Project B Report

Tsunami Alerts
CS 361 Summer 2016

Tsu-Shield

Kyle Bergman, bergmkyl@oregonstate.edu

Sammy Pettinichi, pettinis@oregonstate.edu

David Rigert, rigertd@oregonstate.edu

Alexandre Silva, silvaal@oregonstate.edu

Rachel Weissman-Hohler, weissmar@oregonstate.edu

XP Cycle 1

User Stories

No.	Story/Task	Units
1	Acquire tsunami warning information from government sources	3.5
1-a	Download alert data from NOAA National Tsunami Warning Center	0.5
1-b	Parse data and extract relevant information	1.5
1-c	Create a database table to store received alert information	1
1-d	Store relevant information in database	0.5
2	Keep track of user locations	2
2-a	Receive geolocation from app users at regular intervals	0.5
2-b	Create a database table to store user locations	1
2-c	Store latest geolocation in database	0.5
3	Show warning to users who are in an area affected by a tsunami alert	2.5
3-a	Create HTML proof-of-concept front end to verify server functionality	1
3-b	Identify users in locations affected by a tsunami alert	0.25
3-c	Send notification to affected users	1
3-d	Display received alert information on client app	0.25
3-e	Override silent mode and play loud alert noise with vibration	0.5
4	Display an evacuation route to the nearest tsunami safe zone	12
4-a	Create a table to store geolocations of tsunami safe zones	1
4-b	Collect and store tsunami safe zone data in table	10
4-c	Send coordinates to Google Maps for routing	1
	Total Units:	**20**

Priorities

As a team of 5, we decided we could handle 5 units per week. Our customer assigned priorities on a scale from 10 to 1, with 10 being the highest priority. This resulted in the following prioritized list of tasks.

Priority	No.	Task	Units
10	3-a	Create HTML proof-of-concept front end to verify server functionality	1
9	1-a	Download alert data from NOAA National Tsunami Warning Center	0.5
9	1-b	Parse data and extract relevant information	1.5
9	1-c	Create a database table to store received alert information	1
9	1-d	Store relevant information in database	0.5
8	2-a	Receive geolocation from app users at regular intervals	0.5
8	2-b	Create a database table to store user locations	1
8	2-c	Store latest geolocation in database	0.5
8	3-b	Identify users in locations affected by a tsunami alert	0.25
6	3-c	Send notification to affected users	1
6	3-d	Display received alert information on client app	0.25
6	3-e	Override silent mode and play loud alert noise with vibration	0.5
2	4-a	Create a table to store geolocations of tsunami safe zones	1
2	4-b	Collect and store tsunami safe zone data in table	10
2	4-c	Send coordinates to Google Maps for routing	1

Pair Organization: XP Cycle 1

The group convened on the Friday of the first week to discuss the goals, user stories, and how the members would go about tackling the Tsunami Alerts project from then on. Two teams were created, one that would focus on the back end and one that would focus on the front end. The back end team consisted of Rachel, Sammy, and David, while the front end team was made up of Kyle and Alex. These teams met individually throughout the next week to complete their assigned tasks.

Back End Team

The back-end team ended up having three people, which complicated arranging pairs. Ultimately, we decided to do trio programming instead, with one driver and two co-pilots. We rotated roles, usually between sessions but sometimes within the same session if one of the co-pilots had an idea of how to address a particular problem. Screen sharing and audio chat were used to facilitate the trio programming.

One of the biggest challenges of the trio programming was finding a common time to do the work. It would have been much easier if we were all in the same office and time zone, but with two of us on the west coast and one on the east coast, it was a challenge to find times that worked for everyone. We were able to reach a mutually acceptable compromise and started having regular programming sessions in the early evenings on weekdays and mornings on weekends.

