Agnes Developer Exercise

Robin Kurosawa

3/4/2017

Section I – Improvements

1. Android Capability

The first concern that stood out to me as a new user was that the application did not run on my mobile device. I also tested this on another android device with the same results. I’m not sure what exactly the bug is, but after the initial startup screen, the application crashes without an error message and reverts to its previous screen. My phone is a OnePlus 3T which is (more or less) a standard android phone. I do not know if the Agnes team is aware of this, but if not, this was a clear point of improvement that obviously hinders the potential outspread of the app.

1. Content Control Flow

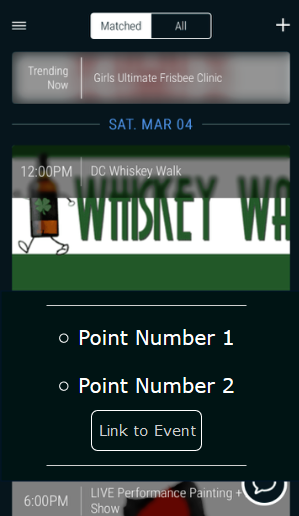
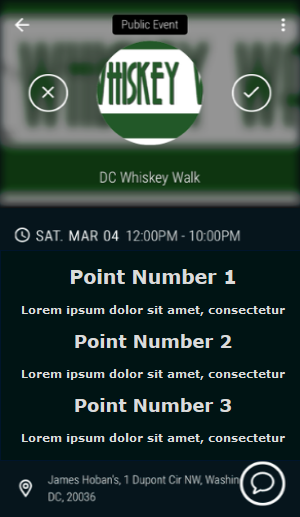
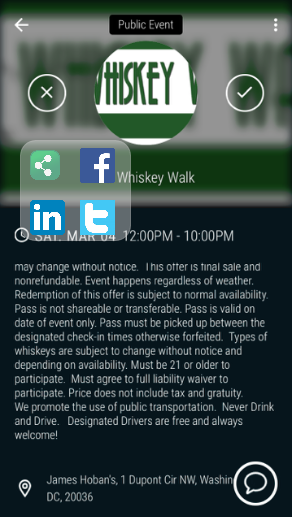
The descriptions for an arbitrary event were another point of potential improvement. I found that the block of text in the event page discouraged me as a user to understand what the event was about and where I could go to learn more due to an overflow of potentially poorly organized information. To improve, I propose the structure shown in Figure 1 in which the main points of the event are more emphasized and may allow the user understand the event with more clarity and smoother flow. This would be implemented through the event creation content management system by allowing new event publishers to organize their content in such a way that is readable by the user. Alternatively, the design choice shown in Figure 2 demonstrates a dropdown preview of an event. In this structure, when a user taps on an event in the events page, a dropdown list appears consisting of the main points about the event and a button leading them to the event page.

Figure 2

Figure 1

1. Social Media Integration / Embedded Calendar

I believe a third potential improvement to the Agnes app is the integration of social media platforms with events, groups, and users. Agnes already does this initially by allowing login through a Facebook account, but this idea could be pursued further due to the nature of the application. I believe the ability to post events to friends’ Facebook/ Twitter/ Linkedin pages would significantly increase event visibility and Agnes usability. One way this might be implemented would be to include a share icon in each event page opening a bubble containing several different social media platforms. This is shown in Figure 3. In addition, a calendar function in which a user can view events in the coming month/week/day seems like a potentially powerful component of an event publisher subscriber system. With such a component, I believe end users could more effectively choose and schedule events that don’t conflict and allow them to more easily view their event subscriptions. I believe a good way to design this would be to add a calendar component in the sidebar, the data for which could be strictly Agnes data or imported from a user’s google calendar as well.

