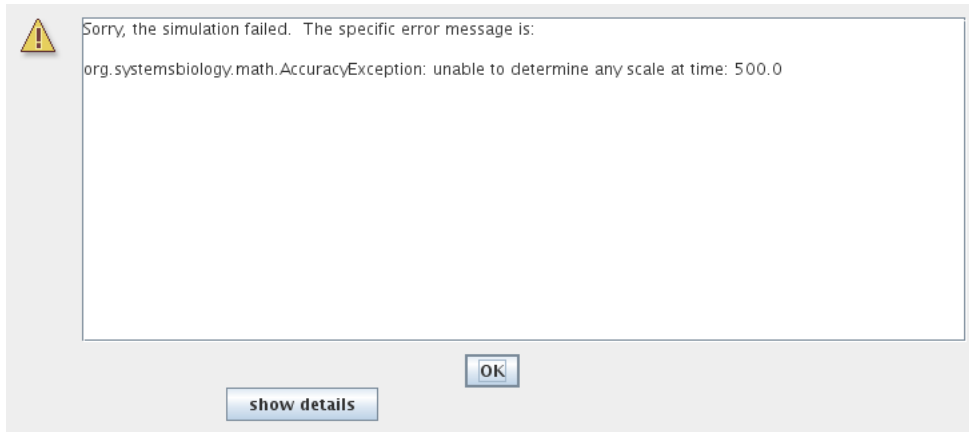


Figure 5.3: Stop time failure with DIZZY

the output from an internal exception in the simulator, an “AccuracyException”, and informs the user that it is “unable to determine any scale” at a certain time. This error is cryptic as it does not indicate which field had an incorrect value, nor the acceptable constraints of the value. Pressing the “show details” button provides no further help, simply printing the stack trace of the exception; very little help to a non-programmer. Also, this error does not appear until the user decides to run the simulation.

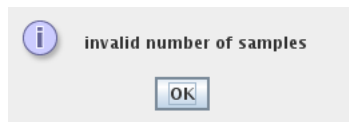


Figure 5.4: Result point failure with DIZZY

Figure 5.4 shows the results from when the user presses the “start” button to run a simulation with an invalid number of result points. This time the dialogue is slightly more helpful, with no confusing jargon, but it still does not refer exactly to the field, referencing “number of samples” rather than “number of results points”. Again, this error does not appear until the “start” button is pressed.

Figure 5.5 on the following page shows the results from when the user runs a simulation with an invalid “number of result points”. This error is similar to the first, in that it simply outputs the Java exception that caused the error. The main problem, that cannot be seen from the screenshot alone, is that this error only appears on run completion, meaning that if the parameter for “relative error” was invalid and a long run was performed that time has been wasted performing a run that outputs no data. As before the terminology displayed in the error is different from that used in the GUI, “relative tolerance” in the error but “relative error” in the GUI.

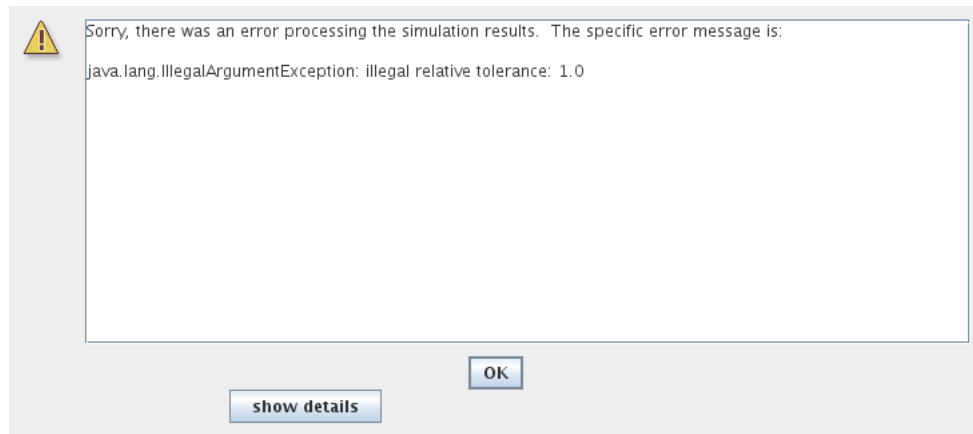


Figure 5.5: Relative error failure with DIZZY

The problem consistent across the errors is a lack of indication of the constraints of the parameter validation. Another problem is the decision to only perform validation when the “start” button is pressed. Java’s GUI libraries provide facilities to manage and handle the user’s input on each individual field’s entry, but this is not done. Due to this the process to ensure data is correct is slow as only one error is thrown at a time, so if multiple there are multiple errors, these must be sorted one-by-one before the run can be started.

5.1.2.2 KLINGON

Figure 5.6 on the next page indicates the outcome from entering bad data into every field of the form. This data is then sent to the server which validates it and in the case of error redirects the user back to the original page marking the errors. If the data was valid the user would simply be forwarded to the next stage of the form.

The first problem seen with the errors is that for the values that were “out of legal range” there is no indication of what the range is so if the user’s entry was correct but just too high to simulate they will have to use trial and error to find out what are the maximum values they can use.

The errors caused by non-double or non-integer values occurred due to the entry of letters rather than numbers in those fields. In this case the indication of the datatype for the field is provided, however this could be made clearer still.

The main problem with the KLINGON approach is, due to purely server-side validation, the user receives no feedback on bad parameters until they have submitted the form to be evaluated. In addition, due to the redirect mechanism, the previous values they entered are lost on the failure of the parameter validation (as observed in Figure 5.6 on the facing page). The server-side mechanism proves little problem on high-bandwidth connections, but when either the client

ODEtoJava-imex443-stiff Simulation Parameters

Start Time:
Out of legal range

Stop Time:
Out of legal range

Number of Samples:
Non-integer Number of Samples

Relative Tolerance:
Out of legal range

Absolute Tolerance:
Non-double Absolute Tolerance

Step Size Fraction:
Out of legal range

Number of History Bins:
Non-integer Number of History Bins

Run Description

Description for this run:

No run description provided

Save Run

Figure 5.6: Simulation parameter verification with KLINGON
(Note: The values displayed are not invalid. KLINGON's server-side validation does not return the invalid values, instead displaying the errors on a new page)

or the server has a low-bandwidth connection this could become a painful process, requiring the data to be re-posted and all the page content reloaded on every mistake made in the form.

5.1.3 Changing chart axes

5.1.3.1 DIZZY

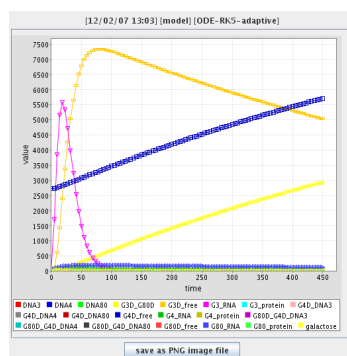


Figure 5.7: Chart generation with DIZZY

Figure 5.1 on page 15 shows the DIZZY interface mid-simulation. The datasets for the graph/table/CSV file are chosen before the run is started and cannot be changed after the run has completed. Figure 5.7 graphically represents the output

from a simulation. In this example, if the most relevant axis on examination was G4_RNA, there is a problem. As the software does not allow modification of the axes without performing the run again and the graph output is a raster rather than vector image, the axis cannot be viewed more closely without repeating the run.

