Carleton University

Honours Project Final Report

Distributed Review System

Li, Charlie (100832579)

Anil Somayaji

December 16, 2016

Contents

1 Abstract

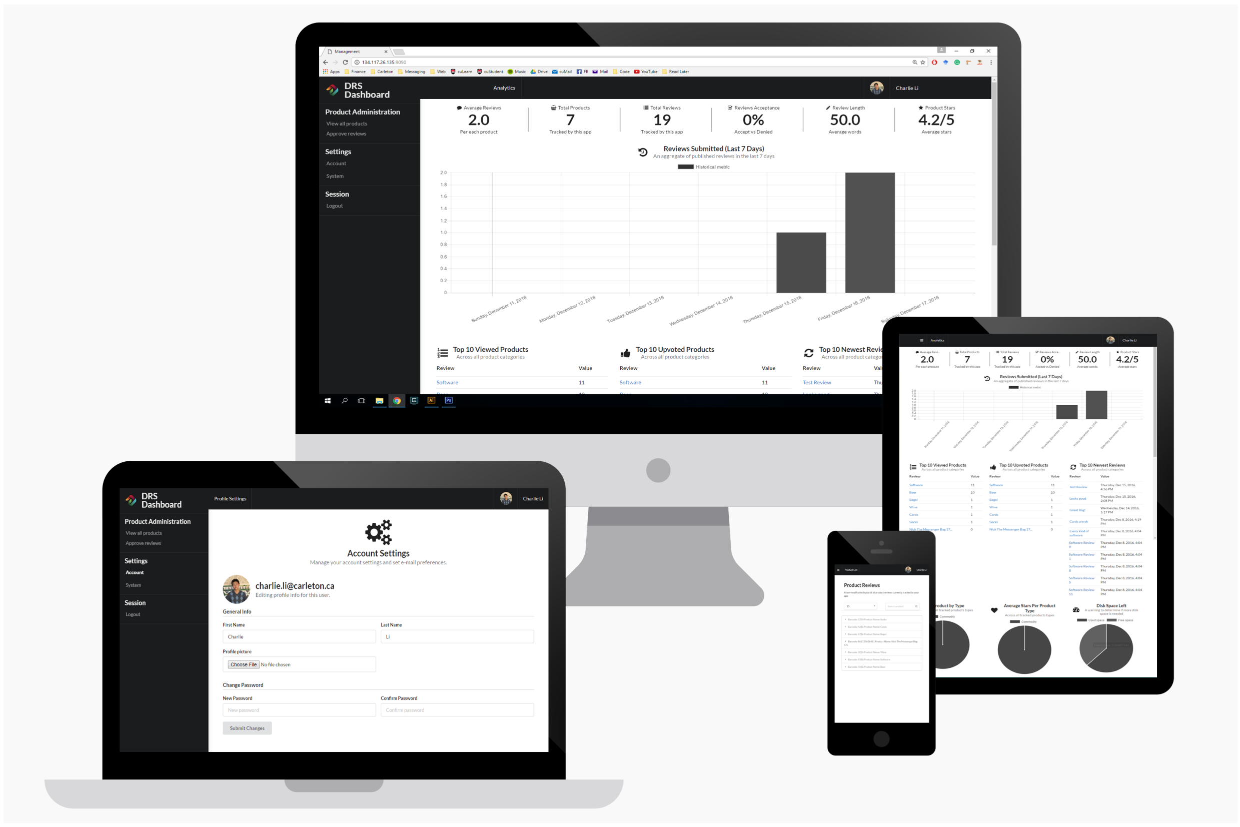


Figure 1: Demo of DRS

In today's time, data has become a precious commodity. Large organizations viciously collect user-submitted product reviews to gain a competitive edge in the marketplace and squeeze out smaller players. Good product reviews provide e-commerce sites with a boosted sense of credibility and can help drive sales up. This project takes an open-sourced, peer-to-peer distributed approach collecting and distributing commercial product reviews, making the data free to use and alleviating the ownership of the data from any single entity. The architecture of the Distributed Review System is powered by a Kademlia protocol distributed hash table, layered under a easy-to-use REST API and management dashboard capable of delivering review data to any interested parties. The approach serves several benefits over centralized architecture systems in that it inherits a self-organizing topology that's scalable and reliable in nature. The solution entails a single executable that acts as client and server for creating, fetching, and approving reviews. Available at github/zcharli/distributed-review-sys.

2 Introduction

One of the driving factors of consumer trust on e-commerce websites is the user generated reviews that provide validation on the products sold. Market researchers have agreed that a listed product with good reviews can potentially increase conversion rates by as much as 5% [1]. Not all e-commerce sites are able to attain the same level of user participation such as the traffic generated by Amazon or TripAdvisor, thus this project is targeted to help the smaller players in the vast market of online shopping to gain credibility from the masses. Large organizations have always made a big deal about sharing their data since it's a factor that gains them the competitive advantage in their respective markets. Product reviews are a form of protected asset at companies such as Amazon or PowerReviews used to leverage the data that faithful buyers write and gain consumer confidence for the institution. Both these companies leverage their vast collections of reviews to ensure competitiveness.

Large organizations generally side with centralized review systems locked away in their data centers which present a single point of failure. The motivation behind this project is to take an alternative open sourced approach to sharing what is rightfully the people's voice about their online shopping experiences. This project attempts to decentralize the ownership of product reviews for common everyday items and can be extended to apply reviews of services, accommodations, and more. The project is expected to empower and serve the needs of smaller e-commerce vendors by allowing them to collectively bank a large catalog of reviews in DRS.

At the core of DRS, a distributed hash table (DHT) is used to structure the entire network. Using locality sensitive keys, queries can be routed across DRS within $O(logn)$ hops. This form of structure allows the system to become self-organizing, perform with enhanced reliability, and respond to scalable requirements without much human intervention. The DHT protocol Kademlia, by Petar Maymounkov David Mazières, was selected to power DRS for their novel approach for key space partitioning, using the XOR distance metrics, which allows efficiencies to be drawn from general queries that benefit from locality sensitive data. Unlike other protocols such as Pastry, Chord, or Gnutella, Kademlia is a well-accepted and time-tested protocol used in many of the popular systems that support our Internet today, for example BitTorrent.

This report will examine the design decisions and development process that I undertook over the Fall semester of 2016 in developing this project. Certain features will be thoroughly described in the report. An integration of a REST API is included in the primary architecture of the system to allow interoperability between different interfacing systems. DRS also allows consumers who don't necessarily want to participate in hosting a DHT to also retrieve reviews through HTML iframe delivery. For those who are kind to contribute hosting, DRS provides a special administrative interface to manage new reviews, apply searches across the entire review domain, and offer an intuitive tracking and analytics tool to the host. Technologies used in composing DRS were selected with great care to manufacture a single platform independent, minimum sized executable with easy setup requirements.

3 Background

This section will generate a discussion on the topic of Peer-to-Peer (P2P) architecture, how it's evolved over time from to this present day, and populate implementation of modern day DHTs.

3.1 Primary Architecture Motive

The continual growth of the Internet, infrastructure, and content included, is accompanied by a wealth of diversified application and requirements. The traditional client-server paradigm, which became popular since the early 1980s, could no longer support the continual evolution of the Internet alone. Over past 16 years, the peer-to-peer (P2P) paradigm has been catching a lot of steam with user and data-centric applications that are virtually cost-free. Originally designed to support file sharing, P2P has become a legally controversial means of resource distribution amongst users in the age of Web 2.0. The approach taken by P2P systems has decoupled the traditional client-server paradigm by decentralizing the ownership of data across participants (peers) while meetings some key aspects of the evolving and improving the Internet.

Scalability is one of the fundamental challenges and prerequisites for the new age Internet. Internet traffic will be growing at a compound annual growth rate of 22 percent between the years 2015 to 2020, an estimate by Cisco Visual Networking Index, an ongoing initiative to track and forecast global IP traffic and growth [2]. Growth in traffic subsequently generates derived demand for more bandwidth, more storage capacity, and more processing power to sustain the hunger for information. As large scale Internet systems anticipate this increase in traffic, it must be designed with scalability in mind. Bottlenecks must be identified prior to scaling up and solutions must be able to handle several orders of the magnitude without loss in efficiency. The traditional centralized model lacks in this area as those systems tend to become prone to bottlenecks while managing large scale resources.

Security and reliability have become ever more important as the web becomes more available to far-reaching countries. As information becomes more abundant, governments or businesses feel the need to control or capitalize on your data. In this age of big data, what do we consumer get out of it? Data is power and should power be concentrated or spread out [7]? Still a philosophical question with no answer, but we know for sure that the traditional paradigm of centralized information control becomes vulnerable by design. It provides a single point of failure for either a hacking or denial of service attack.