In the first week, we were tasked with creating the server that would be used by the front end team to alert the user of whether they are or are not in danger. The data would be pulled from the National Oceanic and Atmospheric Administration in the form of latitude-longitude pairs that would indicate what areas are currently under a tsunami warning. In order to properly use the data, the XML data needed to be parsed and put into a form that would be easily accessible to the front end. Toward the end of the week, we decided to use a REST API for providing the data to the front-end. Multithreading was required because we needed to continue to pull alert information from NOAA while simultaneously serving it via an HTTP server. This introduced a new set of problems that we needed to work through. By the end of the first cycle, we had the server up and running, but we had to kill the process just to shut it down.

Front End Team

Kyle and Alex met during the first cycle two times to discuss and complete the foundation for the front end portion of the project. Alex created several prototypes to work off of prior to meeting so that it would be easier to create a web interface during the pair programming sessions. After slight discussion, it was decided that the front end would use buttons to show the difference in CSS that could be used to alert the user depending on the distance to the affected Tsunami areas. Much of the first meeting was used to get the basic structure of the website down and decide what functionality would take place when receiving information from the server. Problems were encountered when trying to alter certain HTML elements to make them centered or dynamically sized. These problems were resolved with some StackOverflow digging and the use of Twitter's Bootstrap web framework. During the first front end session, Alex was the driver while Kyle was the copilot and the roles reversed during the second session.

Coming Together

At the end of the first week, the group met again on Friday to review what each team had done, what problems were encountered, and what the team wanted to do next. The back end team had some difficulty with scheduling, so they agreed to meet the following day to finish working with the server before the front end team planned to meet again and prepare for making requests to their server.

Unit Tests

For the front-end, we created unit tests based on the notion that testing specific UI features (e.g. asserting that the hex code colors matched their specified parameters) would be less desirable and not nearly as efficient as testing the main functionality of the code. Given the near-prototype nature of the app; the fact that it was coded on HTML/CSS/JS-query -- we resolved to have an easy method to change divs via a button. This of course necessitated a unit test: did the button actually execute and accomplish what it was set out to do? The initial unit tests predictably failed, first because they did not have any code associated to them, and later due to the normal bugs found in the paired programming process. Eventually, all color-changing buttons and the submit button (to submit longitude and latitude) passed all relevant unit tests.

We also formulated a unit test to test our connection to the server we had set up to store all latitudes and longitudes. In asynchronous calls, the crucial return number was 200: which signified that we had successfully connected to the server and were returning data. With that, we asserted that 200 was the correctly returned number. As with the previous tests, this too failed; with continuous development, this test too passed. All unit tests were conducted using the QUnit application, which allows for testing of javascript modules.

For the back-end, we used the Python nose module to create our unit tests. For each Python file we created a corresponding test file which would test every function within the implementation file. For each test we provided various inputs (such as valid, invalid, and edge cases) and tested the outcome of each of those responses. The nose test module allowed us to run all test files at once and receive an output of total tests passed vs failed as well as any errors received throughout the testing. This allowed us to easily obtain an overview of how we were doing, while also pinpointing several specific issues in our code.

Initially we had many failures, as we wanted to ensure the tests ran properly early on in the production process. We kept running them as we completed sections in order to ensure our total successes were increasing and failures decreasing. This was largely true for us, though it was helpful to help us ensure that our code worked properly.

Through this testing we found there were some issue with time zones. Alerts are handled in UTC, so that was sometimes confusing compared to our local (and varying) time zones. We also had to check for an exception when removing locations in the case where the specified location did not exist. Mainly, the unit tests helped us identify typos and inconsistencies in our code where we had not kept consistent with other modules that we had already completed.

Acceptance Testing

For acceptance testing on the front end, we sat down and reviewed the app, from start to finish. To begin, did all elements successfully load? Is index.html the start page? Do the other files successfully load and display? For the most part, this phase of unit testing passed without much failure, though there did exist a few times when certain files would get temporarily misplaced between commits, or when a single change crashed the entire website. These were more typographical and clerical errors than anything else, and as such were easily remedied.