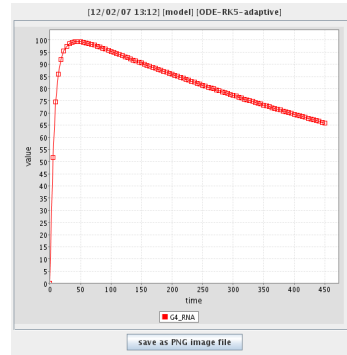


Figure 5.8: Chart generation with DIZZY (a single-axis)

Figure 5.8 shows the output after performing the run again. It is much clearer now what the data represents, and far more useful, but sadly the same simulation was run twice and the other datasets discarded the second time. This simulation data could have instead been cached, reprocessed every time an image is generated.

5.1.3.2 KLINGON

KLINGON handles simulations differently. Rather than performing a run and immediately producing the output, the simulation data is cached in a database and this data is used every time the user chooses to generate a graph, table or CSV file.

Series selection

time
DNA3
DNA4
DNA80
G3D_G80D
G3D_free

Chart options

Chart Title:	Dizzy Run Output Ch
Chart Height:	400
Chart Width:	400
<input type="button" value="submit"/>	

Figure 5.9: Chart setup with KLINGON

Figure 5.9 shows the axis selection, very similar to DIZZY.

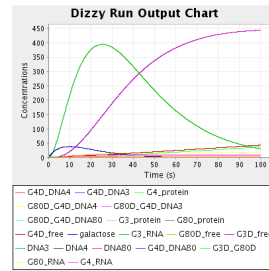


Figure 5.10: Chart generation with KLINGON

Figure 5.10 shows the chart generated for the user on a new page. If a similar approach is taken to that with DIZZY, wanting to focus on the G4_RNA, there is no need to perform the run again but it is necessary to hit the “Back” button in the browser and re-choose the axes. This method does not require redoing the run, but the user still has to navigate between different pages to make the new choices.

5.2 Online problems

The following problems are those introduced only in KLINGON, due to its online interface. These do not have comparable equivalent problems in DIZZY.

5.2.1 Saving files

Rather than using DIZZY’s method of requesting a local file location for the simulator file for every run, KLINGON allows two methods to create these files. The first is to manually enter the file into an editor and the second to upload the file. The latter will be covered in the next subsection.

Figure 5.11 on the following page shows the text editor available in KLINGON. This can be used to create simulator model files or modify existing model files. The main problem with the editor is that when “Update file” is clicked the browser sends the new contents and the page redirects. This means that to edit the file once more the user must reselect the file and open the editor again.

The problem with this is that it encourages users to not save the file until they are finished, and with files upwards of 100 lines, if the browser crashed this could be frustrating for the user.

5.2.2 Upload progress

As mentioned in the last section, model files can be uploaded. However, these files can get quite large, and over a slow network connection or a loaded server, the page appears to hang, with the browser indicating it is “Busy” until the upload

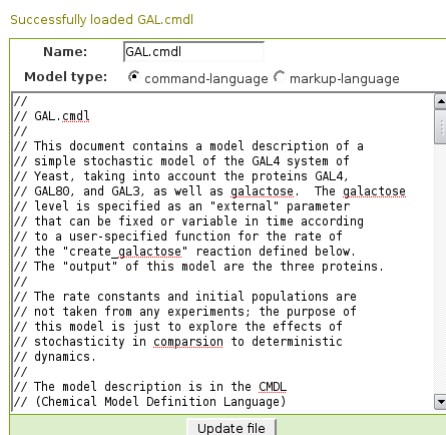


Figure 5.11: Simulator file editing in KLINGON

completes. It would be useful for the user to be notified of the progress of the upload, so they can estimate how long it will take and whether to allow it to complete.

5.2.3 Password update

Figure 5.12: Updating the users details in KLINGON

In the form seen in Figure 5.12, the user is invited to enter the old password, the new password and confirm the new password. If the new password does not match the confirmation, the request still requires to be sent to the server before this is notified to the client. It would be sensible, in this situation, to check the fields differ before this information is sent to the server. Similarly it would be sensible to check the old and new passwords are not the same.

5.3 Solutions

The aforementioned problems, while not the only problems in the application, are broadly summarised into six main areas with DIZZY and/or KLINGON:

1. Feedback for long-running server-side tasks

2. Input validation from server-side parameters
3. Quickly modifying the output dataset
4. Save status without redirection
5. Feedback for lengthy client/server interaction
6. Input validation based on client-side data

These areas are those that can be considered to affect the usability of these applications most severely. Simple aesthetic and organisational changes have been ignored, as the focus of this report is on using Ajax to improve applications, not classic Web design or usability techniques.

These will be approached in SHINY by using KLINGON as a base, and using Ajax techniques, modifying both the front and backend code, but leaving the simulator logic itself intact. The difficulties in implementing these solutions and an evaluation of their effectiveness will then follow.

Chapter 6. Implementation

For each of the problems mentioned in Section 5.3 on page 22 the implementation of the solution will be outlined, any problems encountered and a detailed explanation of the workings of the solution. Each section also includes a subsection to discuss how the solution gracefully falls back to browsers that do not support the needed Ajax functionality.

6.1 Feedback for long-running server-side tasks

The first step in implementing this feature was adding the necessary hooks into the existing parts of the Web application used to run the chemical simulations. This was needed as the previous method of performing simulations was by a blocking method called from the JSP. This was made non-blocking, and the progress was made available to the JSP which allowed the implementation of a graceful fallback method for long-running simulations.

With all Ajax applications, as there is not currently universal browser support, it is essential to ensure that applications fallback gracefully if the XMLHttpRequest object cannot be created or if JavaScript is not running on the browser at all. With this in mind, the progress data now available to the JSP pages was used to create a static page that contained the progress-bar and time remaining. However, for the user to be able to see the current progress of the run, the page has to be regularly refreshed periodically. While this is possible using the Meta refresh tag, it is discouraged by the W3C's Web Content Accessibility Guidelines [13] as if the user is in the middle of another task in the Web browser, such as entering a new URL, this could be interrupted or lost by a page refresh. Also, as the progress page is around 3KB, with 26KB of other data (that can be cached), this a large amount to be periodically refreshed without user intervention. It was decided to simply allow the user to manually refresh the page, and they would be automatically redirected and informed on run completion. This is an improvement on the complete lack of progress feedback in KLINGON, but far from a desirable result, as it still requires user interaction, unlike DIZZY.

