Ajax Live Regions: Chat as a Case Example

Peter Thiessen
Adaptive Technology Resource Centre
University of Toronto
130 St. George St., Toronto, Ontario
peter.thiessen@utoronto.ca

Charles Chen
Empirical Software Engineering Lab
The University of Texas at Austin
713-557-7289
clc@clcworld.net

ABSTRACT
Web 2.0 enabled by the Ajax architecture has given rise to a new level of user interactivity through web browsers. Many new and extremely popular Web applications have been introduced such as Google Maps, Google Docs, Flickr, and so on. Ajax Toolkits such as Dojo allow web developers to build Web 2.0 applications quickly and with little effort. Unfortunately, the accessibility support in most toolkits and Ajax applications overall is lacking. WAI-ARIA markup for live regions presents a solution to making these applications accessible. A chat example is presented that shows the live regions in action and demonstrates several limitations of ARIA live regions.

Categories and Subject Descriptors
H.1.2 [Models and Principles]: User/ Machine Systems—*human factors, human information processing*; **K.4.2 [Computers and Society**]: Social Issues—*assistive technologies for persons with disabilities*

General Terms
Human Factors, Design, User Agents

Keywords
Accessibility, Web 2.0, Ajax, ARIA, Live Regions, User Agents

1. INTRODUCTION
With the advent of Web 2.0 and Ajax, web applications can now provide a level of interactivity that can rival traditional desktop applications. Many new and extremely popular Ajax applications have been introduced such as Google Maps [6], Google Docs [7], and Flickr [20]. However, Web 2.0 poses a new problem for screen readers and other similar assistive technologies. Part of the Web 2.0 advancement is the ability to access and modify the DOM of a XHTML document and not have a full page refresh. Modifying a DOM element creates a look-and-feel similar to a desktop application and has allowed for useful features such as drag-and-drop XHTML document elements.

Traditionally, Assistive Technologies (AT) have treated information on a web page as content that can be linearized. Ajax web applications break this assumption; new content can appear in arbitrary locations and user interactions with the page are far more complex. Since these Ajax web applications behave more like desktop applications than web pages, solutions for making desktop applications accessible can be applied to these Ajax web applications. One of the most important aspects of making desktop applications accessible is to inform users of important events that are occurring on parts of the screen, even if those parts are not focused. For example, in a chat application, the user's focus is on the input blank, but it is essential to inform the user of what the other chatters have typed. On the other hand, it is important not to overwhelm the user with a flood of information, especially if that information is trivial.

In traditional desktop applications, there are a set of known widgets such as buttons, trees, data cells, etc. These widgets behave in a predictable manner; thus, an AT simply needs to know how to support the events of a type of widget in order to provide reasonable support for any instance of that type of widget. However, Ajax applications do not share this uniformity. Many Ajax applications use custom widgets created out of span and div elements, mixed with input elements and graphics, and laid out by CSS. Sometimes, Ajax applications will even use custom widgets for standard HTML widgets such as a button because the application developer wished to change the behavior and/or appearance of that widget to fit the particular application better. As a result, while AT can pick up DOM mutation events, it is very difficult, if not impossible, for an AT to understand what that event represents in the context of an Ajax application.

We developed an Accessible Ajax chat application called Reef Chat. The goal of this chat is to demonstrate that a Rich Internet Application (RIA) could be both accessible and aesthetically appealing. Reef Chat has currently reached the first phase of development and has the basic functionality of a typical accessible chat application. The chat is targeted to work specifically with screen readers and follows the WAI-ARIA [15] guidelines and more specifically, the ARIA live regions markup to expose chat events to the DOM. The future roadmap of Reef Chat includes converting it into a Dojo [4] widget and adding more graphical components to improve the user experience for sighted users.

Dojo [4] is an open source Ajax toolkit that allows developers to create Ajax Web applications without having to worry about details (such as cross browser functionality) that are often a problem when using Javascript.

Reef Chat was developed to work with the Fire Vox[5] screen reader and for the purpose of this paper, to demonstrate how ARIA live regions can be used to develop a highly interactive Web 2.0 Internet application. Hence, we see our contribution as being both a proof of concept/demo and a discussion of "lessons learned".

W4A2007- Technical Paper , May 07–08, 2007, Banff, Canada. Co-Located with the 16th
International World Wide Web Conference.

Charles Chen is the creator of the CLC-4-TTS Suite [5]. One of the applications in this suite is Fire Vox, an open-source, freely-available, talking browser extension for the Firefox web browser. It is essentially a screen reader that is designed for Firefox. It is cross-platform compatible and can run on Windows, Macintosh, and Linux.