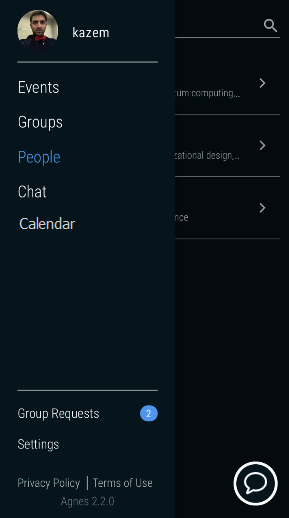


Figure 4

Figure 3

Section II – Backend Infrastructure Schema

1. Overview

To begin, I started with the understanding that the Agnes mobile application runs on a distributed publisher/subscriber system specifically with a topic-based subscription (filter) model (also referred to as a distributed event-based system). The core problem of any distributed pub/sub system is first to ensure that events are delivered efficiently to all subscribers that have filters defined that match the event (Coulouris *et al.* 2012). In addition, depending on the level of funding and function of the system, addition requirements could be: (but not restricted to) security, scalability, failure handling, concurrency, and quality of service. In each of these requirements, tradeoffs with performance, stability (availability), and cost must be taken into account. I found that the Google Cloud Pub-Sub concept overview covered the base ideas pretty succinctly.

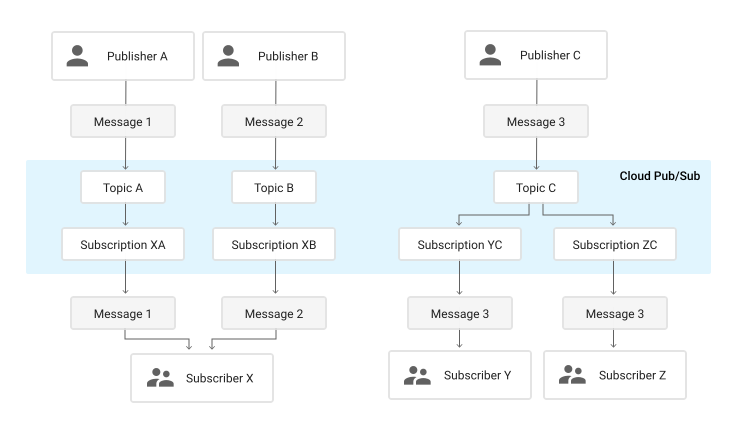


Figure 5

In Figure 5, the high level concept is illustrated. In our case, I believe the desired traits are security, scalability, availability and concurrency. Going forward, I am going to operate under the assumption that the speed of messages (i.e. performance) is not a critical aspect of the infrastructure because if Alice (Publisher) publishes an update to an event to which Bob is subscribed, a slightly higher latency will not decrease the functionality of Agnes. This increased threshold for acceptable latency will allow for a stronger security and more availability as a tradeoff. That being said, a lower latency is always preferred. Of course, if cost was not an issue, we could simply throw a ton of hardware at the problem to make it fast and stable (i.e. implement a distributed system similar to FaRM [exploit DRAM and RDMA for low latency transactions]), but we are going to assume that this isn’t the case. Finally, the network bandwidth of the system must be a factor to take into account, but without knowing this, I will assume this is not a bottleneck.

1. Schema

My design is illustrated in Figure 6. My model follows the general structure of Google’s Cloud Pub/Sub system, with the caveat that I didn’t need to worry about multiple datacenters (but this is something that can be taken into account). It’s immediately noticeable that stable storage is this giant cloud, and in practice this would be something like a distributed fault tolerant file system (i.e. replicated state machines running some sort of consensus algorithm like RAFT/PAXOS).

Key Terms:

Forwarder: Server responsible for moving updates between publishers and subscribers

Load Balancer: Server responsible for assigning clients to forwarders.

Publisher: Creates updates and publishes them to the specified topic

Subscriber: Receives messages on a specified subscription



The design itself relies on the fact that “not every subscribing forwarder needs to be aware of or receive [updates] from every publish [update] source for a topic.” In other words, subscribers can connect to several different subscribing forwarders to receive updates about a certain topic. This allows for high horizontal scalability in addition to an inherent level of fault tolerance due to the fact that, if a subscribing forwarder fails, a subscriber can still receive an update on a topic from a separate subscribing forwarder. It is also important to note that when a publisher sends an update to the publishing forwarders, it immediately writes it to persistent state (Stable Storage) and does not acknowledge to the publisher until the write has completed (usually meaning a majority consensus on a data object). Also there exists a load balancer that assigns publishers to publishing forwarders (i.e. ensuring that a publishing forwarder does not get overloaded), this can be implemented with a simple hash, or a smarter algorithm requiring storing state on the load balancer.