Flexibility and Quality of Service in today's network infrastructures raise the need for modular application architectures that can support the growth and expansion of the Internet. New emerging services should be easy to integrate into the existing network and removed at ease. Centralized architecture lacks the agility and can prove to be difficult and expensive to modify as they had to work around their architectures drawbacks to manage scalability. These three key requirements are what decentralized system tackle on an architectural level and attempt to simplify the solution to the challenges. DRS is meant to distribute data storage across participating nodes, effectively leveraging shared bandwidth and memory stores across many independent hosts. Each peer in the network can contribute their own processing power which alleviates stress from a single-entry point on the network comparative to centralized architectures. Peers in DRS interact directly assuming the responsibility as both client and server, a new concept called *servents*. This communication technique allows consumers of the system to act in cooperation and avoid bottlenecks when fetching resources.

There is no way to take the entire system down unless a distributed denial of service attack could find all peers in the network, hardly an easy task. Without the single point of failure, DRS can withstand single hackings without too much information compromise.

Information is not addressed by the location, the address of the server, but by the data itself through a hash value which helps mask the identities of peers in contrast to the location-based routing of the Internet. Peers would connect to DRS in an ad-hoc manner. Each peer is symmetrical in functionality and is responsible for only a partition of the entire data set. No one peer in the network owns all the data, creating a sense of fairness amongst the members of the network. If a peer wants to disconnect, it can take its' data offline preserving what's theirs. DRS allows resources to be located without any central entity which allows a flexible network topology for any number of peers in the network.

With the architecture of DRS defined and contrasted between the traditional centralized solution, the following section will provide an overview of the specifics of P2P history, adoption, and key concepts.

3.2 A Short History on P2P Sharing

The approach taken in this section is written in a time traveler's format, where the history of the technology is presenting in chronological order up until today. Through this journey in time, different methods of decentralized data sharing techniques will be discussed and related toward the project described in this report.

3.2.1 The Beginning of P2P File Sharing

In the context of this report, files refer to chunks of data that are shared among different hosts and can manifest under the specific definitions such as messages or raw data. A file is not limited to the conventional definition of a singular resource container, a one-dimensional array of bytes, but can also be a subset of bytes that return from a persistent data storage utility. The concept of P2P sharing originated from Usenet, a worldwide distributed hybrid of email and discussion system, like a web forum, conceived by Tom Truscott and Jim Ellis in 1979. Since the time was before the internet age, the protocol for file sharing was done with the Unix-to-Unix Copy (UUCP) protocol that's connected over the publicly switched telephone network, also known as dial-up [3]. Truscott and Ellis pioneered decentralized file sharing as an alternative to the popular Bulletin Board System, a centralized data exchange platform that performs network I/O for reading news, exchanging messages, and email managed by an authoritative administrator. Usenet was adopted by many universities in its early stages which allowed it to receive and re-distribute files among other Usenet servers, effectively duplicating files upon those who are connected. The significance in Usenet was the fact it was the first decentralized Internet community that hosted some of the major developments in the pre-commercial Internet era. Many high-profile projects had been announced on the network such as the World Wide Web by Tim Berners-Lee, the Linux project by Linus Torvalds, Mosaic by Marc Andreessen, and among others. In the same year as Andreessen's announcement (1993), Eugene Roshal released the RAR format, a way for large files to be compressed and split into multi-part archives. The release helped drive the popularity of decentralized file sharing protocols since files can now be uploaded more efficiently and effectively. The RAR format allowed large documents to be split up, avoiding a full re-upload if the entire file if it became corrupted during transmission.

There was a turning point in 1999, through all the advancements to date, P2P finally hit the masses. The release of Napster brought P2P music file sharing to the public, overlaid with a user-friend interface. The company was a short-lived success, shutting down 3 years after its release due to a major flaw in their design. Over the following years, P2P sparked elevated interest in academics to search for ways to build a decentralized network without the complications of Napster. Early versions of Chord, BitTorrent, Kademlia, among many were conceptualized using the method of distributed hash tables. These early versions had different implementations for their protocol but are similar in the same way that they all approached the architecture with self-organization, fully decentralized, and content-addressable storage (shifting away from location addressing).

3.2.2 Moving Towards Decentralization

The first-generation P2P systems such as Napster relied on a central database to coordinate lookups on the network. Napster would return the address of the peer who had the requested resource and only after obtaining the peer address, would then the originator commence a data transfer from the located peer. Although sharing was conducted between downloader and uploader, the central server, where lookups and indexes were managed by Napster provided a single point of failure and was sued for piracy in the court of law. This flaw ultimately resulted in the end of Napster by a lawsuit from the Recording Industry Association of American (RIAA).

During the legal battles of Napster, a small open source team, by the name of Nullsoft, invented Gnutella as an open-sourced project. The advances Gnutella made to P2P field involved the removal of the central server, thus making it impossible for RIAA point the finger and track whose sharing with who. Other commercial P2P distributors such as KaZaA approached the RIAA's lawsuit threat by encrypting peer traffic, thus rendering it impossible for RIAA to trace illegal activities back to the distribution firm. The major enhancement introduced into *servents* by Nullsoft involved delegating the responsibility of routing to peers among the network [4]. A new Gnutella client must know at least one other node in the network, called the bootstrap peer, and connecting to it will begin the journey of mapping out the network. Gnutella followed a simple, but bulletproof idea which allowed queries to be flooded to all connected peers. This form of distributed query can be propagated up to seven levels deep and hit thousands of machines on the network but has no guarantee that the queries can return successfully.

The design and network structure was presented with two challenges, one of which was solved using the notion of UltraPeer. During the early Internet era, peers were often plagued with slow connection speeds caused by dial-up or bad broadband modems. Gnutella solved this problem by presenting the notion of UltraPeer, a proven and dynamically elected peer based on its suitability for P2P transfers. Some factors that attributed to this election were sufficient uptime, sufficient RAM and CPU speed, and suitability of the operating system that can handle many concurrent sockets at once from OS types such as Linux, Window 2000/NT/XP, and Mac OS/X. The role of UltraPeers profoundly contributed to the success of Gnutella, allowing it to scale since slow peers would never be elected to this role. UltraPeers acted as a proxy to slow peers by keeping slow connections alive through open sockets and by utilizing its own bandwidth to increase the speed of transfer to other UltraPeers and ultimately made the experience of sending tolerable queries everywhere. In addition, an analogy can be made by relating them to major airport terminals across the world such that UltraPeers manage most of the major routing between far away peers across the Gnutella network.

The second challenge of the network structure was the flood-based query routing protocol Gnutella follows which can cost a tremendous amount of bandwidth consumption on the network. A query can take linear time across the diameter of the network, by that time the query would have hit every single peer. Although this issue was not fully addressed by Gnutella on a design level, later P2P protocols faced this problem with a more structured setup for routing queries across the network.

3.2.3 Improvements in Network Structure

Prior to the discovery of distributed hash tables, file sharing involved passing direct addresses, typically the IP, of the peer with the requested file to the requester to establish a direct communication channel for transfer. A query would involve an entry point, the requester where the initial query is generated, to pass a request to directly connected neighbors in the search for the file under their host. If the file was not present, the request would be propagated to the neighbor's neighbor, ultimately into the network until a reply is generated by either the address of the file being held or reach an expiry on the time-to-live (TTL). This type of search technique lacks a runtime bound and can consume unnecessary resources per query. Researchers have considered applying distributed hash tables (DHT) as the backbones of P2P networks to leverage their ability to perform distributed indexing (content-addressable data storage) and build performance bounds for efficient routing. Typically, lookups are in the order of logarithmic. DHTs have introduced the general map interface put(key, value), get(key), remove(key) to implicate a self-organizing protocol on files in the network. The DHT protocols vary between implementations but all possess a few similar characteristics on key space partitioning and routing duties.

3.2.3.1 Key Space Partitioning

Peers in this structure will operate a small subset of the key space from the hash table it represents. Each node typically will hold references to $O(log(n))$ other nodes, such as the Chord implementation. Data is cataloged by keys, usually between 128 to 160 bits, which forms the foundation of DHTs. Each implementation of DHTs varies in the way it approaches key space partitioning, a method of indexing/hashing the file based on the content it represents, and the approach of allocating the range of keys each node is responsible for. A common method to determine the partitioning is called consistent hashing and is used in most ring structured topologies. Each node is assigned a random identifier as key, and stores a set of file hashes that are close to the node ID. Closeness is defined by a distance function which varies by implementation, such as the XOR function Kademlia uses. This simple distance metric can help locate the file's address space by taking the distance relative to each node's ID. Most DHT implementations attempts to spread the keys evenly across all hosts so the network access is load balanced. However, bottlenecks can still be experienced if a bulk of the files end up hashing to one node's address space, a node manages a very large section of the file hash address space, or if the node address space holds very popular files. Load balancing is needed to address these sorts of issues to ensure even dispersion of lookups amongst all nodes of the network with rehashing or rerouting strategies [5].