The organization of the paper is as follows. In section 2, we describe the markup for ARIA Live Regions. In section 3, we discuss several ARIA test cases. In section 4, we demonstrate Reef Chat using the Fire Fov screen reader as an ARIA test case example, as well as the problems encountered and the explanation of design decisions in the development of Reef Chat. In section 5, we discuss potential improvements to Fire Vox and Reef Chat. Finally, in section 6, we discuss related work, followed by the conclusion in section 7.

2. WAI-ARIA MARKUP FOR AJAX LIVE REGIONS

2.1 WAI-ARIA

The solution to the DOM accessibility problem is to markup the live regions, the regions on the page which can be changed by Ajax. Markup for live regions is part of the Web Accessibility Initiative - Accessible Rich Internet Applications guideline (WAI-ARIA) [15]. Live regions are only one part of ARIA, other parts enable desktop-style Javascript widgets, specify typing restrictions on data, or mark regions of a page with landmarks (such as the main content). Future versions of ARIA are expected to allow accessible diagrams as well as author-defined roles, properties and relations. The only part of ARIA that is currently supported by Fire Vox is the live region markup. Window-Eyes and JAWS support the widget-related markup, but not the live region markup. This is the limitation of ARIA support in current AT products.

There is ARIA-ROLE and ARIA-STATE. State is the most important. Role is secondary when it comes to live regions, but is useful because it includes higher level roles like "log" and "status". Specifically:

- Roles for Accessible Rich Internet Applications (ARIA Roles) [16]: In partial fulfillment of ARIA Roadmap, describes mappings of user interface controls and navigation to accessibility APIs on different platforms.
- States and Properties Module for Accessible Rich Internet Applications (ARIA States and Properties) [17]: In partial fulfillment of ARIA Roadmap, enables XML languages to add information about the behavior of elements.

The ARIA set of specifications, which is currently still a working draft, suggests adding markup to the live regions of a XHTML document to help solve the issue of what should be done with DOM mutation events. By looking at the markup for a live region, an AT can understand what should be done when DOM mutation events are fired for that region.

The following subsections present the properties for live regions, discuss the issues with the current set of properties, and describe the existing support in Fire Vox for WAI-ARIA markup for live regions.

2.2 live=POLITENESS_SETTING

live=POLITENESS_SETTING is used to set the priority with which AT should treat updates to live regions. These are only the default priority settings for live regions; AT may provide ways for users to override/change these priority settings.

Table 1. live=POLITENESS_SETTING

Setting	Description
live="off"	This is the default. Any updates made to it should not be announced to the user. live="off" would be a sensible setting for things that update very frequently such as timers that change every second.
live="polite"	The region is live, but updates made to it should only be announced if the user is not currently doing anything. live="polite" should be used in most situations involving live regions that present new information to users, such as updating news headlines.
live="assertive"	The region is live. Updates made to it are important enough to be announced to the user as soon as possible, but it is not necessary to immediately interrupt the user. live="assertive" should be used if there is information that a user should know about right away, for example, warning messages in a form that does validation on the fly.
live="rude"	The region is live. Updates to it are extremely important. In fact, the updates are so important that the user must be interrupted immediately. live="rude" should be used sparingly and only with great consideration as it can be very annoying to users.

2.3 controls=[IDLIST]

controls=[IDLIST] is used to associate a control with the regions that it controls.

Table 2. controls=[IDLIST]

Setting	Description
controls="myRegion1 myRegion2 etcEtcEtc"	controls=[IDLIST] associates an element with one or more regions that it controls. If it controls more than one region, the regions are separated by a space. When a change to one of these regions occurs because of a user action on the control, then the change should be announced immediately to let users know that their action did have an effect.

2.4 atomic=BOOLEAN

atomic=BOOLEAN is used to set whether or not the AT should present the live region as a whole. This is only the default setting for the live region; AT may provide ways for users to override whether or not the live region is treated as atomic.

Table 3. atomic=BOOLEAN

Setting	Description
atomic="false"	This is the default. It means that when there is a change in the region, that change can be presented on its own; the AT should not present the entire region. atomic="false" is generally a good idea as it presents users with only changes and does not cause them to hear repetitive information that has not changed. However, web developers should take care that the changed information, when presented by itself, can still be understood and contextualized by the user.
atomic="true"	If atomic is set to "true", it means that the region must be presented as a whole; when there is a change, the AT should present the entire region, not just the change. atomic="true" can make it harder for users to understand changes as the changed areas are not presented independently. atomic="true" can also be annoying as it can force users to listen to repetitive information that has not changed. However, atomic="true" is necessary in cases where the change, when presented by itself, cannot be understood and contexualized by the user.

2.5 labelledby=[IDLIST]

labelledby=[IDLIST] is used to associate a region with its labels.