With Ajax this user interaction is not required. A few mechanisms on the client and server are combined to give a transparent update of data. The first step is creating an XML server. A JSP file makes use of a server-side session variable to locate which user and run are requested and then returns an XML file containing the progress and the time remaining. The next step is creating the necessary client-side JavaScript, which is run by the browser automatically on the page load. This JavaScript creates the XMLHttpRequest object and sets the URL to asynchronously get the XML file generated by the server. Rather than busy-waiting on a response from the server, a callback method is set which is run

on a change of the ready state of the XMLHttpRequest. This state progresses from the initial value to open, sent, receiving and finally loaded. When it is loaded, the callback method checks the status of the HTTP response. The XML server makes use of the HTTP response's status codes to provide information about the status of the run; a lower bandwidth method than using XML.

The following status codes are used:

- 200 OK - This code indicates there has been a change since the last client progress update, and that progress data is available.
- 303 SEE OTHER - This code indicates that the run has completed.
- 204 NO CONTENT - This code indicates that there has been no change in progress since the last client progress update.

The OK status code provides the progress data as XML, whereas the other two provide no XML content. This is slightly more complex than using a purely XML-based method but is more bandwidth-efficient as no XML data is transferred and the client does not need to respond to the server's initial HTTP response.

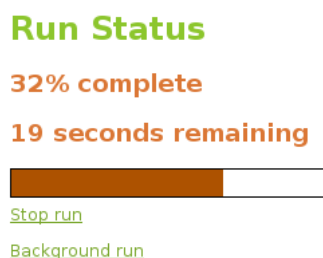


Figure 6.1: SHINY run progress

On an OK status code the callback method parses the XML provided by the server to obtain the progress information. This information is used to then manipulate the DOM and CSS of the progress page in the browser, updating the colour and length of the progress-bar, the progress percentage and the estimated time remaining. This can be seen in Figure 6.1.

On a SEE OTHER status code the page is redirected to the runs page, as occurred in KLINGON on completion of a run.

On a NO CONTENT status code the page is not updated, but in order to inform the user an update but no new progress data was available has occurred the status bar of the browser displays "No Update".

Regardless of the status code, on the evaluation of the code and associated response, the callback method is set to run again in a one second.

To further optimise the XML size, the XML tag names were shortened as this XML will not be read or downloaded other than by JavaScript.

For example, a typical XML response with non-optimised tags might be the following:

```
<progress>
  <completed>50</completed>
  <remaining>20 seconds remaining</remaining>
</progress>
```

After optimisation:

```
<p><c>50</c><r>20 seconds remaining</r></p>
```

The optimised XML file is, on average, around 82 bytes. Compared with the 3KB minimum from the meta refresh method, it is clear that for large amounts of clients, or low-bandwidth connections, the Ajax method is far more efficient.

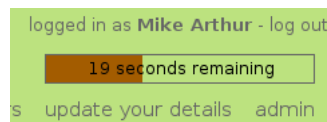


Figure 6.2: SHINY run progress embedded in the navigation bar

Also added to SHINY was the ability for the run to complete in the background, allowing the user to accomplish other tasks while the run was completed on the server. This was facilitated by the method of handling runs, and also the new progress-bar. The progress-bar code was designed to be portable, so it was easily embedded into the navigation bar as a small reminder of the run progress; this can be seen in Figure 6.2.

6.1.1 Graceful Fallback

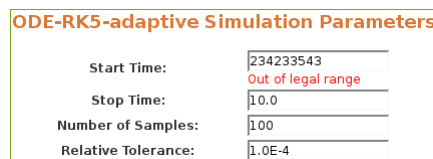
In event of the XMLHttpRequest object failing to be created or disabled JavaScript the progress will simply update whenever the user takes a navigation action in the browser. For instance, when they move to a new page it will display the latest progress in the navigation bar, as shown in Figure 6.2 or if the user simply refreshes the progress page they will see a display similar to Figure 6.1 on the facing page. The only difference will be a lack of animation due to the updates being initiated by the user rather than the server.

6.2 Input validation from server-side parameters

Due to previous input validation in KLINGON occurring on the server-side, there was no need to delve too deeply into the existing code in order to access the server-side parameter checking. The first step in this case was creating an XML server to

serve the error information to the client. The previous method in KLINGON sent the contents of the form to the server, which was validated and then accepted or returned with any errors. In SHINY this was handled by sending the output of a single input box from the form, and the unique identifier for the input box to the server. With SHINY this is not done on submission of the form but instead when the current input box loses focus, that is the user moves the text entry indicator to another input box.

The transmission of the value is done in much the same way as the progress indicator, except this time the XML server accepts the previously mentioned parameters and, in event of an error, produces a relevant error message. Also, as in the progress example, this is done by means of an XMLHttpRequest in JavaScript, not on direct user prompting, with HTTP status codes and XML optimisation again used to save bandwidth. This method, however, is not a regular poll, but only occurs when the user has changed the text entry indicator, indicating that they have finished with that field.



ODE-RK5-adaptive Simulation Parameters	
Start Time:	234233543 Out of legal range
Stop Time:	10.0
Number of Samples:	100
Relative Tolerance:	1.0E-4

Figure 6.3: SHINY server-side validation

Figure 6.3 shows the results of an incorrect start time. This looks very similar to the results from KLINGON, but there are two key differences. Firstly, the illegal value is still displayed. Secondly, the invalid value will be displayed instantly, with only 64 bytes used in the transmission of this error; KLINGON's method requires 6-11KB. The error messages provided are not any more helpful, but this was felt to be not an Ajax-related issue, and could have been fixed trivially. The main advantage the SHINY solution provides is the speed the use trial-and-error to find acceptable values, with near-instantaneous feedback without the removal of the invalid value.

A possible alternative to the Ajax method used in SHINY would simply be to use pure JavaScript to do input validation. This is discussed in Section 6.6 on page 32 but was not used in this case. An advantage of this method of server-side validation is that the input parameters are not made known to the user. This could be seen as a disadvantage, but it can help with two factors: security and flexibility.

The first benefit is that using server-side validation allows the criteria for input rejection to be kept secret from a client, requiring brute-force attacks in order to attempt to work out the acceptable range. If this were done using JavaScript client-side validation, the ranges could be easily determined and attacked more effectively. Also, if only client-side validation is used, a malicious client could

simply disable JavaScript support in the browser and their input would not be subject to any validation.

The other benefit comes in the handling of change. In this example, if a server administrator wanted to immediately change the validation ranges, or remove them completely, any clients who had a Webpage with the existing JavaScript code downloaded or cached would check the old ranges rather than the new.

6.2.1 Graceful Fallback

In event of the XMLHttpRequest object failing to be created or disabled JavaScript the parameters will simply be evaluated in the same manner as in KLINGON, posting the whole form to the server upon the user's request.