Beyond that, we did encounter larger problems with the UI and connection to the back end. Specifically, our methodology of centering elements was not perfect, and without properly sizing the screen, the app simply does not work as intended. However, we concluded that to make the app responsive and thus function on a mobile platform, this was not a bug, but rather a feature. To fully actualize the feature, one must click (on Chrome) F12, Ctrl+Shift+M.

While we could connect to the server after a handful of tries, we could not truly edit the CSS without further trial and error. And, during the process, we often found we would break the connection due to misplaced attributes. However, eventually we met with great success as we not only simplified our code to be fully modularized, but connected to the server and changed the CSS.

For acceptance testing on the back-end we actually ran the server and manually tested through variations of what we expected to work successfully. The first issue we ran into was that we could not run our server on flip. Sudo privileges were required to install the necessary python modules for the code to run. One member of the group has their own private server and volunteered it for use to run the back-end server for this project. This allowed us to quickly get it up and running.

Our next biggest issues related to multithreading. At first we attempted to run two separate processes, but that would never return the expected result. We finally realized that the issue was that separate processes do not share the same set of global variables. We then switched to running a separate thread within the same process instead of an entirely separate process. Another concurrency issue we found was that we were unable to properly shutdown the server. Sending a SIGINT via CTRL+C would have no effect. We had to set the thread as a daemon to allow it to receive the SIGINT and properly exit. Until we got that working we had to just force kill the process.

The remaining issues we identified through acceptance testing were specific to our project. First we had to determine what is a valid latitude vs valid longitude. At first we considered having them sent as a pair, but decided to split into separate values to more easily handle them. Lastly, our server stores alerts until they are expired, which means that even if they have been cancelled they will be stored there, but we do not currently want them returned to the front-end since that is simply announcing whether there is an alert in effect now. So we had to ensure that the server would only return active alerts, while ignoring any canceled alerts and removing any expired ones from memory.

XP Cycle 2

During the first cycle, we discovered a number of additional features that would be required to implement the requested functionality. This resulted in our story list growing significantly, even as we finished off a number of tasks from cycle 1.

First of all, we made good progress on story 1 and finished off tasks 1-a and 1-b. However, we realized that we forgot to account for cancellation notices and for when notices expire, so we added 1-e and 1-f. In addition, we had not decided which protocol or data format to use for communication between the clients and server, so we added tasks 2-d and 2-e. The proof-of-concept front end had some bugs that were discovered during acceptance testing, so 3-a1 was created to address them. As we worked through the implementation details, we discovered we had not yet fully considered exactly how the system would go about notifying users of a tsunami warning. We split task 3-c into five subtasks, 3-c1 to 3-c5, with three ways to notify users along with tasks to implement the necessary server-side infrastructure. We also found that in order to accurately detect whether a particular user was in an affected area, we would need detailed coordinates of the coastline and the elevations along the coast. For these, we created tasks 3-f and 3-g.

Finally, during some feasibility testing using Google Maps, we discovered that the Google Maps routing algorithm was insufficient for providing an evacuation route in areas without roadways. In order to meet the system requirement of guiding the user to a safe area, we need to collect reliable evacuation data and create our own weighted graph to use for routing. For these tasks, we replaced 4-c with 4-c1 through 4-c6. Many of these changes greatly increased the estimated units for completing the stories, especially the tasks that involve data collection. In the following updated table, new entries are marked with cyan.

User Stories

No.	Story/Task	Units
1	Acquire tsunami warning information from government sources	2
1-c	Create a database table to store received alert information	1
1-d	Store relevant information in database	0.5
1-e	Remove notifications when they expire	0.25
1-f	Correctly handle cancellation notifications	0.25
2	Keep track of user locations	4
2-a	Receive geolocation from app users at regular intervals	0.5
2-b	Create a database table to store user locations	1