1. Implementation

Since we are using node.js as a framework, the API allows for multithreading via the cluster module where a master process creates child processes via fork() which can all share server ports. This in turn creates an environment in which we can have several child processes (threads) running on a single server at once. Since we are not interested in running serializable transactions, there is no inherent requirement of atomicity or synchronization using locks. Thus, we can increase the throughput of the system by requiring publishing and subscribing forwarders to be running several instances of cluster concurrently. In addition, the Node.js framework by nature allows for asynchronous communication between servers, which allows for unacknowledged updates to be sent and not be required to block while waiting for a response. The culmination of these factors allows for a system with high scalability, availability and concurrency while also avoiding busy waiting and therefore lowering operational latency. Additional frameworks can be incorporated to increase modularity and simplicity including: [Express.js (process management), Total.js (overall functionality), and Meteor (cross device coordination)]

1. Testing

I found that Google’s testing methodology made a lot of sense to me in that they incorporate three separate environments to allow the ability to test software updates before they are used by active customers: Testing, Staging, and Production. Testing and Staging do not have any user traffic, and are used to identify issues before release. This obviously is restricted by the amount of funding allowed, but even having two different environments makes a lot of sense to me. In terms of monitoring, a set of Service Level Indicators (SLI’s) , metrics that describe the overall state of the system, are constantly outputted (i.e. how long it takes to service a publish request). By taking measurements over a given period, the averages balance out and certain Service Level Objectives (SLO’s) arise that can be used to benchmark SLI’s. To perform this monitoring, a set of fake clients are used to publish or subscribe to the network and measure various aspects of the system. In addition, forwarders and an additional load balancer can be used for similar reasons.

Section III – Node.js Application

1. Security Methodology
   1. DDoS:

A DDoS attack really manifests itself as trying to overload either the load balancer or particular publish forwarders. In the case of the load balancer, this can be mitigated by either storing state on requests from all IP’s and denying socket connections repeatedly requesting within a certain time threshold, or paying a lot of money to Cloudflare to do it for you. The tradeoff here is pretty obvious. Unfortunately mitigating DDoS attacks can be very difficult (i.e. the hashing function used to distribute load is leaked and a publish forwarder gets overloaded due to an adversary repeatedly causing hash collisions), so outsourcing it is probably the safer option.

* 1. Brute Force

By Brute Force, I’m assuming this means a brute force attack on the authentication system used in Agnes, in which a simple counter can be used to disallow multiple consecutive requests on the same userId or IP.

* 1. SSL Attack

This type of attack can come in many different forms, perhaps one of the most prominent is BEAST (Browser Exploit Against SSL/TLS), in which a third party can silently decrypt communications between a browser and a server by attacking a weakness in a deprecated use of the AES\_CBC mode encryption scheme. This type of attack is mitigated by using TLS v1.2 which uses explicit IVs. Another type of SSL attack might be the SSL renegotiation attack, in which the attacker blocks an initial TLS handshake request by the client and causes a renegotiation to occur with the client. This consumes a disproportionate amount of server-side resources, making it a potential vector for DoS attacks. This is mitigated by limiting the renegotiation number of requests per time period and specifying the time window for which the renegotiation can happen.

Github Repository: <https://github.com/Rkuro/Node-JS-Application>

References

1. Coulouris, George F. *Distributed systems: concepts and design*. Harlow: Addison-Wesley/Pearson Education, 2012. Print.
2. <https://cloud.google.com/pubsub/architecture>
3. <http://www.evilprofessor.co.uk/634-protecting-node-js-from-beast-tls-attack/>
4. <https://nodejs.org/api/tls.html#tls_client_initiated_renegotiation_attack_mitigation>
5. <https://raft.github.io/raft.pdf>
6. <https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-dragojevic.pdf>
7. <https://www.sitepoint.com/how-to-create-a-node-js-cluster-for-speeding-up-your-apps/>
8. <https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-sharma.pdf>
9. <http://www.cs.ucsb.edu/~agrawal/fall2009/PNUTS.pdf>
10. <https://nodejs.org/dist/latest-v7.x/docs/api/>
11. <https://www.cloudflare.com/>