Table 4. labelledby=[IDLIST]

Setting	Description
labelledby="myLabel1 myLabel2 etcEtcEtc"	labelledby=[IDLIST] associates one or more elements that serve as labels with the live region that they label. These elements do not have to be HTML <label> elements. If there is more than one label, the labels are separated by a space. The labels should be presented to the user when there is a change to the region that they are associated with.

2.6 describedby=[IDLIST]

describedby=[IDLIST] is used to associate a region with its descriptions.

Table 5. describedby=[IDLIST]

Setting	Description
describedby="myDesc1 myDesc2 etcEtcEtc"	describedby=[IDLIST] associates one or more elements that serve as descriptions with live region that they describe. If there is more than one description, the descriptions are separated by a space. The descriptions should not be presented to the user when there is a change to the region that they are associated with as they are likely to be too lengthy and would annoy the user; however, there should be an easy way for users to find the description for a particular region when they want to find out more about the region.

2.7 relevant=[LIST_OF_CHANGES]

relevant=[LIST_OF_CHANGES] is used to set what types of changes are relevant to a live region. Multiple types of changes can be listed as relevant; the types are separated by a space. The default is relevant="additions text".

Table 6. relevant=[LIST_OF_CHANGES]

Setting	Description
relevant="additions"	relevant="additions" states that the insertion of nodes to the live region should be considered relevant.
relevant="removals"	relevant="removals" states that the removal of nodes from the live region should be considered relevant.
relevant="text"	relevant="text" states that changes to the text of nodes that already exist in the live region should be considered relevant.
relevant="all"	relevant="all" states that all changes to the live region should be considered relevant. This is the same as doing relevant="additions removals text".

2.8 Issues

The WAI-ARIA markup for live regions does still have a few issues to be worked out. These include difficulties with determining causality, giving developers the ability to group updates, handling interim updates, and providing higher-level abstractions for web developers.

Although WAI-ARIA has a controls=[IDLIST] property to specify that a control will change certain live regions, if these live regions can be changed by world events, then the AT will not be able to distinguish between a change caused by the user and one that is not. This can be an important distinction since changes caused by the user should be spoken immediately to let users know that their actions did have an effect; however, if the change was caused by world events, then the change should be

announced according to the appropriate politeness setting for that region.

Sometimes, web developers may have an application that needs to update several pieces of information at the same time. If these updates are expected to take a noticeable amount of time, web developers will need a way to tell the AT when the updates are completed and ready to be spoken. By grouping these updates together, web developers can prevent users from hearing the same information multiple times, as well as making a large update more meaningful by having all of its parts presented together. The current WAI-ARIA specification does not provide web developers with this ability.

In most cases, the AT should not announce something that is not currently displayed on the page since, in general, if it is not displayed anymore, then it is not current – skipping it will help prevent users from falling behind. However, there are cases where obsolete items should still be announced. For example, if an Ajax application provided a play-by-play description of a game in the form of a log that only contained the last 5 plays, then all the updates should be read, even if the AT were to fall behind in trying to read all of the updates and a play disappeared from the 5 current plays on the page before it could be read. It is important in this case to not skip updates that have since disappeared because they contain important information that the user needs to hear in order to make sense of the most current information. There is currently no way for web developers to specify whether or not interim changes are relevant.

Finally, the WAI-ARIA specification does not have defaults for the live region properties for the roles that it has defined. Having defaults is important as this would give web developers a higher level of abstraction. Rather than trying to manually specify all of the live region properties for each individual widget, web developers should be able to specify what type of widget they have and expect that there be reasonable defaults for how that widget will behave.

2.9 Implementation and Support for ARIA

Overall, the WAI-ARIA markup approach is a promising solution to the issue of Ajax live region accessibility. Rather than forcing web developers to completely redesign their Ajax applications, it simply asks that web developers make clear their intentions regarding the various changing parts of a page by adding some markup that will provide guidelines to the AT about how the changes should be presented to the user. Although the WAI-ARIA markup for live regions is quite new (the first draft did not come out until September of 2006), it is already being supported by Fire Vox.

Currently, Fire Vox supports live=POLITENESS_SETTING, atomic=BOOLEAN, and relevant=[LIST_OF_CHANGES]. In addition, Fire Vox also supports the use of "interim" to allow web developers to specify whether or not interim changes are relevant for a particular live region as mentioned in the previous section.

In addition to supporting the WAI-ARIA markup, Fire Vox also has a "smart" default mode which will try to guess the most suitable behavior for live regions that are not tagged with WAI-ARIA. While untagged pages do not perform as well as they could if they were to be explicitly tagged, these heuristics have so-far proven reasonable in a number of untagged real-world pages, such as Yahoo Finance [19], where page information is being constantly updated. Fire Vox also offers a strict mode that uses WAI-ARIA tagged regions only and an off mode that completely silences all live regions.