2-c	Store latest geolocation in database	0.5
3	Show warning to users who are in an area affected by a tsunami alert	15
3-a 1	Refine HTML proof-of-concept front end	0.5
3-b	Identify users in locations affected by a tsunami alert	0.25
3-c1	Create REST API to interact with clients	1
3-c2	Design data exchange format and structure	1
3-c3	Respond with alert if user sends a location affected by an alert	0.25
3-c4	Send push notifications to the last known IP address of all active users	2
3-c5	Warn users who are near, but not in, a location affected by an alert	0.25
3-d	Display received alert information on client app	0.25
3-e	Override silent mode and play loud alert noise with vibration	0.5
3-f	Create a table to store geolocations of coastal areas and elevations	1
3-g	Collect and store coastal geolocation and elevation data	10
4	Display an evacuation route to the nearest tsunami safe zone	28.5
4-a	Create a table to store geolocations of tsunami safe zones	1
4-b	Collect and store tsunami safe zone data in table	10
4-c1	Create a table to store designated evacuation route information	1
4-c2	Collect and store designated evacuation route information in table	10
4-c3	Create a weighted graph to represent all feasible evacuation routes	3
4-c4	Calculate optimal evacuation route based on current location	2
4-c5	Display evacuation route overlay on Google Maps	1
4-c6	Update overlay based on user’s current geolocation coordinates	0.5
	Total Units:	**49.5**

Priorities

The customer was pleased with our progress and wanted us to continue with the previous prioritization, with an eye toward having the prototype front-end actually exchanging data with the server back-end. The highest priority story had far too many units to complete during the second cycle, so the customer decided to have us omit the database-related tasks for this cycle. We also had to simplify the logic for detecting whether a user is in an affected area by calculating whether they are within 10 miles of one of the locations specified by NOAA instead of having it be based on the actual coastline and elevation map.

Priority	No.	Task	Units
10	3-a1	Refine HTML proof-of-concept front end	0.5
10	3-c1	Create REST API to interact with clients	1
10	3-c2	Design data exchange format and structure	1
10	3-c3	Respond with alert if user sends a location affected by an alert	0.25
10	3-d	Display received alert information on client app	0.25
9	1-c	Create a database table to store received alert information	1
9	1-d	Store relevant information in database	0.5
9	1-e	Remove notifications when they expire	0.25
9	1-f	Correctly handle cancellation notifications	0.25
8	2-a	Receive geolocation from app users at regular intervals	0.5
8	2-b	Create a database table to store user locations	1
8	2-c	Store latest geolocation in database	0.5
8	3-b	Identify users in locations affected by a tsunami alert	0.25
6	3-c4	Send push notifications to last known IP address of all active users	2
6	3-c5	Warn users who are near, but not in, a location affected by an alert	0.25
6	3-e	Override silent mode and play loud alert noise with vibration	0.5
2	4-a	Create a table to store geolocations of tsunami safe zones	1
2	4-b	Collect and store tsunami safe zone data in table	10

2	4-c1	Create a table to store designated evacuation route information	1
2	4-c2	Collect and store designated evacuation route information in table	10
2	4-c3	Create a weighted graph to represent all feasible evacuation routes	3
2	4-c4	Calculate optimal evacuation route based on current location	2
2	4-c5	Display evacuation route overlay on Google Maps	1
2	4-c6	Update overlay based on user’s current geolocation coordinates	0.5
2	3-f	Create a table to store geolocations of coastal areas and elevations	1
2	3-g	Collect and store coastal geolocation and elevation data	10

Pair Organization: XP Cycle 2

For the second cycle in the project, the teams agreed to stay on the teams they were currently on to maintain consistency and because they were already familiar with what was happening on that part of the project. During the Friday meeting, the teams decided that the next steps were to have the backend team create JSON objects with a Boolean value that would determine whether the user should be alerted and a distance value describing how far away the danger is to the user’s location. The front team was tasked with preparing the webpage to take in the user’s location (in the form of a user submitted latitude and longitude), making the request to the server, and changing the page’s CSS based on the server’s response.

Back End Team

By the second week, the back-end team was able to meet regularly for pair programming sessions. We managed to fix the threading issues by researching further details of the multithreading and paste modules, and we identified and implemented several key pieces of missing functionality for correctly parsing alerts received from the NOAA website. One key missing feature was checking whether an alert received from NOAA was a cancellation alert. We added logic to the API used by the front end so that it no longer reported an active tsunami after a cancellation alert was received.

