Collaborative Web Page Change Detection

Sebastian Messier
Isha Bajekal
**Introduction**

To our surprise, with the growing responsiveness and dynamicity of the web, there are few solutions to track content changes on a page. We sought to address this by building a collaborative, non intrusive tool which would rely on users, rather than computers, to detect web page changes as close to real time as possible. We figured that most people would be interested in tracking sites that others will be visiting later in the day. Leveraging this thought into a solution was particularly interesting to us, so we sought to conceptualize an implementation which eventually became this project.

Existing technology is more rigid and less creative. If you search for a page-change detector, most sites offer cheap services which simply wget a page periodically, and compare it with some preceding or initial result. Some, going a step further, compare screenshots of specific areas of a website, which are taken periodically as well. All of these sites have a limit of once, if not twice a day checks. When one considers that many other individuals will visit this same page throughout the day, the idea of using their visit to another’s advantage gains traction. It was these conditions which finally convinced us to pursue this idea as our senior design project.

**Approach**

Our technical approach used quite a few existing technologies. Our chrome extension was implemented using the chrome API, a real-time database called Firebase, xpath, and node.js. Our approach and implementation used these tools in various steps. The two major components of the extension were tracking the web page changes and notifying the user.

Tracking web page changes required quite a detailed approach. First, we implemented a tracking tooltip using the chrome context menu API. This allowed us to offer users a way to select items to track. In our implementation, a user can track any type of element, including text, urls, and images. In order to do this, the user right clicks on an element and they see the option to “track element” next to the logo of our extension.

Once an element is clicked on to track via the chrome context menu, we move on to the next step of our implementation. In this step, we parse the element in such a way that it can be stored in firebase for later use. Building the parser was one of the most difficult components of our implementation. In order to parse and store an element in firebase, we first cleaned up the url of the current web page in order to remove extraneous parts of the url that were not necessary to uniquely identify the page.
Examples of things we removed were timestamps (since kept track of time ourselves) and randomly generated tokens. We then further identified the element selected by its xpath. XPath is a language used for traversing and defining specific parts of an XML document, so it was integral in uniquely identifying elements on a webpage, since different users could be tracking different elements on the same page. Once the parsing was completed, the url, element xpath, timestamp, and user email of the user tracking the element were all stored in firebase using the firebase web API. Tracked elements were stored in a table called “elements.”

The other key component in tracking web page changes is the comparison that determines whether a change has occurred. In order to determine this, we used parts of the parser that we had already implemented. Anytime a user of our chrome extension visited any page, whether or not they were interested in tracking an element on that page, our extension parsed the url of that page and checked whether it matched the url of any page stored in firebase. If it did, we then searched for the elements being tracked on that page using the xpath stored. If the element expected at that section of the xpath was not present in the web page or if the element had changed, then a change was marked as detected. In turn, this would update the firebase entry so that that element’s entry in the “elements” table would then be pushed over to a “notifications” table in firebase. If there was no change detected, then the entry would remain unchanged in the “elements” table.

The other major part of our chrome extension was notifying the user of changes in elements they were tracking. In order to implement this, we incorporated nodemailer into our extension. Nodemailer is an email sending module that uses node.js. In order to use nodemailer, we added a listener to our notification table. Therefore, whenever an element was pushed from the elements table to the notification table, the listener would listen for the change and then send out an email to all the emails listed as tracking that change and then immediately delete that entry in the notifications table. This completes the lifecycle of a tracked element.

There were various challenges that we faced in our implementation. Initially, just exploring and familiarizing ourselves with the chrome APIs was quite a difficult task as they are extremely extensive and it was challenging to weed through APIs that were relevant and useful to our extension and those that were not. The next challenge that we faced was in learning how to use xpath correctly for our application. Neither of us had experience with xpath previously, so there was quite a steep learning curve involved. Another major challenge we faced was with how dynamic the web is and the impact that had on the way we implemented parsing and comparison. Due to the fact that seemingly identical web pages often have completely distinct urls or web pages
with identical urls sometimes have slightly different layouts, we had trouble figuring out a meaningful way to store web pages and elements such that useful comparisons could be made. We tackled this issue by cleaning up the urls before storage in firebase, though there may be other equally valid solutions to this problem. Finally, the last challenge we faced was in figuring out a testing approach, since we did not have direct control over the web as a whole and therefore could not change element values to test our extension. We addressed this issue by implementing a sample web store and sample news website which we did have control over.

Results and Measurements

Benchmarking a collaborative, community based platform is no trivial task. We focused our benchmarking efforts on quantifying our promises: real-time notifications, and a smooth user experience. Respectively, we wanted our service to notify the user of a change within seconds of its detection, and we wanted the extension to be fast enough so that users wouldn’t experience substantial lag while browsing the web. Since these conditions are time-dependent, all of our benchmarks were in the time domain. The benchmarks were ran on a 15" MacBook Pro (early 2013).