3.2.3.2 Routing

To find a file, the binary or name is hashed into a file key before the query is forwarded into the overlay network, a virtual network that connects all the address partitions to form a full view of the system, typically through direct UDP or TCP connection. The address partitioning ensures that only a small group of related nodes will receive the query. The query is expected to move the closer to the result with each hop until the destination is reached in a deterministic fashion. The concept of what closeness is still relys on the underlying routing algorithm. For example, Kademlia uses the XOR distance function to compare the numeric difference between two node IDs and file hash. This routing solution solves the common scalability issue from Gnutella. By eradicating the need for UltraPeers, no nodes will play a vital or distinct role within the operation of the distributed system, thus, bottlenecks in the system are avoided from an architectural level. Key partitioning avoids the need for queries to be forwarded past the desired location/node ID which avoids any unnecessary traffic in the overlay network. If the key is not present in the partition where it's expected in, then it must not be anywhere in the network. In contrast to Gnutella, which will flood the network until the request times out. On the contrary, the drawback to using address partition keys is due to the immutability of the identifiers which makes it impossible to perform fuzzy searching in the cases where a similar file is suitable for return. Lastly, the distributed indexing mechanism embedded in DHTs are very robust against failures since the absence of a key, possibly due to disconnect, will simply return nothing leaving the rest of the network unaffected. In summary, the improvements made over unstructured P2P systems helped bound the expected runtime of a lookup while alleviating the network from any unneeded message passing. DHTs provide an efficient abstraction layer to handle routing of requests and data across multiple nodes while addressing the scalability problem plagued by unstructured networks.

3.3 Kademlia in DRS and Related Implementations

As discussed previously, the technology that supports the decentralized nature of DRS is the DHT backbone. A special implementation called Kademlia was picked for its performance characteristics and latency-minimizing routing. Kademlia is considered a later generation DHT that was designed as an improvement upon two of four original DHTs Chord and Pastry. This section will discuss each implementation lightly and followed by a description of Kademlia and how it overcame the weaknesses of the former.

3.3.1 Related DHT Implementations

Chord was designed by Stoica, Morris, Karger, Kasshoek, and Balakrishnan at MIT computer science labs in early 2001, often praised for its simplicity in its design. Elements added into the DHT were hashed into keys between zero to 2^b–1, where b is often selected based on the upper bound to the number of elements the DHT will hold. This key partitioning scheme defined Chord and theoretically formed a ring topology where the element keys were increasingly ordered in a clockwise rotation. The topology allowed queries on node keys to wrap around the ring with circular modulus. Routing is handled by a finger table stored at each node which each entry of the table mapping a proceeding (clockwise) keys of the ring in logarithmic scale with a maximum of b entries. A query would begin from a source node and follow the finger tables on each hop in hopes of finding a range of keys which the query key is located between. The protocol implies a logarithmic jumping distance each time around the ring. The lookup algorithm is reminiscent of binary search in ring topologies. A problem with this approach is that the comparison on the query key and the finger table entries do not lead to any useful information as to the distance or destination locality. The queries are forced to travel one direction, clockwise, and it's not known whether hopping to an entry in the finger table is getting closer or farther from the destination. Consider the case where the destination node is rank i on the ring and source node is rank $i + 1$. Source and destination are very close, but the finger table will try to route the lookup around the ring to the other side. Adhering to the ring structure, traffic orientation can result in unnecessary hops when the destination is within the sources locality. The bottleneck in this implementation is dependent on the latency of the next network link to follow, which must be checked for timeouts, leading to possibly slow query response times. Kademlia is designed to handle both the locality routing problem as well as compensating for bad latency and will be discussed after a brief overview of Pastry.

Pastry was designed by Anthony Rowstron (Microsoft research) and Peter Druschel (Rice University) and features the similar decentralization and self-organization properties as Chord and other DHT implementations [6]. Pastry's is designed with routing efficiency in mind by applying a method of binary key partitioning to be able to deduce key closeness as well as network locality. The resulting key space belongs to the zero to 2^b–1 set where b is typically a large number like 128. Every node joining the network would randomly select a number from this set. Routing works by finding a node ID that is the closest to the desired query key, closest is defined by the number of matching prefix bits in the node ID and the search key. Unlike Chord, Pastry can route to any node in the network. Instead of using log base two in the finger table, Pastry uses log base 2^b entries in its routing table, effectively minimizing the number of hops needed to reach the destination of the query. This tradeoff in Pastry's design implies a larger cluster on data located in the locality of a node ID called the leaf set. The leaf set contains a cluster of node IDs which are close in the identifier space that's used by the routing table to route short hops near the vicinity of the current node ID. Separate from the leaf set, a neighborhood set is designed to handle failures, arrivals, and recoveries for nodes that are close in terms of network locality, identified by a scalar proximity metric like the number of IP routing hops or geographic location of nodes. Pastry applies a heuristic to minimize the cost of network hop as well as minimizes the number of overlay network hops to successfully route a query. On the first hop, Pastry begins the lookup by looking for a node that has the largest common prefix in its routing table, effectively killing off half the possibilities. Then when it reaches a node within the locality of the search key, Pastry switches routing algorithms to look for numerically close keys instead of distance to ensure routing loops do not occur (caused by having the same number of common prefixes). Unfortunately, the second step for Pastry may reach a closer node ID numerically, but can be very far in distance geographically from the first step leading to difficulties in worst-case analysis. Typically, this flaw implicates that messages can be routed over increasing distances (generally logarithmic) as they get closer to destination

ID.

3.3.1.1 Kademlia

In 2002, Maymounkov and Mazieres had the chance to the identify and redesign a new DHT protocol that patched over the flaws identified by earlier DHT protocols, Chord and Pastry [7]. The new protocol, Kademlia, provides better efficiency than Chord as it can relieve its structure from a ring topology. The new approach also corrected Pastry's ID prefix routing metric with ID comparisons that can derive the closeness of two keys through a novel XOR metric approach. They also simplified the routing algorithm removing the need to switch to a second numeric closeness routing algorithm. In the end, Kademlia can closely bound its runtime, without the difficulties in Pastry's algorithm switch routing, resulting in $O(logn)$ operations for all DHT interface methods. Kademlia is also able to overcome the slow connection issues that may plague search paths in both Chord and Pastry by utilizing parallel lookups basing itself on the locality results from its XOR metric. It is clear why most DHT deployments nowadays are Kademlia protocol based [8], powering LimeWire, eMule (in early days), and BitTorrent's tracker-less torrents.

The brilliance and academic significance in Kademlia is its' XOR metric, which measures the numeric closeness of two identifiers. Whether that be a node ID or a search key, all key entities have a significant utility in the functioning scope of the protocol. Key partitioning uses a 160-bit identifiers and can be visualized as a binary tree, where all the leaves are unique identifiers. The implications on the network structure has some nice properties that ultimately leads to a grouping of elements that are similar with respect to low XOR differences. Nodes can maintain knowledge of the set of neighbors that have an address space closest to their own identifier and exponentially decreasing knowledge of more distant addresses. If we had a distance function d, that returns the exclusion OR of two identifiers, then we can derive $d(x,x) = 0$ and $d(x,y) > 0 if x! = y$. The metric is symmetric, $\forall x, y : d(x,y) = d(y,x)$ solves the problem in Chord (asymmetric). In a Chord ring, node 1 is one distance unit away from node 2, but node 2 is the entire distance of the circle minus one distance away from node 1.

When routing a query, Kademlia favours old links over links to newer nodes based on the observation that long uptime is correlated to the higher probability of stability. Each node is designed to keep a constant k number of entries in their routing table from known links. On a query, a node can send out a constant number of queries a, where a $a \leq k$, of queries out in parallel. This method allows the link with the lowest latency to reply first and the slower links ignored, increasing the stability and performance of the query. Kademlia will ping all k nodes each hour to refresh its routing table. If any queries fail, the node will gain an early signal to make an adjustment towards its routing table. Since properties such as parallel lookups and configurable routing table size can be adjusted, Kademlia makes it easily prove upper bounds while maintaining performance behaviours using its' trade-off in bandwidth consumption and lookup latency.