6.3 Quickly modifying the output dataset

Users frequently make mistakes. They are required in DIZZY and KLINGON to make decisions on output, with no idea of what the output will look like. This problem was amplified in DIZZY, as to change the output it was necessary to redo a lengthy simulation. With KLINGON, it was still necessary for the user to navigate back and forth, losing the previous output in order to modify its appearance.

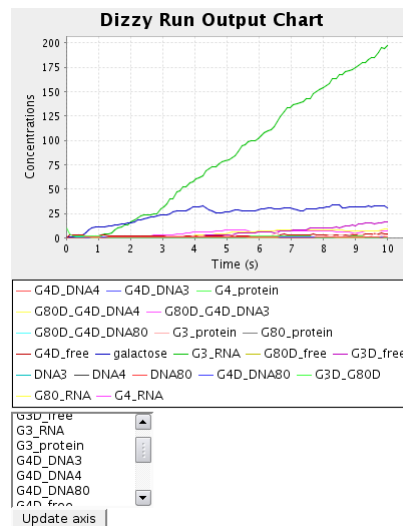


Figure 6.4: Chart generation with SHINY

Figure 6.4 shows the new selectbox available in SHINY to modify the chart on the fly. In this example, the chart is not very useful if evaluation of the differences between G80_RNA and G80_protein was needed, as these two values are far too small to be useful. The selectbox allows the user to select the desired axis and

have the chart updated. The “Update Axis” button calls a JavaScript function that forms a new image source URL for the desired new image, downloading it in the background and updating the source for the image when the download has completed. This means that the user can now observe the graph as they decide on the new datasets and also not lose this chart while the new one is updating.

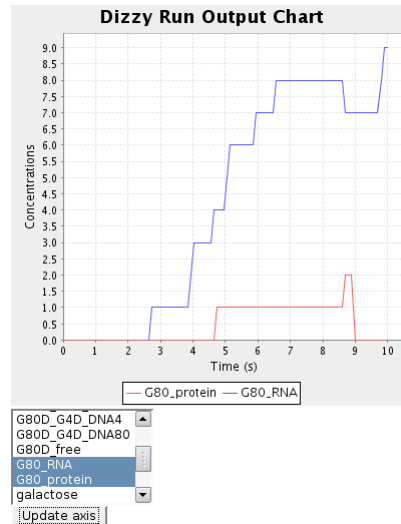


Figure 6.5: Chart update with SHINY

Figure 6.5 displays the result of the update. The comparison of the two datasets is far easier now, as the Y-Axis scale has been adjusted dynamically.

Also added was the ability to export charts as SVGs, allowing the chart to be dynamically resized after generation and therefore viewed or printed at high quality at any resolution, due to the nature of vector graphics, something lacking in the previous PNG format. The text in an SVG file, such as chart axes in this case, can be searched through and copied. Using an SVG editor such as Inkscape, these charts can be easily modified, annotated or edited; difficult with a PNG file. The SVG output library used generated large SVGs (in the above example 44KB compared with a 12KB PNG), Thankfully this was not an issue as it can be compressed automatically by the Web server and client’s Web browser, as most modern browsers support gzip compression, after which the PNG is 11KB but the SVG only 3.8KB (with default compression). The added SVG functionality also has many possible extensions that could not be currently implemented. These are explained further in Subsection 6.7.3 and Section 8.1.

6.3.1 Graceful Fallback

With disabled JavaScript the “Update Axis” button will simply update the current page, reloading everything, but also updating the image.

6.4 Save status without redirection

Web browsers, like any applications, are not without bugs. They are complex applications required to do increasing numbers of differing tasks, and increasing numbers concurrently. This is one of the reasons some Web browsers become unstable. A common problem with long-running user tasks using the Web browser is, on event of a crash, everything is lost.

The previous method used in KLINGON of creating and editing simulator files was shown to be problematic. It was necessary for a user to navigate away from the current page, losing focus in the file, every time they wanted to make a save. This naturally encourages users to not navigate away, and therefore, not to save. More technical users may type the file up in an application that allows saving first, before entering it into the browser, but this is not an ideal solution.

The previous sections have already shown that browsers with Ajax support can easily send and receive information from a server asynchronously, without breaking the user's workflow. For editing simulator files SHINY uses JavaScript to submit the contents of the editor, using a form, in the background, using the XMLHttpRequest object. This form data is sent to the server in exactly the same format as in KLINGON, but this way was done behind the scenes and did not require page navigation to return to work.

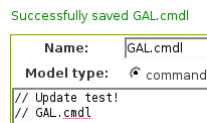


Figure 6.6: Simulator file saving in SHINY

Figure 6.6 shows the result of a successful save. The file has been updated and stored on the server, with no need to break the user's workflow. If the users were trusted even less, these updates could be made after a certain number of keypresses or fixed time periods.

6.4.1 Graceful Fallback

In event of the XMLHttpRequest object failing to be created or disabled JavaScript the file will saved in the same manner as in KLINGON, posting a form, redirecting to another page and requiring the user to navigate back.

6.5 Feedback for lengthy client/server interaction

The other way of adding files to the server is by traditional HTTP uploading. The previous KLINGON method works satisfactorily for small uploads, but for larger

uploads or slow Internet connections, the page will appear to hang until the file has finished uploading, and, as with the KLINGON's simulation runs, there is no indication as to how long this will take.

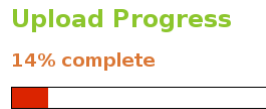


Figure 6.7: File upload in SHINY

The KLINGON backend code made use of an external module to handle file uploads. The latest beta of this module added the ability to associate a given upload with a progress listener. This was used to create an XML server for the upload progress. When a file is sent to the server the browser informs the server of the number of files being sent and the size of each file. The progress listener can then use this information, combined with the size of file already uploaded, to calculate the percentage through the upload. An XMLHttpRequest object is used with callback objects to asynchronously poll the XML server for the file upload status, and display a progress-bar on the page while the file is uploaded. A example of an upload in progress can be seen in Figure 6.7.

This method is less successful than the simulation run progress-bar, as uploading the file will make maximum use of either client or server bandwidth, so polling the progress from the server is far slower than with the simulation progress-bar, as the connection is being saturated. However, this method still allows a user to more easily estimate how long an upload will take and can be useful for large uploads.

6.5.1 Graceful Fallback

In event of the XMLHttpRequest object failing to be created or disabled JavaScript the progress-bar will simply not be created and the client will see the same as with KLINGON: the page appearing to be busy until the upload completes.

6.6 Input validation based on client-side data

Sometimes input validation on the client-side is actually more secure and sensible than using a server-side method. For example, when checking password data over a non-secure link, it is sensible to not send the password in plain-text over an untrusted network. Also, for basic checks it may be a waste of resources relying on the server to validate data, for example checking the similarity of two strings.