3. ARIA TEST CASES

The ARIA test cases at http://accessibleajax.clcworld.net include web application examples such as a form that validates input as it is being entered, a chatroom (with bots), and a scoreboard.

The form example shows a typical online form for a fictitious fan club called the "CLC Fan Club". If a user enters something that is not allowed (such as a user name with whitespace or passwords that do not match), an error message will appear. Once a user has successfully completed the sign up process, a certificate that includes the user name and date will be displayed. Both the error messages and the certificate will be announced when they appear.

The chatroom example allows the user to chat with scripted chatbots. As the messages appear, they will be read to the user. In addition, if a user tries to do something which is not allowed, such as send a blank message or a message that is too long, an error message will appear. This error message will also be read to the user.

The scoreboard example shows a 4-on-4 sports game. As the game progresses, the points and the player stats change. Sometimes, multiple things can change at the same time, for example, if one player scores after being assisted by another player. In that case, there are 3 changes: the score, the number of points scored by the player, and the number of assists by her teammate. All of these changes are read out to the user.

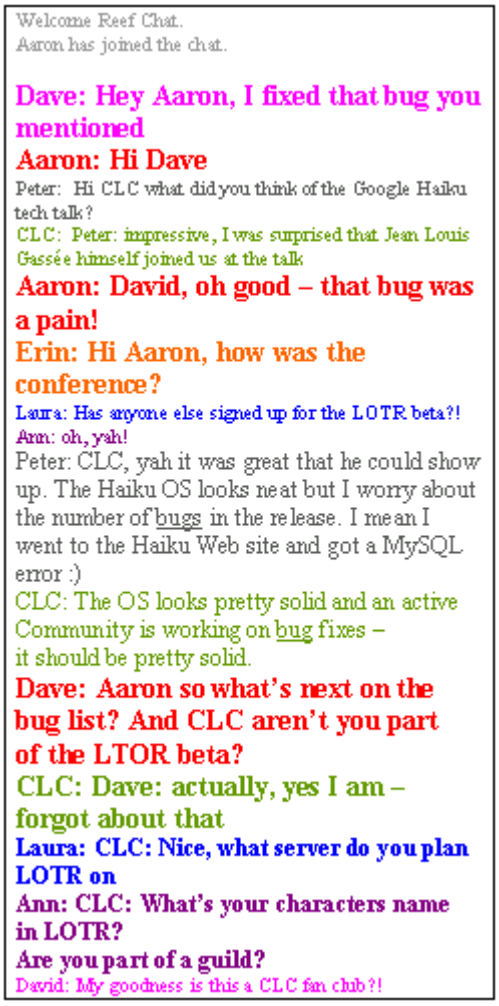
4. Case Example: Reef Chat

A trend in Web 2.0 Internet applications has been increased complexity. New features have been repeatedly added to Ajax applications causing Web 2.0 applications to behave increasingly more like applications in a desktop environment. [3] Reef Chat, developed by Peter Thiessen, was designed to be responsive like a desktop application and, in future releases, include drag and drop features and other user interface elements by leveraging the Dojo toolkit.

The main contribution of the chat application is in demonstrating that a Web 2.0 application can be accessible. This is accomplished by making chat updates accessible to AT. The chat also includes features such as text highlighting to aid sighted users in scanning text, contrast and font scaling options and so on. Reef Chat is also compliant with Section 508 [14], WCAG2 [18], and WAI-ARIA guidelines.

Reef Chat uses live regions and text highlighting to aid both visually impaired users and sighted users. The live regions are used to notify the AT of DOM updates when a new chat message is received. The text highlighting helps sighted users scan a chat log for relevant messages – the more relevant the message the greater the attention given to the message. This feature can optionally be disabled by the user. Below is an example implementation of the chat in action with nine people and a duration of 45 seconds.

Figure 1: Reef Chat, a 45 Second Chat



The chat widget functions by rendering new messages on the client-side based on relevance. The purpose of ranking messages is to assist users in scanning data and help point out meaningful information. The ranking is based on a three tier system. Table 7 describes the message-ranking relevance schema used in the widget.