Our real-time notifications benchmarks consisted of tracking all elements within tables of varying sizes (10, 50, 100, and 200 elements). Once finished tracking the entire table, we changed the table’s contents to trigger notifications. From there, we timed how long it would take the notifications to be sent to firebase, and how long it would take the email server to read the notifications and send them out to a mock user. The timing mechanism was a bit complex in order to account for network latency. We first send the notification with a timestamp of its creation. We then calculate the latency for the push. Then, when the email server sends the notification, we compute the difference between the initial timestamp, the current time, and the network latency calculated earlier. This calculation omits network dependent latency, and only considers actual duration of our computations. We ran this benchmark for 1000 trials.
The benchmarks on the overarching notification flow were very promising. The means were 18.83605ms, 18.33437ms, 23.77454ms, and 35.7767ms. The standard deviations were 3.545137ms, 3.276838ms, 65.13035ms, and 16.22191ms. The first two tests show similar results, with the 50 element one showing a few more outliers. It is not until the table size was raised to 100 that we see values in the seconds range. This was troubling, but the 200 element test showed no extremes greater than ~250ms, which makes the results from the 100 element test questionable. The higher standard deviation in the 100 element test further reflects that. Perhaps my machine was experiencing extraneous background work? Nonetheless, these results are promising. With the majority of the times within 100ms threshold, we add very little to the total time required to push out notifications. We can rest assured that network latencies will surely account for most of the actual time required, but that is simply beyond our control.

The last two benchmarks were performed solely in the extension. Specifically, we measured the speed of element tracking, and the change detection procedure. Again, we did not consider the latency of a firebase push in our measurements.

To benchmark element tracking, we used the tables from our previous benchmark and measured the time it took to track all rows. We ran this benchmark for 1000 trials.
The means of the data were 24.46569 ms, 71.28295 ms, 206.5442 ms, and 765.8087 ms. We find that it takes the extension less than 150ms to track up to 100 elements in parallel on average. It is not until we are tracking 200 elements that the average gets close to the 1s mark. These are exceptional results, given the fact that users are only permitted to track a single element at a time. If we allowed parallel tracking capabilities (users can track multiple elements simultaneously), we could comfortably allow users to track up to 100 elements simultaneously.

To benchmark the speed of change detection, we started a timer once the elements payload was received, and measured how long it took to compute a result and push it to firebase. We performed this in the cases where all elements changed for 1000 trials.
The means of the data were 27.35115 ms, 37.10616 ms, 27.46205 ms, and 30.05753 ms. The standard deviations were 7.645471 ms, 7.526851 ms, 14.83354 ms, and 14.09272 ms. We find that it takes the extension less than 50ms on average to detect 200 element changes. As we rack up the amount of elements being changed, we notice an increase in the standard deviation of the time elapsed. This could be due to the fact that there are other background tasks conceding for CPU time. Perhaps the larger the job, the more likely the scheduler will concede CPU time to other tasks. Regardless, these too are exception results - this part of our extension seems to behave in constant time.

**Privacy Considerations**

The project faced a considerable privacy concern early on. We originally wanted to notify the user of what an element had changed to. For example, if we track a price at $16.00, on a change, we wanted to notify the user that the price was now $16.01. We then realized that this feature would provide malicious users entry access to what could be sensitive information. If a user, for example, tracks elements on their amazon account profile page, assuming the profile url is the same for all users, another user could accidentally provide their account details unknowingly. This was an unintended side effect of a pretty neat and simple feature which we ended up omitting from the final product.

Another concern, which is easily addressable given more time, is a lack of user authentication. We allow users to input an email address to receive notifications at, but never verify ownership of the email account. We did not bother to address this since it is uninteresting and irrelevant to the core focus of the project. Before deploying this to the world, however, this would certainly need to be fixed.

**Closing Discussion**

Overall, our chrome extension addressed the initial problem we set out to tackle. Our implementation had many strengths that were important to us and reflected some of the priorities we had set in the beginning. The most important factor was that we wanted to make sure to have a working product and we did achieve this goal. Our extension, though brittle in some regards, accurately detected true web page changes. That is, at no point did it notify a user that a change had occurred when in fact no change had happened nor did it fail to notify users of changes that had occurred. In addition to this, one major strength of our extension was its speed. The delay between a change being detected and a user being notified via email was trivial. This is important in situations
such as buying items on an auction style web store like ebay, where time is extremely important. Another strength of our approach and implementation is that since the web page changes accessed via users, many of the traditional barriers that are implemented to prevent crawlers from accessing a web page are ineffective here, since a human user would be able to bypass these barriers. Speed, effectiveness, and a product that worked were all important to us and we were glad to be able to achieve these in our project.

However, we did still face some limitations in our implementation. The biggest change we would have made in our implementation had we had more time is that we would have generalized our solution more. Currently, our approach is quite brittle in the sense that two very similar web pages that do not have the same url, perhaps due to being accessed in different ways or as the result of different search terms, but do contain similar information would not give users updates on elements that they may be interested in since they are technically considered different pages. However, if a user is tracking something like the price of an item, that item may appear on a variety of different pages and any change in its price may be of interest to the user. Due to the dynamic nature of the web, we were not able to finish implementing a generalized solution to this problem, but we did take steps towards mitigating the problem by cleaning up the url to capture a wider set of web pages. However, in the future, it may be useful to consider not using urls at all and instead tracking elements and changes purely by comparing the Document Object Model (DOM) of two different pages. Though this is future work that exists, overall we were quite happy with our results.