An example of this is the password update page. The user should be warned if the old and new passwords match or if the new password and the confirmation do not match. This was handled easily and efficiently using JavaScript. On

First name:	Mike
Surname:	Arthur
Old password:	***
New password:	*** New password matches old password!
Confirm new password:	**** The confirmation does not match the new password.
Update details	

Figure 6.8: Password validation in SHINY

every change of a relevant input box a JavaScript function is called to check the validity of the input and, if necessary, modify the DOM to insert warning text. An example of this can be seen in Figure 6.8.

First name:	Mike
Surname:	Arthur
Old password:	*****
New password:	*****
Confirm new password:	*****
Password Strength	<div style="width: 50%; background-color: yellow;"></div>
Update details	

Figure 6.9: Password strength validation in SHINY

With aid of a BSD-licensed password library, JavaScript was also used to perform slightly more advanced client-side tasks. A simple password strength meter was added, with a graphical progress-bar, to allow the user to see how strong the new password is, before they decide to update the details on the server. The password strength bar and form can be seen in Figure 6.9.

6.6.1 Graceful Fallback

With no JavaScript the client-side input validation is impossible so the user will not see any effects from any of the otherwise triggering criteria. The “Update Details” page will perform identically to KLINGON.

6.7 Problems Encountered

As with any program, some problems were encountered in the creation of SHINY. However, some of these problems are not specific to SHINY itself, and may be prevalent with the development of Ajax applications.

6.7.1 Design of previous system

The architecture of KLINGON was carefully thought out and planned to allow proper, secure access to the needed underlying elements of the application using

JavaBeans, reducing the need for the JSP to worry about the logic occurring in the chemical simulator, instead focusing on the UI and I/O with the user.

A problem that arrives early on with developing an Ajax application, and in this case took a large proportion of the development time is re-engineering the existing system to allow the new Ajax elements of the Web application to present meaningful data to the client. The simulation run's progress-bar, specifically required information that was previously only available in the lowest levels of the application, communicating directly with the logic of the chemical simulator. This meant that with the multi-tier architecture in place it became very difficult to retrieve this information cleanly for use in the browser. This is not a criticism of the previous code in KLINGON; the application was designed in an efficient and secure structure, but these very strengths actually worked against the implementation of the Ajax feedback. Fundamentally Ajax relies on the user being kept up-to-date with any actions, and allowing the user to incrementally access the backend logic, rather than simply submitting a request for a lengthy operation and receiving a result, as with a standard Web application. As mentioned in "AN ARCHITECTURAL STYLE FOR AJAX" [7], The approach to writing Ajax Web applications is more similar to writing a GUI desktop application with the added difficulties of multi-user access, security and distributed processing that occur from writing a Web application. Fundamentally, the strengths from both applications can be combined, but in order to do so an Ajax software engineer is usually also required to battle with the difficulties of both.

6.7.2 JavaScript

Ajax applications rely heavily on JavaScript for the development of the UI, the logic and the asynchronous communication with the server. However, using JavaScript brings new problems to application development.

6.7.2.1 Cross-browser compatibility

One of the main problems with all Web programming is that of cross-browser compatibility. Each of the main browser rendering engines (Opera, Webkit/KHTML, Gecko, Internet Explorer) have different quirks, their own unique features and bugs. In the case of Microsoft Internet Explorer, pages that render in other browsers perfectly frequently fail in Internet Explorer. JavaScript, sadly, is no different. The DOM is slightly different between browser implementations and, again, Internet Explorer is the main contender for problems, even with the latest version (Internet Explorer 7.0) failing to meet the DOM specification fully. Also, as mentioned earlier, Ajax makes use of CSS manipulation to style the page, and with the two main browsers (Mozilla Firefox and Microsoft Internet Explorer) failing to fully meet the CSS2 specification more difficulties arise.

Essentially, a truly cross platform Ajax application must be rigorously tested across all the main browsers and must try to detect the current running browser and use different code-paths depending on that browser's quirks and bugs. For this reason, in SHINY, browser compability was only checked with the latest stable versions of Mozilla Firefox (2.0.0.2) and KDE's Konqueror (3.5.6) available at the time of writing.

6.7.2.2 Debugging

As has been established, JavaScript code may need to be modified to work across different browsers. When trying to implement and test this code, where unexpected behaviour is found, it can be very difficult to debug. This is partly due to the nature of JavaScript. JavaScript is a weakly-typed interpreted language, and for programmers coming from strictly-typed or even compiled language backgrounds this can be a difficult transition. When JavaScript code fails in a standard browser, by default, rather than returning an error to the user, it simply stops executing. For example, a common stumbling block is trying to access a method in a DOM object that does not exist. This will simply cause the script to stop executing at this attempted method call. Mozilla Firefox and Microsoft Internet Explorer both have debuggers available, but these are not shipped with the default install of the application and are still in continual development.

Another minor "gotcha" in JavaScript debugging is that most browsers will, like any Web content, attempt to cache JavaScript. This means forcing the browser to not use its cache or forcefully flush it every time a script is modified. This is easily done during development, but with large-scale Ajax applications it can be hard to ensure all clients have the latest version of the JavaScript logic needed for the application.

6.7.2.3 Speed

This is a minor problem but being a interpreted language the source code for a JavaScript application must be fully downloaded before its execution, and executed dynamically inside the browser. As a result JavaScript is far slower than compiled languages, and logic in JavaScript will generally be far slower than logic executed on the server itself. Also, for large JavaScript applications the long download time can cause a noticeable delay in the execution of an application.

6.7.2.4 Security

JavaScript applications are server-provided applications to a client, and are transparently run without the users' request (by default) on accessing a page. As SHINY has demonstrated, JavaScript can be used to send and receive files from a server, connect to external URLs and perform large calculations that may cause the browser to lock up. With modern browsers it can be difficult to arbitrarily

access files from disk without the user's consent, but cookies, for example, may store confidential information and could be scanned, processed and uploaded to a server if containing any information useful to the malicious provider of the script. Cross-site scripting (XSS) is a broad term used to describe some of these attacks, allowing an attacker to do anything from bypassing the browser's sandbox, accessing local files in Internet Explorer, to stealing passwords stored in browser session cookies.

6.7.3 SVG

SVG is a language for describing 2D graphics in XML. Their primary use is in vector graphics, but they can also embed raster graphics and text. These graphics can also be interactive and scriptable, much like traditional XHTML and can be embedded in browsers. They may be slightly larger than raster equivalents, but when using gzip compression, commonly used by both servers and browsers, they are almost always smaller. SVG is an open format and the specification maintained by the W3C.

SVGs are promising contenders in the future development of Ajax applications. So much of Ajax relies on XML and JavaScript technologies, both of which can be easily utilised both within and dealing with SVGs. SVGs can contain JavaScript within the file, animation, hyperlinks and are valid parsable XML.