Table 7. Chat Ranking Summary

Rank	Font Size & Weight	Criteria
Max	14pt, 100%	Client name in message, or a direct reply to client
Mid	11-13pt, 60-80%	Ranked depending on similarity to client's past messages
Min	10pt, 50%	Remaining messages

The table shows how the different ranking levels are used to markup the weight of each message. The visual formatting of each message is done using CSS and sectioned off into three ranks: MAX, MID, and MIN as shown in table 7. A message that is flagged as important (max), is given the largest font size, 14pt, and the strongest font weight. The MAX flag is used if a message has the client's user name in the message. Also, a direct reply is flagged as MAX, as are the subsequent three messages from that user. A medium ranked message is given a medium font size between 11-13pt. The MID flag ranking algorithm is fairly simple. The higher the count of similar words in a message compared to the client's messages, the higher the rank of that message. The remainder of the messages are flagged as low priority and given the smallest font size of 10pt and lowest weight. As a side note, font em percentages are actually used for font sizes and the font point sizes shown are the defaults. This allows scalable font sizes for users with low eyesight. Also, a message chime to help notify the user of a new message is used as an optional preference that can be enabled. To prevent a sudden decrease in ranking simply because a message does not contain enough similar words, the importance decays at a gradual rate. Therefore, a message with an 80% weight will not suddenly drop to 50%; instead, it will go down slowly as the conversation seems less and less relevant.

For DOM updates, the chat widget uses Ajax live regions to inform the AT. Several options exist for using live regions. The ideal solution would be to mimic the visual formatting of the chat widget by using multiple spoken voices in parallel with varying volumes. The ranking system of MAX, MID, and MIN would determine the volume for each message to be spoken. Unfortunately, technical barriers exist and the solution is not currently feasible. The multiple voices solution is investigated further in section five of the discussion.

The remaining options attempt to mimic the ranking system in Table 7. One option is to markup messages individually with ranked live settings. Another option is to group messages based on rank, all with the same live setting. A further option is to simply speak messages as they are received.

The first option, assigning a live region setting to each message, allows messages to be queued. A message could be prioritized on the client-side and given a rank and a corresponding live region setting. For example, a message assigned a MAX or MID priority could be given an assertive live region setting. A MIN ranked message could be given a polite live setting. Below is an example of what the chat in Table 7 would look like using the described markup.

Figure 2: Live Region Ranking Markup

```
<ul id="chatLog" role="role:log" aaa:live="off">
<li class="max" aaa:live="assertive">Dave: Hey A
<li class="max" aaa:live="assertive">Aaron: Hi D
<li class="min" aaa:live="polite">Peter:  Hi CLC
<li class="min" aaa:live="polite">CLC:  Peter: i
<li class="max" aaa:live="assertive">Aaron: Davi
<li class="max" aaa:live="assertive">Ann: Hi Aar
<li class="min" aaa:live="polite">Laura: Has any
<li class="min" aaa:live="polite">Erin: oh, yah!
<li class="mid" aaa:live="polite">Peter: CLC, ya
<li class="mid" aaa:live="polite">CLC: The OS lo
<li class="max" aaa:live="assertive">Aaron: CLC:
<li class="max" aaa:live="assertive">CLC: Aaron:
<li class="mid" aaa:live="polite">Larua: CLC: Ni
<li class="mid" aaa:live="polite">Erin: CLC: Wha
<li class="min" aaa:live="polite">David: My good
</ul>
```

Several design decisions were made when using this markup. First the live="rude" setting was avoided altogether for message queuing. A message marked with a rude setting would interrupt the current spoken message and could disorientate the user. Second, the role="role:log" [16] tag tells screen readers to treat

the elements in this block as separate entities. The consequence is that the *li* elements are spoken individually and could have varying live settings. The messages that were flagged as MAX and MID rank, would be given an assertive live region setting. As a result any polite message would be bumped off the queue and the assertive marked message placed on the queue. This allows a method for queuing messages that the AT can understand. The downside to this approach is that the polite messages could actually be discarded altogether and never spoken when competing against many assertive messages for a queue slot. This risk of "starvation" is probably a poor design choice, given that the message relevance and ranking algorithm cannot be perfect, nor is the goal of total accessibility being achieved if the experience of a disabled user is diminished. (Even if it were possible to ensure that the discarded messages were always irrelevant, disabled users would still miss out on the full community experience within the chat room.)

The second option, grouping and ranking messages, allows messages to be organized and queued. This option receives a group of messages and performs ranking operations on the messages before the DOM is updated. Only the assertive live region setting was used but messages were organized into two groups of messages, relevant and not relevant. Figure 3 shows an example of how the chat log from Figure 1 would be organized.