3.3.1.2 Kademlia's Fit in DRS

As the previous section noted, Kademlia is one of the best DHT implementations out there and DRS has leveraged this protocol to optimize the lookup paths for read operations. DRS traffic is global in scale and queries can come from any corners of the world. It was important to avoid the unpredictable behaviour of Pastry's 2nd stage routing process and the unnecessary traffic from ring routing that Chord entails. For successful deployment of Chord and Pastry, a closed off, controlled environment such as data center is needed. An example is Amazon's Dynamo and Cassandra databse is a production deployed Chord DHT solution. In uncontrolled environments, efficiency is crucial for a system to become useful. Kademlia's XOR metric forms a tree like structure that clusters IDs that are similar, close to each other. The XOR metric also provides sparse connections between very different IDs, linking far away clusters. This form of mapping which resembles a small world phenomenon [9], where there always exist a few edges in the network that make long edges to other parts of the world. The locality of file storage has a direct benefit to the way lookups take place in the network. The entire DHT key of DRS is composed of a concatenation of the following hashes in order: barcode, status, content, and version. Barcode represents the product barcode that the review if for. Status represents whether the review is in the acceptance or publish phase. Content represents the hash of the review, where uniqueness is identified as the actual content of the review such that you may not have the exact same written review for the same product. Lastly, the version keeps track of the number of changes and edits that were made to this review, where reverting back to a previous version is possible. Kademlia inherits the prefix routing protocol from Pastry's first routing step that groups the 40 prefix bits by the barcode together.

Looking up a number of reviews for product Y is anticipated to be the most popular read query across DRS. Therefore, it would be most strategic to group all reviews for the same product together. The XOR metric becomes a highly attractive routing protocol in this regard as it allows all products with the same prefix, barcode, status to be stored close to each other leading to optimal lookup queries. Only the last 80 bits in the file identifier will be unique to each singular review which are prefixed by 80 identical bits representing the product and status. The last 40 bits also has significance in attempting to keep each version of the file within the same locality to avoid network hops to search for the latest version.

In summary, Kademlia was a very good choice as a network routing backbone for DRS as it will allow optimized locality lookup queries to be fired. By this point of the report, the core technology has been thoroughly discussed. It's time to move to upper layers of the technology stack. The preceding sections will provide discuss DRS's approach versus the competition.

3.4 Comparison to Other Review Systems

DRS is a classification of its own in the ecosystem of public product review systems. There is currently no open sourced solution to access product reviews, let alone operate with a distributed approach. Big players in the commercial domain including Yotpo, PowerReviews, and TrustPilot reap the benefits of centralized solutions as it's proven to be a monetizable business case. These providers charge a subscription or flat fee for usage of their services to tap into millions of reviews collected over user submissions. Yotpo, for example, can control the complete product technology stack and deploy numerous scalability measures to be able to handle 5 billion requests a month. With an infrastructure composed of smart caching layers and geographically mirrored databases, Yotpo queries tend to avoid the bottlenecks created from disk reads as much as possible resulting in improved delivery times. The quality of service (QoS) is guaranteed by the popular content delivery service, Akamai, through the Dynamic CDN solutions, that scales resources when needed. The entire Yotpo system is hosted on AWS cloud capable of handling failovers and maintain a high degree of uptime. To no surprise, the service is worth up to 399 per month in subscription fees.

DRS's QoS cannot be compared to the level of precautions taken by Yotpo and their centralized review service. Instead, the advantages are drawn through the masses. With distributed processing power and self-organizing structure, DRS is impermeable to some forms of DDOS attacks and is capable of handling failovers by rerouting traffic around troubled nodes. In many ways, DRS bears subtle resemblance with Yotpo from an architectural standpoint.

4 Design

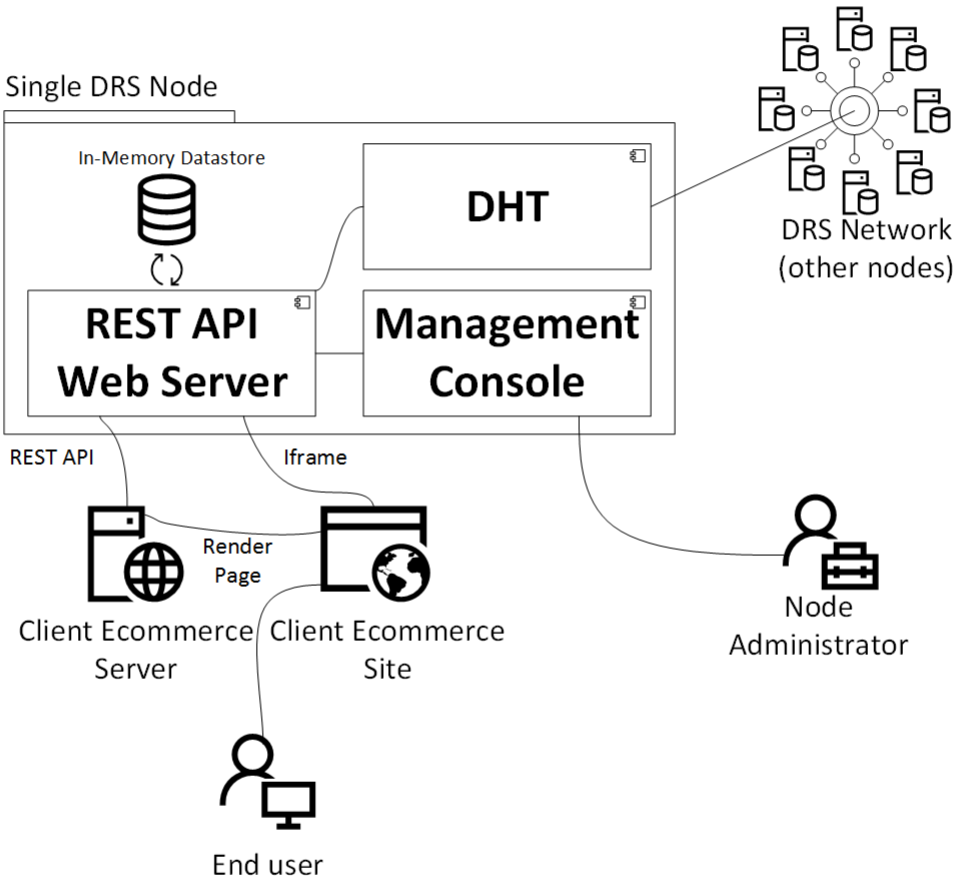


Figure 2: The Component diagram of DRS and its architecture

In this section, an overview of the system design is given to map out each component of DRS. The decentralized, self-organizing structure leads to a single executable for each deployment of DRS, referred to as a node. A node is essentially a self-contained adapter to the DRS network that responds to search and insert queries. An overview of the design is complemented with a component diagram by figure 2, indicating the activities each node is involved in.

When a node starts up, it will open two ports (4000) to connect to a designated node within the DRS network called 'the bootstrap'. The purpose of the bootstrap node is to provide some metadata about the current state of the network, a list of neighbors, and some network configurations to start the life of the new node. The new node will generate a random number which becomes the key space partition that the node is responsible for. The executable will open another port (default 9090) for web traffic and REST API calls. The overall system can serve two different types of queries a server side query or client-side query.

A server side query requires the client to be running an e-commerce application server and interface with DRS directly through the REST API. Typical queries a client would fire are as such: ask for a subset of reviews for a certain product, post a new product review, or upvote and downvote a review. Product pages must follow a look and feel standard thus, it's better if each client renders their own pages. DRS REST API plays the role of a back-end system for the application servers to first query then render the page with reviews. In this method, it's suggested that reviews are lazy loaded onto the page to avoid DRS becoming a bottleneck when striving to meet response metrics. It's also suggested to build a caching component into the application server if service level agreements (SLA) exist for performance characteristics. The queries that trigger DHT lookups can travel at most $O(logn)$number of hops where n is the total number of participants in DRS. During high network congestion, this can be slow, thus a caching layer can avoid high volume requests since queries are managed by the DHT subsystem which masks the network operations. The REST API treats the DHT like an ordinary map interface and each call to the interface can be costly. The interface abstracts the network communication calls through UDP, which can prove to be unreliable and can fail silently without warning. In these network error cases, caches can help lower the risk of failed UDP messages.

A client-side query does not require a web server to render the review results. To request for data, a client-side query can hit any DRS node and provide a flag to receive HTML encapsulated iframe back instead of JSON. The benefit of this approach is that the requesting web server does not have to expose the overhead to host a node, which takes disk space and require additional CPU cycles time. To use, simply include a remotely fetch a script within the page where the reviews are needed. As the page loads, the script will make a call to a public DRS server to fetch a pre-formatted list of reviews in the form of a standalone HTML document. Not everybody wants to contribute resources and pilot a node in the DRS DHT and that's ok. Client side delivery allows outsiders to query for review data without the complications of running a REST API to manually aggregate reviews before rendering the page. If the end user chooses the way of client-side delivery, then the functionality is limited. Any read operations are allowed, but not writes, therefore there is no possibility of writing new reviews for products this approach. The regular use case for an e-commerce site will just pull reviews for a single product before rendering it to the page.