SHINY has basic SVG support, allowing the export of charts to SVG format. However, this was intended to be more fully-formed, but sadly some of the current problems with SVG hampered the efforts made. Currently the main problem with SVG is the simple lack of complete support across browsers. Most of the main browsers now support SVG in some form, but none completely implement the SVG Full 1.2 specification, the W3Cs most comprehensive SVG specification. Sadly this means that using SVG in Ajax is currently fairly buggy and difficult. Potentially the goal for SVGs in this project was to allow each chart axis to be downloaded dynamically from the server when needed, however, this would have required more full support for JavaScript inside the SVG format, not seemingly possible across the browsers used. This technology is still in its infancy and with the growing popularity of Ajax applications, SVG could well see its place inside Ajax, allowing Ajax to be a serious competitor for applications currently implemented as Adobe Flash or applets, with the added animation and graphical support SVG provides.

Chapter 7. Evaluation

Chapter 4 on page 13 set an explicit hypothesis to test, and four criteria used to evaluate its validity.

7.1 Learnability

For an application to be easily learnable it should seem to be deterministic, so that every user action has an obvious, expected and consistent response. For SHINY, as the application is being run in a Web browser, the expected response is for the application to behave in similar manner as other Web applications.

The newly introduced feedback for long-running tasks would not be familiar to a user who has never used an Ajax application, but as this merely presents information on progress rather than information that the user needs to process this does not detract from the learnability of the application. Classic Web applications don't make use of extensive long-running server-side processing, so the progress indication could provide indication that the application hasn't crashed or "broken". The presentation of the progress-bars naturally leads to comparison between those of GUI desktop applications performing long-running operations, and a user who had downloaded files from the Internet or copied files to removable media would be familiar with the concept of progress-bars.

The input validation may be slightly confusing to a new user as the error messages may seem to appear and disappear without the user noticing, causing confusion when they later encounter them. Classic Web and GUI desktop applications tend to respond to errors in forms when the form is completed rather than while the form is in progress. This may make an Ajax application slightly harder to learn, however, these provide far greater response feedback than the previous methods used, and the error generation is predictable and deterministic, so after a few errors the user will quickly realise the difference with this type of application.

The modification of on-screen datasets (in this case, charts) without page reloading is also a novel concept to users experienced only with classic Web applications. However, this is used constantly in GUI desktop applications, so it may be slightly unexpected at first but should not break the flow of the application or the work. This applies equally with the saving of status without navigation (in this case the contents of a file). Were there no feedback on the operation the user may become confused as to why the action they took has not caused the traditional click-load-newpage cycle expected in a browser, but a status message is updated, informing the user of their last action's result and success.

Ajax applications are becoming more commonplace, with sites like YouTube, Flickr, Facebook and Google Maps used regularly by average Web users. To a user

unfamiliar with these applications the instant response and animated feedback may be initially confusing, but as the method of input is still the same as classic Web applications this confusion will rapidly pass, and the new Ajax applications provide more rapid feedback on user input and therefore a faster interface.

7.2 Flexibility

A flexible application should allow the automation of routine tasks, support for simultaneous tasks and give the user control of the task execution.

The progress-bar introduced in SHINY allows the user to perform other tasks, such as editing files, whilst keeping up to date with the progress of the current simulation. It also allows a user to stop a current run. In addition, throughout the application there is automation of tasks such as repetitive form submission or moving back and forward to perform input validation, change a dataset or submit some data to the server. This allows the user's actions to be preempted, providing information in less time than it would take them to request it traditionally. However, a problem with this method is that the user is not in complete control of task execution, as many of the input validation operations occur without any direct request from the user, merely on input.



Figure 7.1: Moving an object in Google Customised Homepage

Most Ajax applications are similar in this regard, second-guessing the users current task to provide shortcuts to its completion. In SHINY, due to the simplistic and short nature of tasks, this works well as there is only usually one path the user can take to complete an action but other applications may result in a user having to fight with the interface in order to perform a task in a way that was not expected by the designers of the application. In SHINY the interface is not made customisable to the user, but applications such as the Google Customised Home and Google Maps use Ajax technologies to provide interface customisation, as can be seen in Figure 7.1.

7.3 Robustness

An application's robustness can be shown through its indication of its past, present and possible future states, the ability to undo errors, responsiveness and providing sufficient functionality to conform to user tasks.

Moving a GUI desktop application to a Web browser automatically gives it a state-based architecture, allowing navigation between these states and a history indicating the user's movement through these states. Also, with classic Web applications, this allows simple errors such as a misdirected click to be easily remedied: simply click the "Back" button. However, with Ajax applications this becomes more difficult. This was highlighted earlier, instead referring to applets, in "NAVIGATING THE APPLLET-BROWSER DIVIDE" [10] and "AJAX: HOW TO HANDLE BOOKMARKS AND BACK BUTTONS" [9]. Users of Web applications instinctively press the "Back" button to try and undo an action or return backwards in a process. Ajax applications rely heavily on JavaScript running on each page, manually updating the DOM, and this information is not stored in the browsers history stack, meaning that moving backwards through the history a user of a Web application may not see what they were expecting. In SHINY this is relatively minor, and only clicking back to progress-bar pages may present some confusion, as the progress-bars simply do not appear, or appear to be not progressing, the latter should not be confusing to the user as the run has already been visibly completed.

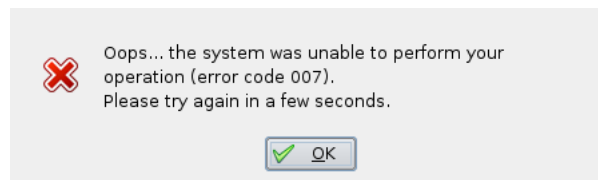


Figure 7.2: Going back with GMail

With larger, more dynamic Ajax applications such as GMail, this presents more of a problem. For example, once logged in to a GMail account, a user would find themselves at the "Inbox". If they decide to navigate to the "All Mail" folder they simply click on the link on the navigation bar. However, as this is an Ajax application this new folder is loaded dynamically using Ajax, so the address bar does not change. If the user then decides to navigate backwards to the "Inbox", they may instinctively press the "Back" button. With limited testing it was seen that this did not always return to the "Inbox" sometimes returning to the login screen and sometimes even "breaking" the application, returning the confusing error seen in Figure 7.2. In this strange situation, clicking the "Forward" button again results in the same error, with the only way to return to the "Inbox" again being to keep pressing the "Back" button until the login screen is seen and logging in once more.

As mentioned in the last chapter, Ajax solutions tend to use lots of small requests for data, rather than re-requesting the page. This allows Ajax applications to provide more responsive interfaces, both to user input and when changing datasets is required, as only the new data need be transferred. A possible criticism is that this requires bandwidth, but the total bandwidth used is far less when using an Ajax method than a classic Web application.

Providing the needed functionality in an Ajax application is simply a matter of implementation. It may require more to implement, but ultimately the sky is the limit as far as Ajax is concerned, as interfaces can be created that are far more dynamic than a classic Web application and may equal the usability of a GUI desktop application.