Figure 3: Chat Batching Option

Relevant
Dave: Hey Aaron, I fixed that bug you mentioned Aaron: Hi Dave Aaron: David, oh good – that bug was a pain! Erin: Hi Aaron, how was the … Dave: Aaron: so what's next on the bug list? And CLC aren't you part of the LOTR beta?
Non-Relevant
Peter: CLC, yah it was great that he could show up. The Haiku OS looks neat but I worry about the number of bugs in the release. I mean I went to the Haiku Web site and got a MySQL error :) CLC: The OS looks pretty solid and an active community is working on bug fixes – it should be pretty solid. CLC: Aaron: actually, yes I am – forgot about that Laura: CLC: Nice, what server do you plan LOTR on Erin: CLC: What's your character's name in LOTR? Are you part of a guild? Peter: Hi CLC what did you think of the Google Haiku tech talk? CLC: Peter: impressive, I was surprised that Jean Louis Gassée himself joined us at the talk Laura: Has anyone else signed up for the LOTR beta?! Ann: oh, yah! David: My goodness is this a CLC fan club?!

Messages flagged as relevant, located in the top cell of Figure 3, would be at the top of the queue to be spoken first. The second cell in Figure 3, contains the remaining non-relevant messages. The reasoning for this design decision was to allow the user to hear the most relevant messages first, followed by successively less relevant messages. This allows the user to decide whether or not the remaining messages in this group of messages are worth hearing. If not, the user can skip ahead to the next grouped batch of messages and repeat the process. The main benefit of this design is helping the user scan for the most relevant messages and have those messages read first. The downside to this design is that often the order of the messages carries some meaning in a chat log; this meaning would be lost if the order is changed. For example, the chat log example in Figure 1, begins with Aaron logging in and receiving a message from Dave. Soon after a series of fifteen messages were displayed on the screen in under 45 seconds. Fire Vox finishes speaking the first message and then has a queue of prioritized messages to speak. The thread about the LOTR beta originated with the question from Laura but instead the thread appears to have begun by Dave. The original intention of the thread, which was to see who signed up for the LOTR beta, was lost and may have been disorientating for Aaron. This issue is mentioned on the Mozilla Developer Center. [10] The many events occurring simultaneously on the chat, or any Web 2.0 application, can create a synchronization problem.

The third option, speaking messages as they are received, involves simply flagging each message with a polite live region setting and having it spoken by the screen reader. This solution was not very elegant, but worked with Fire Vox. When a new message is received, it is flagged with the ARIA live region polite setting and queued to be spoken after any remaining messages. The assertive setting was avoided because the sequence of spoken messages would be lost and potentially be disorientating for the user. If, for example, a user is reviewing a chat log, line by line, s/he would be interrupted at the end of each log message with any new messages in the chat. This would break the sequence of the chat log and potentially disorientate the user. The polite setting allows a user to review the log in peace, and without interruptions. This solution was chosen as the best design decision given the current technology.

Throughout the development of Reef Chat, WAI ARIA live regions were shown to be subjective in their use. A DOM update taking place in a live region could have any of the three live region settings assigned to it and remain consistent with the guidelines. However, as the different options or iterations of Reef Chat showed, the use of live regions can vastly affect user experience. If the wrong live region setting is used, an update can interrupt the user or flood the user with too much information. Best practices exist, such as using the rude setting sparingly and generally falling back to the polite setting whenever in doubt. However, the mastery of live region settings will probably involve trial-and-error for a Web developer to get it right.

5. DISCUSSION

The solution presented solves the immediate need of Web 2.0 Internet applications. Following the ARIA guidelines and using Ajax live regions enables a graceful method of informing an AT of a DOM update. However, as previously mentioned, active environments with a large number of page updates over a short duration of time can still pose a problem. Live regions were shown to be sufficient for several DOM updates at a time using different levels of politeness. However, what if the scale of DOM updates was increased to twenty or more at a time? If even ten of the DOM updates were labeled as assertive or rude, the AT would

be overflowed with information. A highly active chat environment is an excellent example. Suppose fifty participants entered a Reef Chat instance as described in Section 4.. The number of DOM updates would soon overwhelm a screen reader that supported live regions, such as Fire Vox, and the user would fall behind and be unable to participate in the chat effectively. The ARIA live region framework cannot support the high levels of activity in a highly active chat environment.

Taking a step back and looking at a screen reader, a rather large assumption has been made, that a user can and should only hear one message at a time. In a way, this model has decided that a human ear can only process one audio message at a time. This assumption is obviously false. People have the ability to filter and distinguish between multiple audio conversations or sounds at a time. For example, take a student in a noisy pub on a Thursday night. The environment is loud and full of sensory input, and the ear has the difficult job of filtering music, as well as near and far conversations. The student is attentively following and participating in a conversation with his/her current group of friends but is also able to jump in and out of nearby conversations. The student can hear key words such as their own name or a favorite topic in someone else's conversation, and begin attending to that conversation while still following their original conversation, and any number of other conversations nearby. This is known as the Cocktail Party Effect [2] and is an example of how humans can adeptly filter among many parallel auditory signals. By only allowing a visually impaired person to hear one message at a time from a screen reader, we are undercutting this natural ability. (Note that in a text-based chat domain, for sighted users this parallel processing can be achieved because vision is inherently parallel as well. And the relevance-based sizing described above is intended to facilitate the visual filtering process.) Hence, the serialization forced upon a screen reader user seems unnecessarily limiting and artificial.