DRS has made it very easy to pull reviews onto the page without any server-side processing. The strategy is to import a small script onto the page. On load, the script will automatically listen for the DOM loaded event and proceed to make an asynchronous call to a public DRS node for reviews. There is a small gap of time to wait before the reviews are all returned, but it's acceptable because the shoppers are probably looking at the product descriptions and images during the first 10 seconds after page load before scrolling to find the reviews. On the public DRS server, an HTML doc is compiled from a DustJs template then returned to the client. When this HTML doc arrives on the client website, it will be encapsulated into an iframe underneath the script tag. Since all HTML is encapsulated inside an iframe, it is protected from CSS classes outside leaking into the review list. I tried to include custom JavaScript to use inside the iframe for some cool effects, but it turns out, iframes cannot evaluate encapsulated JavaScript. However, it can fetch custom stylesheet and apply it to the scope of the internal iframe. The downside of this approach is the look and feel of the page might not match the iframe. Therefore, the UI design has been kept as simple as possible while providing as minimum functionality as possible. The client-side approach grants read-only permissions to the data to avoid destructive, untraceable damage to the data stored in DRS.

Lastly, the management console subsystem is a separate web application hosted by DRS for administering newly submitted reviews. One of the major benefits of becoming a part of the DRS network is the offering of the management dashboard. The main purpose of the backend is to allow reviews to undergo an approval process before it is formally published into the network. The DRS administrator on each node is responsible to collaboratively verify submitted reviews for the products they track, improving the quality of reviews that become published on the DRS network. Typically, an administrator would filter out spam, unrelated reviews, attempts to inject harmful HTML tags, profanity, and bad quality reviews. The definition of bad quality review is subjective to the administrator, but simple spelling mistakes can be corrected by the admin before being published. The secondary purpose of the dashboard is to look up and search reviews. Although the search module is quite primitive, there are plans to enhance its features to produce more useful results in the future. A tertiary purpose is to provide the administrator with some analytical tracking on how the node is currently performing. The analytic module is also primitive and the tracking features are simple as of now, however, the modular architecture of the application can support improvements in later iterations to this subsystem.

At this point of the report, I hope you've gotten a bird's eye view of DRS and its design. Next, up, we will discuss the gears and cogs that each component is built upon as well as the implementation setup used during the development of DRS.

5 Implementation

With much of the background, design, and core network structure defined, this section of the report will describe the software layer implanted in each node, the design decisions made during each stage of the development, and the challenges that I overcame at each step. This section is broken up into several subsections, each touching base with an important aspect of DRS.

5.1 Development Environment and Tools

The nature of network-driven applications fundamentally requires a different kind of development environment. For starters, the operating system must exemplify an unrestricted policy on network traffic. To simulate a small network, at least three nodes are needed, hence, draws a clear hardware requirement. The process for code, build, and the test should be fast and painless. With these requirements in mind, the perfect operating system to develop DRS on is UNIX. UNIX has very unrestrictive network policy that come right out of the box so the newest version of Ubuntu 16.04 server edition was selected. There's no way 3 instances of DRS could have been manageable on one host so a cloud solution was important. After a quick back and forth with Andrew Pullin, the system admin for the Carleton OpenStack, a small virtual space was created to make it possible to run 4 hosts simultaneously. After installing one Ubuntu image successfully, a snapshot was taken to be replicated amongst the remaining hosts.

To rapidly and efficiently develop DRS over the course of two-three months, a proper set of development tools must be carefully selected. Java language was chosen to fit the bill over a few main benefits: type safety, network programming-centric, and platform independency. Java has one of the best developer toolkits available that often works exactly the way you want to out of the box. One of the most popular Java IDEs, IntelliJ, was picked for the development of this project since it's well integrated with revision control, Git, has seamless secure file transfer protocol (SFTP) built in. Using the IDE has significantly improved tracking down bugs through remote debugging with socket protocol. These benefits tremendously improved the development time of DRS. When a file is changed on a development machine, SFTP would automatically transfer the file to the server. Although UNIX systems are quite liberal with in and out traffic, OpenStack was not. This presented a hurdle when configuring remote debugging. In the beginning, I missed the fact that all OpenStack VMs were restricted to SSH port 22, HTTPS port 443, and ICMP ports for ping commands. After running Wireshark to debug the network fault, the log showed a pattern of consistently retrying requests with no reply with can come from a misconfigured firewall or a closed port. After a discussion with Andrew, we decided to open two ports above 2000 (so super level privileges are not necessary to attach to the port) for debugging the front end, the DHT, and backend.

As the project gains complexity with the addition of mixed web files, numerous libraries, and differentiated modules, a correct build procedure is essential to output resources compilation into a Java Archive (JAR) for distribution. Previously, I was familiar with build procedures using Broccoli and Grunt, both build systems run with Node which helped automate repetitive tasks such as compilation and execution. On the Java side, there is Gradle and Maven, both very good, but I choose to go with Gradle as it provides richer functionality out of the box. Writing a build procedure that enabled the program to automatically open sockets for remote debug, invoke the Ember (frontend) unit tests, building the Ember app for production, and fetching project dependencies such as the DHT library, had made the development experience simple and painless. On the downside, Gradle was a totally new experience for me and the learning curve was steep. Out of all the Java build systems, Gradle is the most complex, but learning it had paid heavy dividends over the course of the development of DRS. Gradle allowed me to write a bash script to trigger the compilation command then synchronize the deployable JAR across all 4 hosts to begin testing. The most helpful aspect of using Gradle was from the build and invoke scripts. Since there are 4 hosts that serve as my test environment, I had programmed execution commands specific to each host to include metadata like the OpenStack floating IP configuration, whether the node must run or connect to the bootstrap, or certain client routines that were set up for testing. This reduced the amount of information I had to remember during testing as well as reducing the amount of commands required to launch the system.

The two extra ports Andrew opened for me were utilized to their fullest during development. Port 8080 is opened to Java Debug Wire Protocol (JWDP), a protocol used to communicate between the debugger running on IntelliJ IDE and the JVM hosting the executing Java code on the server. Port 9090 is shared between the DRS backend and the Node.js hosting my Ember.js application in development mode. Lastly, the HTTPS port was opened for web traffic which is happened to be used for client and server side rendering tests. I despised the idea of binding a Node.js e-commerce application to HTTPS without a cert so I came across with a technology called Nginx, a high-performance web server capable of acting as a reverse proxy. This means Nginx can be configured to listen on an open port like 443, then proxy all requests between localhost and the outside world. This was interesting to me since Nginx can recognize URI components to redirect traffic towards any standalone app, essentially acting as a router to your system. Using Nginx makes for an interesting microservices architecture where multiple service processes can run on localhost while Nginx redirects traffic between the micro processes rather than through a monolithic application.

5.2 Persistence Storage

On the talk of storage, picking a database means trading off between the CAP theorem: consistency, availability, and partitioning tolerance. Known as the Brewer's theorem, Eric Brewer raised a conjecture that proves it's impossible for a distributed system to have all three properties [10].

> "It is impossible in the asynchronous network model to implement a read or write data object that guarantees availability and atomic consistency in all fair executions (including those in which messages are lost)."

The theorem states a contradiction which is proven by utilizing the fact that the asynchronous network model has no way of determining whether a message is lost or delayed when it's sent over the wire. Thus, concluding that in the case of high network congestion, a read message that depended on a congested write, can prove the lack of consistency versus availability in the database.

Consistency refers to guaranteeing every read receives the most recent write or error message. Databases designed to around the consistency pillar strive for ACID (atomicity, consistency, isolation, durability) transaction properties. Proper adherence to the ACID properties ensure each transaction is atomic, all or nothing operation, in every situation such as power failures, errors, and crashes. The consistency in the ACID properties refers the compliance of new data against the database rules, schemas, and constraints, thus ensuring the database to be always in a valid state. Isolation refers to the ability for the database to handle concurrency and maintains valid system states during concurrent transactions as if the transactions were serial. Lastly, the durability property ensures data that's been properly commit will be permanently stored in the database in the event of power failure, system crashes, or other errors. Many of today's relational databases are designed with ACID transaction in mind such as MySQL and Oracle.

Availability in the CAP theorem refers to the reachability of the database. Every transaction must end with a response to the originator, even during failure conditions. Brewer defined availability in terms of the duration of an algorithm execution, unbounded runtime however, must always terminate at some point and return a response.

Partition Tolerance highly depends on the tolerability in the availability and atomicity of the database implementation. Some databases perform an operation called sharding, which is to split a large database into smaller, more easily manageable parts either in on the same host or across a network among many hosts. Sharding allows indexes to be spread out which can gather performance improvements from searching in a smaller subset. Partition tolerance is the ability for the architecture to handle failures during sharding for example. Consistency requirement implies each transaction to be atomic, however, a network link cannot guarantee the arrival of the messages. Availability implies that the server must generate a response, but messages can be lost during transaction again. Therefore, amidst network failures or process errors, a response must be actualized nonetheless which is the key to a partition tolerant system.