7.4 Time Affordances

Evaluating Time Affordances in an application requires examining possible and forced delays and reassurance on long running operations completion. The reassurance on long task completion have already been covered in Section 7.1 on page 37, and delays and application speed in Section 7.3 on the previous page.

Essentially, Ajax Web applications allow the creator to provide to the user a method of input equally as quick as a GUI desktop application and provide faster data transfer, due to the lighter overhead of using XML to transfer data than an HTML page with all the other content that is needed. This essentially allows the browser to not only cache the images and stylesheets for a page but actually some of the other content (such as navigation bars) by simply modifying the current page rather than reloading it in its entirety. Also provided is the ability for some processing of data and input validation to be performed on the client rather than server-side, further reducing the latency in making requests and bringing the speed of the application closer still to that of a GUI desktop application, whilst still providing the benefits of a Web application. With an application such as SHINY, with a powerful server performing the simulations, this may be far faster than using DIZZY on the local machine.

Chapter 8. Further Work

Some areas of this project have been limited by time, expertise or technical infrastructure. Increased resources in any of these areas could be used to perform further work related to this project.

8.1 SVG

As previously mentioned, SVG is a vector graphic format that can be embedded in Webpages. SVGs can potentially be used to create event-driven graphics and animations, allow dynamic retrieval, scripting and animation. It is an open standard, and the tools to create, view and edit SVGs are freely available, with both open-source and proprietary solutions.

The current problem with SVG is the lack of browser support. No browser fully meets the latest specification and the most commonly used browser (Microsoft Internet Explorer) has no native SVG support. As a result, this technology does not have much usage on the Web, with most SVGs used for vector art offline, such as icons or diagrams.

With increased support and compatibility between browsers, SVG could potentially allow for even more rich interfaces than Ajax alone allows, and Ajax can be used within SVGs for dynamic content retrieval. Furthermore, SVG, with its ability to be scripted and animated, could feasibly topple Adobe Flash as a tool for creating dynamic, animated Websites while allowing pages to be individually bookmarked and give a more browser-native feel to dynamic applications, common criticisms of Websites using Adobe Flash.

8.2 More vigorous usability testing

This project did not make use of user-based usability validation, instead relying on guideline and task based evaluation methods. A more in depth user-focused usability study could make aware of some of the more subtle effects of Ajax on usability, such as how likely users are to attempt to use the “Back” and “Forward” buttons. With SHINY the changes made and added functionality was sufficiently small to allow guideline based evaluation to be sufficient, but a comparative evaluation of GMail or another Ajax version of a common, complicated GUI desktop application, such as a word processor or spreadsheet, could be beneficial, allowing developers to ensure their focus is directed to any common stumbling-blocks users encounter when trying to use Ajax applications in a working environment.

8.3 Testing under poor network conditions

An examination of the negative implications for Ajax of a low-bandwidth connection was discussed in this report, but not that of other conditions such as packet loss or high latency. High latency in particular is an area in which Ajax applications could potentially suffer greatly compared with classic Web applications. Due to insufficient infrastructure available to test this further, this report does examine the effects under these conditions, however the frequent small update structure usually used in Ajax applications will result in a sluggish or “broken” application with poor network conditions, so a formal quantitative analysis of this could be beneficial for evaluating the potential for Ajax replacements for GUI desktop applications.

8.4 Security

A common criticism of the wave of Ajax applications and the increased use of these to replace GUI desktop applications is the perceived lack of attention to security. Previously mentioned are the possibilities of cross-site scripting vulnerabilities in JavaScript but also the weakly-typed nature of JavaScript, the dependence on client checking the client data, the inability to prevent modification of the local copy of JavaScript and poor error handling all create potential security flaws, bringing security risks to the server and/or the client. A recent virus named “Yamanner” spread through the Yahoo! Mail service, sending the contents of a targeted user’s address book to a remote server.

Clearly there are potential and real security problems with Ajax, but a formal evaluation of these problems and how they can be prevented by server administrators, clients and Web application developers is beyond the scope of this report, but could be useful as a tool for those developing Ajax applications.

8.5 Comet

Comet is a further evolution of Ajax. Where Ajax relies on a polling loop to check for changes on a server and to mimic event-driven behaviour. This model is flawed when the client must wait for an event to occur on the server, as it relies on the client checking for the event before it can be handled or detected. A better solution to this problem would be allowing the server to notify the client on the event, but this is not possible with Ajax. Comet is a technique that, rather than using a polling loop keeps a consistent HTTP connection between client and server, allowing the server to send data to the client on an event without the client requesting it.

Comet was, at the planning stages of this project, very much a bleeding-edge technology. During this project more frameworks have become available

and Comet is becoming a viable option for event-based user interfaces. Some of the concerns still remaining with Comet are on its scalability, with each client requiring a long HTTP connection to the server, and existing Web servers are not designed for such a large number of connections.

SHINY's progress feedback relied on the client checking for new data from the server, and sometimes it would not be available. With Comet this could have occurred whenever the server had new data, rather than the client unnecessarily polling, resulting in lower bandwidth usage and also a more smooth progress indication mechanism.

Chapter 9. Conclusion

9.1 Hypothesis

To restate the hypothesis of this project:

“Ajax applications can provide a more effective interface user interface than those of GUI desktop or classic Web applications with fewer drawbacks than either individual approach.”

The conclusion of this project, as expected, is not a clear agreement or contradiction of the hypothesis, but more complex. The key outcomes of this report and the project are split into the benefits and drawbacks of using an Ajax application.

9.2 Benefits

Ajax applications, due to the nature of their interaction with the server, can provide a far more responsive and rich user interface. Users can now drag and drop items, be notified on server events without requesting, and transform datasets on the fly without the need to reload the page. Their input can be quickly regulated and automatically corrected, minimising the use of bandwidth. Lengthy interactions with the server can have their progress indicated, whilst allowing other tasks to continue, allowing better multitasking.

The application developer can benefit through the ability to offload some of the application logic and processing to the client and needing to send only required data, rather than repeatedly send the same stylistic information. Also, the creation of these applications becomes increasingly less complex with the addition of new Ajax libraries to aid the developer.

9.3 Drawbacks

One of the biggest struggles with Ajax development are the difficulties arising from the use of JavaScript. JavaScript suffers from its weak-typing, security risks and different implementations across browsers and also that some users may have disabled JavaScript within their browsers. In addition, debugging can be very difficult and time consuming as Ajax applications frequently make use of multithreaded JavaScript, resulting in race conditions and obscure bugs. For logic implementation, JavaScript, as an interpreted language, is frequently slow and can use a lot of memory, leaving client machines somewhat unresponsive. In addition, the nature of an application in a browser means that other potentially

“buggy” sites may crash the browser and cause loss of work by the user. A minor issue is that of retaining backwards compatibility. An Ajax application must be accessible to clients without JavaScript, those with text-based browsers or search engines.