Ideally, some form of parallelism across regions should be supported. Not only would this provide more natural engagement in chats, but it might also be easier to support within Ajax because less filtering would need to be built into the software (as more would be taken care of naturally by the user's ear itself) and thus could also be useful in other Ajax applications. A partial solution here would be to develop voice synthesizer technology that could speak multiple messages in parallel in multiple voices. Although most current synthesizers do no support this, even if separate instantiations are used, Charles Chen has developed a partial solution by using multiple synthesizers. After experimenting with his CLC-4-TTS core speech libraries for Firefox, Chen discovered that it is possible to have both the Microsoft SAPI 5 synthesizer and the Java FreeTTS synthesizer speak simultaneously. While this only provides two voices, it will allow us to test the utility of a system that allows multiple simultaneous voices. For a test program, we intend to use one voice at a lower volume than the other in a chat. The louder voice will be used for the main conversation, and the softer voice will be used for background conversations.

Since synthesizer technology here currently lags (it is unlikely that any synthesizer system could support 50+ simultaneous speakers with different voices), we currently need alternative ways to facilitate perceptual filtering. Reef Chat can be used to do basic ranking or filtering of messages based on importance. The markup described in Table 7 could be read by a screen reader and used to give a volume to a message, effectively using different volume levels as an imperfect, but perhaps passable, substitute for different voices for the ear to latch onto. A message marked with a MAX flag, would be given a 100% volume setting, a MID flagged message would be given a volume between 50-80%, and a MIN lagged message a volume of 50%. The screen reader could be speaking multiple messages at a time with the most relevant in the foreground at the highest volume, and the least relevant in the background at the lowest volume.

One way to describe this idea is that the accessible chat system is auditorily simulating the visual fisheye effect. [13] This effect works by visually highlighting a text element and bringing it to the foreground while pushing surrounding text elements to the background. The fisheye effect aids a user in scanning information by bringing attention to important elements. An audio representation of the fisheye would work similarly. In an active chat environment, at any given time, a message marked highly relevant would be spoken in the foreground, with a lower-ranked message concurrently spoken in the "immediate background", and the lowest ranked message concurrently spoken in the "background". Using this system a user could scan audio messages based on a three tier relevance ranking hierarchy. The granularity of filtering the human ear probably does not reflect a three tier filtering system. For this reason, the solution is not a perfect representation of the human ear's ability to filter. Future work is needed to study how many audio conversations the average user could follow without being overloaded with information. Also, the ranking algorithm used in Table 7 is fairly simple and more complex algorithms would be required to best support information processing. Several other technical barriers also remain as well, especially if this model is to be extended to support the wide diversity of Ajax applications that exist beyond chat.

6. RELATED WORK

The issue of Web 2.0 accessibility has become increasingly prominent. The WAI released an editor's draft [11] of a guideline to solve the problem of Web accessibility; the result was the use of roles and states. The document provided a framework for current best practices and instructions on embedding accessibility states and roles into an HTML document. Prior to this guideline, there was no standard way of providing markup to make a Web 2.0 Internet application accessible. Past work had been done on adding semantics to web content that was human readable and could be extended to widgets with dynamic behavior. [1]

Ajax toolkits and frameworks have also begun to mature, with many new powerful projects such as Dojo [4] and Prototype [12]. Toolkits allow the simplified development of rich user interfaces, often with complex visual elements that provide a level of abstraction away from the Web browser. In 2006 IBM contributed technology support to Dojo in the form of "… intellectual property to help establish a common, open industry framework and ecosystem around Ajax software development, IBM together with the Dojo Foundation and others hope to foster more innovation and adoption of Ajax." [9] The contribution has led to an accessibility library in the core Dojo libraries headed by Becky Gibson at IBM. The accessibility support still needs work, with elements such as the image generation for rounded corners not allowing transparency for contrast levels. Dojo however does have many accessible functions and an extensible library. The extensibility of the Dojo library has lead to its rising popularity and predictions [8] of it becoming the standard Ajax toolkit.

The release of the HearSay voice browser system in 2006, by Zan Sun, Amanda Stent, and I.V. Ramakrishnan, advanced screen readers by giving the user more control over how the content was read. [21] The voice browser worked via the combination of three elements: a browser interface object, a content analyzer, and interface manager. The browser interface object handled retrieving a Web page and contained features such as automatic form filling. The content analyzer broke down the content of a Web page into a partitioned tree structure. The interface manager handled classifying elements of the partitioned tree using pre-trained classifiers. The resulting system was a screen reader that could more effectively aid a visually impaired user to navigate a Web page.