In the context of DRS, my persistence storage would support each node individually, acting like a containerized file system to power Kademlia. The purpose of the persistent storage is to protect the data in case of system failures. When a node goes down, the in-memory store that the DHT heavily relies on for fast lookups would be wiped from the network. Restarting the node would require repopulating the catalog of product reviews into the network, starting with the reviews that were hosted by the current node. Therefore, each DHT node would persist its own set of key partitions that it's responsible for.

In evaluating the persistent requirements of DRS, I came up with a few key properties the system would needs to function as best as it could. The database system should be always available to supporting one process, achieve acceptable consistency since missing a review is not life threatening, employ fast reads, low memory and processing footprint, and be generally performant in respect to the application's read and write operations. Form these requirements a few databases were considered.

First, it was the juggle between relational databases or non-relational memory/document stores. It was a rather easy decision to be made since the system did not need to complex relational modeling where SQL could excel at querying. After considering the data, the relationship between products and reviews can be modeled in one dimension, instead of drawing a join relationship between them. By storing only sets of reviews, we can reduce the coupling between product information and its review and give each endpoint of DRS its own ability to associate this binding. While the requirement to support fast reads still stand, there becomes a need to de-normalization a relational database to make this optimization, thus raises the question of why it's even needed in the first place. Most relational databases are designed to support a high degree of concurrent accesses and failure recovery, which simply was overkill for the scope in which DRS is used. With all relational databases filtered out, it's time to look towards document/memory stores.

There were a few options to select from the category of non-relational databases such as the distributed, column-oriented Cassandra database, the document-oriented MongoDB, and the simpler key-value store Redis. Starting off with Cassandra, DRS and Cassandra share the same embodiment of decentralization principles. Being a column-based database, it shines for applications that require a highly dynamic schema, which DRS did not. Interestingly, at the core of Cassandra distributed structure is a Chord DHT implementation, which weaknesses were addressed in a previous section: Kademlia and Related Implementations. This decentralized apparoch would favour availability and partitioning over consistency with respect to having no central point of failure yet not guaranteed delivery of transactions. Cassandra was an interesting option since it was entirely possible to fulfill the requirements of the entire DRS project with it alone. However, I needed more flexibility in the design to improve search speeds through data locality so I looked onwards for the perfect data store.

Next up, MongoDB at first seemed to be the best for our persistent requirements. They provide fast key-based lookups and indexes to help traverse large datasets on reads. The database is full featured and favours availability and partitioning over consistency through the default out of the box configurations. Transactions are deemed complete right after it is sent off, even before the write to disk operation has started. Like Cassandra, MongoDB also excels at distributed architecture, taking a sharding approach to very large documents. Since the sharding mechanism tends to shard based on the key, this would be a provide a nice horizontally scalable solution when we want to partition each DRS node's keys between shards. In these terms, MongoDB would have been a scalable solution if we had centralized our system. In comparison to DRS, the Kademlia protocol essentially provides similar sharding properties, using its locality addressing scheme, so MongoDB is still quite an overkill for the needs of our system.

The last option, Redis earned the right to exist in the DRS tech stack choose due to its simplicity, overall lightness, and nice storage formats through built-in data structures. The highly attractive feature of Redis was its single core, in-memory key-value characteristics capable of persistence through a append-only file storage (AOF) format. AOF allows Redis to keep most its keys and values in-memory for fast access while persisting all changes made to the store through a file that records each transaction like a ledger. Resetting the database involves a lengthy playback for the AOF file on startup, but for a trade off between fast read and write operations, the wait is well worth it. Redis was designed to support one local process which made it a perfect for DRS. In addition, the single threaded nature allowed the data store to consume predictable amounts of resources under heavy loads. Typically, the memory store would run side by side with the DHT and API server under one host and not across a network, which helps ensure consistency and availability. The biggest selling point was the format which the data was saved closely matched the lookup and retrieval process and format. Since the Redis was a key-value store that had built-in data structures which supports lists, maps, and sets, it became very easy to associate a product barcode key to a list of reviews while preserving the natural ordering of their insert date. A query does not need any extra effort filtering or condition matching. Queries has been simply reduced to "get range 0 to 5 from key x" to get the last 5 reviews for key x. With the full Redis executable coming in at only 784 bytes, the entire binary file could be bundled with the installation process without the end user having the need to install dependencies. Simplicity became an advantage, thus, Redis was selected as the persistent storage system for DRS.

5.3 Kademlia Library

Chord, Pastry, and Kademlia all have Java implementations for their respective protocols but none had as much documentation, GitHub activity, appraise, and users as TomP2P, a nonblocking Kademlia implementation and extension. There are a few distinct features that give this library an edge over others. This section will be discussing the library in general while providing an overview of the security mechanism, the concept of B-Trackers [11], and bloom filters [12].

TomP2P is an extended DHT library because it serves additional operations over the standard interface: $put(key, value)$, $get(key)$, $contains(key)$, and $remove(key)$. On top of those methods, the library introduced $putIfAbsent(key, value)$, $add(key, value)$, $send(key, value)$, and $digest(key)$. The first two are self-explanatory, however, the add interface allows a list of objects to be associated at one location of the DHT. The send interface allows a higher level of granular control to which the DHT node can send the data to, limiting the number of routing hops needed. Lastly, $digest(key)$ allows a node to retrieve configuration settings from another node directly, such as bloom filter settings. Although the granularity of the control is more suitable for under the hood operations, they are provided to the user just in case. The $add(key, value)$ operation was found to be very useful to attach a list of reviews, but after considering load balancing, the design shifted towards $put(key, value)$ instead to equally dissipate the reviews across a few, but very local nodes.

B-Trackers were discovered by the author of the library et al and is another feature in TomP2P that provided for better load balancing and reduces the number of duplicate messages sent to a node's known peer list. The canonical tracker is usually implemented with the centralized model, like how a Bit Torrent's announce find peers, where one peer is responsible for the tracking. In Tom's implementation, every peer becomes a tracker that keeps a maximum k entries, the maximum number of peers a node will track and help route. In the implementation, the author advised 20 would be generally enough. Updating the tracker data assumes a pull-based message exchange protocol. The purpose of a tracker is to aid the resource discovery phase, mapping resources to other nodes where the physical data could be located. The primary strategy faced with B-Trackers is to confront load balancing to reduced network hopping by increasing the number of replica files proportional to the content's popularity. There are two types of trackers, primary and secondary. If a node with address ID x has a popular file, the nodes close to it (based on the XOR metric) are elected as its' primary tracker. The number of primary trackers is controlled by a configuration setting called the primary tracker replication factor. Primary trackers also are limited to only hold a constant number of endpoint addresses per resource (the file). Each endpoint address is called secondary trackers which could be any node in the network that stores the duplicated file. A get(key) request will use the XOR routing metric to query a key. There is a chance the query route will connect to the real peer with the original file and or the primary peer. If the primary peer is hit, the addresses of all secondary peer where the file may be held by is returned. The originator will then attempt to connect with a secondary tracker to retrieve the file, if one secondary peer fails, then it can try another. In comparison to other works, the BitTorrent protocol that uses Peer Exchange (PEX) messaging protocol is less efficient than the distributed B-Tracker since a pull from PEX will generate a random subset of peer information with some irrelevant peers that won't have the file that the orignator needs. B-Trackers improve load balancing on top of the pure DHT implementation by relieving 'hot-spots' on popular addresses through routing the request to a secondary peer.

Bloom filters are a space-efficient probabilistic data structure that is used to test if an element is a member of a set. These filters are used in conjunction to B-Trackers to help determine if primary and secondary tracker have an element key during a $get(key)$ request to avoid redundant mappings of files. The added efficiency is used by peers to avoid rediscovering already known peers, thus avoiding the coupon collector's problem [13]. The original request will pass their bloom filter forward with the request, then get a reply that will consist of a set of addresses that are not in the bloom filter. This technique helps reduce the bandwidth needed for a $get(key)$ request during the peer discovery stage while trading off some extra bytes used during the transmission of the bloom filter itself.

In summary, TomP2P's Kademlia implementation provides unique set of tools that ascertains the bundle as one of the most powerful DHT libraries on the market. It's use of B-Trackers and bloom filters help optimize the efficiencies of peer discovery and routing throughout the network.

5.4 Server Side Framework

Frameworks are supposed to make life easier for the developer, thus it's was a crucial for DRS to be equipped with the right tool. Aside from the DHT network aspect, the system must have a web service component to provide end users with a representative state transfer (REST) API, delivery client side rendered reviews, and power a backend dashboard for managing the system. In an ideal case, there should be no software dependencies that the end user would have to install; all libraries and frameworks should be statically compiled into the final product. What we essentially want is a web application that can handle a set of routes, serve web pages, static content, and dynamic content without the end user having to have to install a full-blown web server like Nginx, Apache or Tomcat. DRS requires a web server to be embedded into the system to serve content.