Ajax’s lack of browser compatibility is evident from the need for “hacks” to make the same code work in different browsers and how easily Ajax applications “break” common functionality such as the “Back” button. This leads onto an argument growing somewhat less valid now; Ajax applications are unfamiliar to a user of classic Web applications and can cause confusion.

The largest development pitfall encountered in the development of SHINY was the difference in architecture between Ajax and classic Web applications. With Ajax applications, the logic needs to be easily manipulated and accessed by the Ajax code, and long running tasks buried deep within an application need to be made accessible to a user interface, necessitating large amounts of re-engineering in order to access this.

9.4 Contribution

This report has provided a glimpse into some of the usability benefits and drawbacks of Ajax Web applications. The usability study provided between DIZZY, KLINGON and SHINY could be used to evaluate moving a classic Web application or GUI desktop application onto the Web. For creating new Web applications, Ajax can be a powerful tool in providing more usable, intuitive and responsive user interfaces. However, porting existing Web applications to Ajax could prove to be very time consuming and difficult and therefore must be analysed carefully before implementation. There are no clear benefits of Ajax application interfaces over those GUI desktop applications, but the benefits of distributed storage, computing and portability can be counted in the favour of Ajax. Technologies such as SVG and Comet are likely to push the boundaries for these applications even further.

Ajax is a new technology that has captured the imagination of many application developers. The dream of the Internet as an application platform may be realised and as browser support and libraries make development easier and users become more familiar with the new interfaces, Ajax Web applications will provide more services to users in an easy-to-use manner from anywhere with an Internet connection, without the requirement of a plug-in.

Bibliography

- [1] D. Best and T.U. Eindhoven. Web 2.0 Next Big Thing or Next Big Internet Bubble? January 2006.
<http://page.mi.fu-berlin.de/~best/uni/WIS/Web2.pdf>.
- [2] Institute for Systems Biology. Dizzy home page.
<http://magnet.systemsbiology.net/software/Dizzy/>.
- [3] J.J. Garret. Ajax: A New Approach to Web Applications, February 2005.
<http://www.adaptivepath.com/publications/essays/archives/000385.php>.
- [4] B. Gibson. DHTML accessibility: solving the JavaScript accessibility problem. *Proceedings of the 7th international ACM SIGACCESS conference on Computers and accessibility*, pages 202–203, 2005.
<http://doi.acm.org/10.1145/1090785.1090830>.
- [5] R. Khare. Beyond AJAX. August 2005.
<http://www.knownow.com/products/docs/whitepapers/KN-Beyond-AJAX.pdf>.
- [6] J. MacKenzie, A. McAlister, S. Desai, and K. McCarthy. Usability in Web Design. October 2006. <http://osddp.org/files/issues/Usability.pdf>.
- [7] A. Mesbah and A. van Deursen. An Architectural Style for Ajax. *Arxiv preprint cs.SE/0608111*, 2006. <http://arxiv.org/pdf/cs.SE/0608111>.
- [8] T.P. Moran and S. Zhai. Beyond the Desktop Metaphor in Seven Dimensions. September 2006.
<http://www.almaden.ibm.com/u/zhai/papers/DesktopMoranZhai.pdf>.
- [9] B. Neuberg. Ajax: How to handle bookmarks and back buttons, October 2005. <http://www.onjava.com/pub/a/onjava/2005/10/26/ajax-handling-bookmarks-and-back-button.html>.
- [10] J. Nielsen. Navigating the Applet–Browser Divide. September 1997.
<http://ieeexplore.ieee.org/iel4/52/13290/00605926.pdf>.
- [11] D.A. Norman. Emotionally centered design. *interactions*, 13(3), May 2006.
<http://doi.acm.org/10.1145/1125864.1125894>.
- [12] J. Oberlander. Human-computer interaction, 2006.
<http://www.inf.ed.ac.uk/teaching/courses/hci/>.
- [13] W3C. Web content accessibility guidelines 1.0, May 1999.
<http://www.w3.org/TR/WAI-WEBCONTENT/>.

Chapter 10. Appendix

10.1 Glossary

- **ADOBE FLASH:** Flash allows a developer, with aid of a plug-in in a client browser and Adobe development tools, to create a application that runs in a browser window providing animation, video, interactivity and vector graphics to a user.
- **AJAX:** Asynchronous JavaScript and XML is a technique for creating Web applications, using existing technologies to improve responsiveness with more small interchanges of data, rather than simply reloading on user changes. Uses XHTML, DOM, XMLHttpRequest, JavaScript and XML to perform this.
- **APPLET:** An applet, in the context of this report, refers specifically to a Java applet, which is a software application that can run within the Web browser, with a plug-in providing a virtual machine running Java code.
- **CSS:** Cascading Style Sheets is a language used with HTML or XHTML to describe the appearance of a document, allowing this to be separated from document content.
- **DHTML:** Dynamic HTML, like Ajax, is a technique for creating more interactive Web-pages, by use of JavaScript to allow the interface to appear to change without reload. This phrase was used before Ajax, and is rarely used now, instead being superceded by Ajax.
- **DOM:** Document Object Model is a representation of an XML or HTML document as a tree, and is used by JavaScript in manipulating the document structure or content.
- **HTML:** HyperText Markup Language is the language used in creating Web pages, describing style elements for basic text, by annotating the text with text markup elements.
- **JAVASCRIPT:** This is the implementation of the ECMAScript standard, and is implemented in Web browsers to allow scripting of Web pages by page creators. Confusingly, it is only distantly related to Java, and bears little resemblance outside of syntax.
- **RIA:** Rich Internet Applications are Web applications that provide an interface and features similar to those provided by a GUI desktop application, but generally, perform most of the processing on the server.

- SVG: Scalable Vector Graphics is an XML language for 2D graphics, usually used for vector images.
- W3C: The World Wide Web Consortium is the main standards body for the World Wide Web, creating and maintaining standards for HTML, XHTML, SVG and XML and others.
- WEB 2.0: Web 2.0 refers to the the recent growth in collaborative Internet services and Web applications, frequently making use of Ajax techniques to provide a more interactive and desktop-like user interface. It can also describe the growth of GUI desktop applications ported to Web applications providing the same functionality.
- XHTML: Extensible HTML is very similar to HTML, but, as an application of XML, is more restrictive, and allows easier validation and parsing. It is considered to be the latest version of HTML.
- XML: The Extensible Markup Language is a general purpose language used for sharing data across different information systems easily, particularly the Internet. XHTML and SVG are two examples of XML.
- XMLHttpRequest: XMLHttpRequest is an API used by JavaScript to transfer data to or from a Web server using HTTP. It usually returns XML data, and allows the use of the Ajax technique in the browsers that support it.
- XSLT: Extensible Stylesheet Language Transformations are an XML-based language for transforming XML documents into other XML documents. It is frequently involved in changing the structure or appearance of an XML document, or translating XML applications, such as XHTML.