7. CONCLUSION

Accessibility for Web 2.0 Internet applications is now possible through the WAI-ARIA [15] guidelines. Using live regions, it was shown to be possible to inform an AT, such as Fire Vox, of DOM updates, and in doing so make ARIA-compliant Web pages accessible. More active content poses a problem however, as Reef Chat demonstrated. An AT can easily become overwhelmed with DOM update notifications and the user fall behind when in an active environment, such as a chat. Future work was proposed in the discussion towards a potential screen reader system that could speak with multiple voices in parallel and aid the user in digesting large amounts of information efficiently, while providing a *naturally* accessible interface.

8. ACKNOWLEDGMENTS

Our thanks to the Mozilla Foundation for funding our projects and research. We also thank Stephen Hockema, Aaron Leventhal, and Gijs Kruitbosch for their helpful comments and suggestions and for proof-reading a draft of this paper.

9. REFERENCES

[1] Leventhal, Aaron. “Structure benefits all”. Proceedings of the 2006 international cross-disciplinary workshop on Web accessibility (W4A), 2006.

[2] Arons, Barry. (1992) A Review of the Cocktail Party Effect. MIT Media Lab. 11 January , 2007. <http://www.media.mit.edu/speech/papers/1992/arons_AVIOSJ92_cocktail_party_effect.pdf>

[3] Kouroupetroglou , Christos, Salampasis, Michail, Manitsaris , Athanasios. “Web accessibility (W4A): Building the mobile web: rediscovering accessibility?”. Proceedings of the 2006 international cross-disciplinary workshop on Web accessibility (W4A), 2006.

[4] Dojo. “Dojo the Javascript Toolkit”. 15 January 2006 <http://dojotoolkit.org>

[5] Chen, Charles. Fire Vox. 11 January 2007 <http://firevox.clcworld.net>

[6] Google. “Google Maps”. 10 October 2006. <http://maps.google.com>

[7] Google. “Google Docs”. 10 October 2006. <http://docs.google.com>

[8] Gehtland, Justin, Galbraith , Ben, Almaer, Dion. *Pragmatic Ajax: A Web 2.0 Primer*. The Pragmatic Programmer LLC, 2006.

[9] Market Wire. (2006). IBM Contributes Ajax Software Development Technology to Open Source Community. Market Wire 05 June 2006. 14 January 2007 <http://www.marketwire.com/mw/release_html_b1?release_id=133309>

[10] Mozilla Developer Center. “AJAX:WAI ARIA Live Regions”. 23 January 2007 <http://developer.mozilla.org/en/docs/AJAX:WAI_ARIA_Live_Regions>

[11] Pilgrim , M., Gibson , B., Leventhal , A.. “Embedding Accessibility Role and State Metadata in HTML Documents”. 20 January 2007 <http://www.w3.org/WAI/PF/adaptable/HTML4/embedding-20060318.html>

[12] Prototype. 13 December 2006 <http://www.prototypejs.org>

[13] Greenberg, Saul. A Fisheye Text Editor for Relaxed-WYSIWIS Groupware. Department of Computer Science, University of Calgary 1996. 15 January 2007 <http://sigchi.org/chi96/proceedings/shortpap/Greenberg3/sg2txt.htm >

[14] United States Government. Section 508: The Road to Accessibility. 11 January 2007 <http://www.section508.gov>

[15] World Wide Web Consortium (W3C). Roadmap for Accessible Rich Internet Applications (WAI-ARIA Roadmap) 20 December 2006. 17 January 2007 <http://www.w3.org/TR/aria-roadmap>

[16] World Wide Web Consortium (W3C). Roles for Accessible Rich Internet Applications (WAI-ARIA Roles) 26 December 2006. 17 January 2007 <http://www.w3.org/TR/2006/WD-aria-role-20060926>

[17] World Wide Web Consortium (W3C). States and Properties Module for Accessible Rich Internet 26 December 2006. 17 January 2007 <http://www.w3.org/TR/2006/WD-aria-state-20060926>

[18] World Wide Web Consortium (W3C). “Web Content Accessibility Guidelines 2.0”. http://www.w3.org/TR/WCAG20/complete.html

[19] Yahoo!. Yahoo Finance. 27 January 2007 <http://finance.yahoo.com>

[20] Yahoo!. “Flickr”. 27 January 2007 <http://www.flickr.com>

[21] Sun, Zan, Stent, Amanda, Ramakrishnan, I. V.. Dialog generation for voice browsing. Proceedings of the 2005 International Cross-Disciplinary Workshop on Web Accessibility (W4A) 10 May 2005.