The choice of the server servlet container ultimately resulted in Jetty. Performance wise, it's difficult to make an argument that Jetty is better than its alternatives unless a through investigative analysis is done with respect to other providers such as Tomcat, Resin, Grizzly, or Undertow. The information described in this section comes from a comparison report of Java HTTP server conducted by the JRebel company, a Java development tool company. The main reason for using Jetty as our embedded HTTP server due to the binary size of the overall library, just 8mb. The lightness of the server means a stripped down, bare bones HTTP handler. Therefore, we needed a RESTful web services library (JAX-RS) to complement. That decision means to pull in a third-party framework named Jersey. The Jersey JAX-RS REST framework is used to turn what had been a static web server into a NodeJs-like application structure. This was the first time I had used Java as a server language, but the experience was very comparable to Node.js or ASP.net, except configuring routes is done through annotations. Since the server was embedded within an executable, there was a load of configurations to set. As the compiled app is bundled within a JAR file, a dynamic resource folder had to be defined. Unlike NodeJS where uploads can be saved under the project source folder, it's not possible to dynamically add files to a JAR. While configuring, a standard folder had to be created under system user's home. The great thing about Java is that it's platform independent, thus making it is easy for the JVM to find the "System.home" constant for both Windows and Unix operating systems where an upload folder can be dynamically created at runtime. When user-uploaded static content is requested, Jetty had to be configured to search its internal JAR for application resources first before falling back to the native upload folder. The fallback signifies that the requested resource is not part of the application execution code can is external resource in the format of an image or a client side script.

No other frameworks were needed other than Jersey to keep the executable as small as possible. Great lengths have been gone to find libraries that do not bluff up the resulting DRS JAR too much. With this requirement, a trade-off was made to drop the Spring Framework, which seemly is the standard for Java MVC architecture pattern in favour of a bare-bones structure which is much more flexible. The next section will discuss how this framework was utilized to build the core REST API for data transfer.

5.5 REST API

The term representative state transfer was coined by Roy Fielding in 2000 in his doctoral dissertation [14]. REST is a form of web services to provide interoperability between computer systems on the internet. It is a standard for compliant web services to provide a textual representation of web state using HTTP request operations on stateless servers. REST is essentially a standardization to an architectural style presented by Fielding. The concept had existed since the 1970s but was popularized by Sun Microsystems in 1988 through the design of Remote Procedure Call (RPC) request-response protocol. RPC was originally designed to decouple the system from the UI layer to easily integrate newer frontends as times changed but now has become a de-facto standard for most websites and services today. XML began as the first language to pass structured data across the web which was used by many early protocols such as SOAP (still in use today). However, due to the increasing complexity of today's systems, many have opted out for a more unstructured approach with JavaScript Object Notation (JSON) as the data format of choice. DRS uses this modern paradigm, REST API returning JSON data types. It also supports a newer standard called JSON API that was designed by Yehuda Katz, a member of the ECMAScript committee on standardizing an object-oriented format for data transferring.

DRS uses REST to transfer stateless results to every component of the system by feeding the management dashboard with data and providing an HTTP service for commerce sites to fetch reviews on product IDs. The system has defined an overall structure of the request URI as follows:

$$domain.com/ < provider > / < resource > / < verb > / <?identifications >$$

The providers field defines an association with the subsystem to query, for instance, the API service. The resource field indicates the model of data to request. In object-oriented terms, the resource "review" or "product" might go here to fetch info relating to the resource. The verb section of the URI component indicates what is to be done to the resource component. Possible verbs, for example, can indicate a request to add a new review through the "new" keyword or "all" to fetch all reviews for a product barcode that is passed in. The last section refers to the identification element, such as product barcode, and is optional depending on the request verb. For example, the verb "update' must be coupled with a barcode identifier to request an update on that review ID. A snippet of the return value from an HTTP GET request made to DRS is in the appendix A. The raw data returned by this REST API can used by the requester anyway they like, however, the data was intended to be consumed by web application servers to render or aggregate the data per business logic. This serves the purpose of the REST API.

An issue that was confronted during the development of the REST system was to define who could fetch data from through the API through a cross-domain policy called cross-origin request (CORs). CORs is triggered when a script that invoked an AJAX request had originated from another host other than the one hosting the REST API. In some cases, the API may be hosted on a different server other than the e-commerce site requesting it and would be blocked. To get around this issue, I had to install a CORs policy with extremely liberal rules, allowing all hosts to accept requests from anywhere as long as the backend is reachable. In future reviews, a CORs policy can be used to restrict access to resources and block out unwanted traffic if an e-commerce site wants an open wire to request reviews from.

5.6 Client Side Delivery

The methodology in compiling the HTML doc for return was interesting in my opinion. I wanted to select the best templating engine which was non-standard to the boring old JSP. At the time, I had looked for a lightweight and fast parsing template engine and found DustJs made by LinkedIn (although now I read MarkoJs is faster and lighter) [15]. The great reason to use these templates that they provide a clean separation between the presentation logic and the controller. The controller passes a common JSON object to the view layer, then logic in the view layer (DustJs) renders the DOM on the page. The internals of DustJs worked just like jade (taught in COMP2406). A template is compressed into a long string with HTML elements and placeholders, then a JavaScript object is injected into the placeholders to finish up the DOM. The end results, just like jade, is a JavaScript function that returns a messy DOM string. DustJs semantics were common to any JavaScript templating engine, but since the server rendering the template is Java based, things got a little bit tricky. At the time, there were no simple Java to DustJs libraries for generating templates easily so I had to do it all myself (perhaps spin off a library one after this project). There are two main functional pieces needed to complete the compilation, a JavaScript parser and evaluator and a JSON object serializer to convert java objects into JSON. Java has two JavaScript engines built-in, the newer Nashorn built by Oracle that came with Java 8 and the older Rhino. With the newer engine claiming to be more performant than the old, the entire dust library had to be loaded into Nashorn. Then the template file was loaded into a string and the compile method was invoked from the loaded engine. The Java object must undergo serialization through the Jackson API which results in a JSON string. After the template was compiled, DustJs library loaded, and payload serialized, it's time to write a render script to glue the payload bindings into the template. A render script is essentially a piece of JavaScript code that interfaces with the DustJs API to trigger function calls. For what was necessary, an invocation of the render function followed by an error handler clause would do for the bare minimum. The output of the final procedure was a complete HTML doc, ready to pack into a response and send over the wire. The client would wait until the doc has arrived before injecting the iframe into a child DOM element.

The overall procedure was tested on a mock e-commerce site, hosting a page with the client side injection script coded into the page. E-commerce sites that did not want to carry the burden of hosting a DHT node can now gain value from the DRS service.

5.7 Management Dashboard Components

In this subsection, we will be examining the bits and pieces in which the back is built upon. An overview is given about Ember.js and its virtual DOM implementation with relation to other popular framework implementations such as React and Angular. A short summary follows of the UX toolkit used to design the user interface and apply the same look and feel.

5.7.1 Ember.js and Virtual DOM

Up to this point, we've discussed how the REST API is the primary source of the program state and is used to decouple the presentation layer from the application layer. The target was to keep the DRS server as stateless as possible and I've managed to adhere to this paradigm by extracting all dashboard application logic to the client side with the help of Ember.js, a single-page Model-View-ViewModel frontend framework. Ember belongs in the family of front-end JavaScript frameworks comparable to React by Facebook or Angular by Google. However, the difference is that the project is community supported and not backed by a large corporation. Nevertheless, big names such as Yahoo, Microsoft, and Groupon all contribute and support the framework's evolution due to its philosophy and design. Ember is a very opinionated framework, which essentially tries to convince you to do everything the Ember way. This creates an incredibly steep learning curve as users must understand how the framework does things before becoming productive at it. Once the developers get the hang of the framework, then they will notice the intuitive project organization and layout, the powerfully bundled command line interface (CLI) component that had once made Ruby on Rails an attractive alternative, and neat features such as code splitting that does a professional job at preparing the application for production. These features make the ember-cli have a tremendous impact on the productivity of its projects. Ember is also one of the fastest evolving frameworks with the most bleeding edge of web standards and crowned as a pioneer for the JavaScript language. All thanks to Yehuda Katz, one of Ember's cofounders, previous a core member of the Ruby on Rails and jQuery team, and a member of the TC39 committee which is responsible for the evolution of the JavaScript language. With all reasons considered, Ember is a defining framework for modern day JavaScript applications and the skill will be highly useful in my future career.