Front End Team

During the front end session, Kyle and Alex worked to get the user’s location based on user input. Originally, the idea was to use the location given to the system based on their geo location, but for the sake of time and simplicity, using a form submission would be easier. They were successful in getting the form to work and displaying it on the screen. The next meeting had Kyle as the driver and Alex as the copilot. At this point the backend team has set up the server to return JSON objects so the front end team could just set up the page to make requests upon form submission. Originally, the team had issues connecting to the server. After some brief debugging, they reached out to the backend to see if they knew what could be the problem. It turned out that the server had not been running. After the server was turned on, the frontend continued to have access issues due to the same source policy. The backend team was able to fix this problem by giving access to all domains. Now that these were resolved, the frontend was able to receive the JSON objects and change the CSS dynamically based on what was returned.

What Was Different

The consensus was that the second week was easier on the teams as a majority of the foundation and decisions were made during the first week's meetings. At this point, the teams had become familiar with each other's schedules and could more easily meet up to work. The only added difficulty came when the teams had to work together to make sure each of their components could work together seamlessly.

Reflection

As a group, there were things we liked about each of the programming processes explored this term. Waterfall focuses more on formalized communication and documentation, whereas XP involves more in-process communication (for example, during pair programming) and less written communication and documentation. This is a significant advantage for waterfall over XP, especially in a remote context. Since all of the members of our group were spread out geographically, we found the lack of formalized documentation a significant challenge in the XP process.

Another advantage of waterfall over XP in the remote context is the ability to work independently. Since all of the work can be divided into discrete tasks and there isn't a focus on pair-programming, individuals can complete portions of a project whenever their schedule allows. On the other hand, with XP and pair programming, we had to coordinate our schedules to get anything done, which in turn led to less time available to work on the project.

Finally, we found waterfall preferable in the implementation phase of development, since we could reference all of the planning documents and knew fairly precisely what code we needed to write. In XP, without the architecture and interface planning of waterfall, we struggled to know how to write our unit tests without laying out a plan for how the system would be structured first. This led to some initial floundering as we figured out which direction to take.

XP also had some advantages over waterfall. We were able to produce usable code much more quickly with XP than with waterfall, since the planning stages are so much shorter in XP than waterfall. We were also able to change things more easily during development in XP, since there wasn't any documentation that needed to be changed first; we just changed the code.

Another advantage of XP over waterfall is the focus on testing. Writing unit tests, using those unit tests, and completing acceptance testing each cycle led to a rapid discovery of any bugs or other errors and equally rapid solutions. Whereas, in waterfall, we might not know about any bugs or errors until much later in the process, and they'd be more difficult to correct.

Finally, while we found pair programming to be a disadvantage to XP due to the difficulty of scheduling time when multiple people are available to work, it is also an advantage of XP over waterfall. Having two pairs of eyes on each line of code led to faster solutions, both in terms of writing the initial code and finding errors in that code. Additionally, we found that when one person might not be sure of language syntax or how to approach a problem, the other person could often offer assistance that isn't readily available in a non-pair programming environment. While the short timeframe of our XP project didn't lend itself well to this, we could also see how pair programming could be an excellent opportunity for mentoring when one person is less experienced than the other.

Overall, our group preferred waterfall to XP. Specifically, we felt that waterfall is much better suited to remote teams such as exist in this program, while XP would be better suited to a traditional office environment where everyone is in the same space during the same hours and can work collaboratively much more easily. An additional reason that our group particularly preferred waterfall to XP is that our project was not very well-suited to iterative development. We had some difficulty figuring out a way to iteratively develop a mobile application with a server backend that would result in functional, independent pieces to evaluate each week. The waterfall process would have been a better method of development for our application, even though we couldn't have ended up with functional code in two weeks using the waterfall approach.