Ember, after compilation, is client side application that features a virtual DOM. Virtual DOM is essentially an in-memory tree with JavaScript functions to generate HTML tags that become rendered on the page. This means the developer only must serve an empty HTML doc root to the browser while JavaScript is run to populate the page. There are a few design considerations for this model, whether it is optimal to run virtual DOMs. The benefit of the virtual DOM is that it abstracts the physical web page into memory, which allows manipulations to be fast [16]. The concept of DOM was invented in the 1990s by Netscape and originally was not meant to be dynamic, thus, it lacks optimizations when many elements are required to be manipulated at once. It's very common for a modern web page to move over 1000 nodes at once, which may take over a second to finish. If the DOM tree is abstracted in memory, then we gain more algorithmic control over what to render and when. Running efficient diff algorithms and batching DOM reads and writes are some techniques that enhance the performance of DOM manipulations. There are a few different approaches to virtual DOM theory by the three major frontend frameworks, Angular, React, and Ember that we will discuss. The approaches that these frameworks take is very reminiscent of Java's garbage collection methods, the copying method and the mark and sweep. Prior to these frameworks, updating the DOM was up to the programmer which if done wrong, can lead to poor user experience. Having a credible framework take care of this burden seems to be the way the web is going towards, who wouldn't trust Google or Facebook with their code, right?

Angular's virtual DOM works through a method called dirty checking. Dirty checking works by keeping track of new elements added to the DOM with a dirty bit. When an updated has happened, then Angular will check every element it is tracking to see if changed, then update if necessary. The benefit of this approach is the batching of model/view changes, then when the view is ready, it can reflect all those changes at once. Since a separate mutator checks the dirty bit when it's ready to update, there becomes a loss of traceability to what components influenced that change. Although it sounds that Angular has a stop-the-world approach to check and re-render different DOM nodes, but the way they've batched updates runs relatively fast since it relies only on JavaScript equality checks in memory. Although duly noted, very large DOM trees will suffer performance degradations which need to be addressed with more creative optimization trickery.

React's virtual DOM takes a different approach which is more alike the server side rendering days. It uses this method to minimize the number of elements to be updated on the physical DOM through a series of in-memory "simulations". Every component change will trigger the rendering engine to build an entirely new DOM tree. That's right, React always keeps two full DOM trees in memory, the new one, and the old one. React then runs an efficient diff algorithm on the two trees and find the minimal number of changes it takes to mutate the old DOM into the new. Once this diff algorithm ends, it pushes the changes into the old virtual and physical DOM and discards the other tree. Seemingly inefficient and likely plagued with scalability problems when tackling a very large DOM or trying to keep a high render frame rate, there has some benefits on why Facebook chose to design the framework like this. The new DOM is generated every time like a server side rendered the page, which alleviates the overhead of tracking changes through event propagation. Since everything is done in-memory, every full render runs very fast. An additional add-on I would like to note here is that there exists a third-party add-on to optimize this behaviour for large DOM trees. They enhance the core virtual DOM tree with a persistent data structure allowing it track the latest changes from the latest time slice. A single traversal through the tree can aggregate the newest changes since the last time slice and render them individually onto the physical DOM. Persistent data structures were invented by Driscoll, Sarnak, Sleator, and Tarjan in their debut paper

"Making data structures persistent" in 1986 where they showed how to track changes in pointer-based data structures using the fat node or the node copying method [17].

Lastly, Ember's approach differs from the rest by taking on an observable binding approach in their implementation of virtual DOM. In my opinion, it is the most efficient in terms of operations required proved by benchmarks from a blog by Auth0, an authentication provider. Ember's take is to solve the problem with the observer pattern, but you knew that already from the MV-VM (view model) pattern it's designed with. The view properties can be attached to a model or controller which means it's listening for update events on the view. The tradeoff Ember has made with this approach is to do a lot of work when the application initializes and bind all element listeners to events. The high preprocessing fee pays off once the app is loaded which allows updates to modify DOM elements directly and efficiently. Since this pattern is not built into JavaScript or the browser, modifying the element requires the develop to interact with the Ember API for all modifications of the models which can be "inconvenient" at times. Now that we understand the different approaches to virtual DOM let's get along with the dashboard implementation.

5.7.2 UX Toolkit

Ember was a good choice to implement the frontend dashboard application, however, I used a few more tools to speed up the development of the app. To power the overall look and feel of the UI, I opted to import a UI toolkit called Semantic UI, a competitor to Bootstrap. Both come with UX elements that practically will fulfill everything a frontend developer could ever need. In comparison, Semantic UI comes with a highly configurable library that allows the user to cut features they don't need and change the look and feel for the theme of the UX kit. Semantic is written the language called LESS, a CSS preprocessor language. In LESS, you can tweak the variables and behaviour of Semantic UI before a compilation takes place which a CSS file is outputted. Semantic also has great CSS class names that make defining behaviour very intuitive, for example "class='ui stackable equal height grid'" will format a grid to stack elements when screen space is limited and try its best to preserve equal heights on each stacked element.

5.7.3 User Accounts

Since DRS deals with sensitive data that could impact the longevity of the e-commerce stores it serves, I thought it was necessary to password protect the admin interface. I took a very simple approach to this function. If the passwords and username match, the server will return an HTTP 200 and allow the client to create an credential stamp in the browser's local storage. The existence of the local storage represents an active session for the administrative user. Previously implementing simple authentication methods, I have always used a fixed, hard-coded salt. I discovered an algorithm called Password-Based Key Derivation Function 2 from the RSA laboratories' Public-Key Cryptography Standards that allows a pseudo random salt to pair with the hashed password. Since the project is going to go open sourced, I thought this technique will make it harder for a rainbow table to brute force the password with a known salt

6 Evaluation

Overall, DRS has been a success. This section is dedicated to breaking down the aspects of the project that are completed and my methodology used for testing.

6.1 Testing

There are two major areas to test: the REST API and the management dashboard. I've taken a different approach test each component.

6.1.1 REST API

Since the REST API is implemented as a Java web server, it's important to ensure its' ability to launch from a fresh install and expect all of the the routes all work. Normally, an application such as this one would require a formalized testing framework to accompany the deployment of the system. This software engineering aspect was dropped due to time constraints. As of now, only one developer (me) is working on the project. If the project gains following in the future, unit tests are key to implement quality assurance. If I had implemented unit testing now, white box doesn't really help catch any bugs or defects. The usefulness of such test aids a team of developer to rather identify "what has changed" instead of testing if something works. The practice promotes carefulness when developing new features without break old ones. In this respect, unit tests only become helpful once a software product has grown to near maturity. Nonetheless, without the presence of unit tests, I have utilized my own set of test tools that's proven to have become very helpful during my development and occasionally have caught my unintentional changes.

The communication model implemented by this project is the request-response model which requires an unorthodox approach to quality assurance. Server route function return void and responses are masked with magic-ness of the Jersey JAX-RS framework. A route handler builds a response object internally, then commits the return value to the framework to be taken care of "automagically". We will need to stand on the requester's side of the model to interface with the API. Although I would note, there certainly must exist some type of frameworks specific to testing Jersey routes that I am currently not aware of.

Talk to any API developer and they will tell you about a handy tool called Postman. This little chrome extension is an HTTP request client like the Linux command "curl" but with a GUI to set configuration and scripting engine for automated tests. Aside from being able to trigger any type of HTTP requests, Postman is the complete toolkit for testing API-driven applications. Not only did I use Postman to test whether my routes worked or not, I have also set up a series of integration tests used to populate the network with sample data and test edge cases after new changes were added. Specifically, I have set up and saved a list of POST, PUT, and GET requests that add new unapproved or approved reviews into the network and designed tests to fetch the data to ensure a return subject. These query sets were heavily utilized since I always wipe the database with each build and run these queries after to populate the test. They have helped me on occasions where I had renamed model settings but failed to change the JSON binding names in the association.

Testing the success cases are important but DRS does not lack in the error handling department either. Exceptions during error cases have been mapped using a custom class in the Jersey framework to return a JSON object with a detailed error message on the procedure that failed. These messages cover real runtime exceptions, validation errors, and illegal application states for DRS. Although illegal states and runtime exceptions are not something generally expected from a production system, building these into the API helped track a lot of errors over the wire in the form of messages.

Validation messages are more common to see. To catch these errors, I had built Java interfaces using a new method of defining syntactic metadata, annotations. These annotations would flag required parameters and trigger the validation interfaces with respective method parameters. If you're confused what I mean, imagine a method parameter for an object that implements a "Validatable" interface that can trigger validation on itself. When the client POSTs or PUTs a JSON object to the corresponding route, Jersey can automatically map the request parameters to a Java object. If the JSON object is no match for the parameter type, a runtime exception is returned. If the mapping is successful in at least one field, the property is then set into the object and my custom annotation kicks in. This annotation will call the "validate()" function within the object since all objects that are used this way must all implement this interface. If the validate-able object is a subclass, it will validate itself, then propagate the call to the validate function up its' inheritance chain. At the end of the validation process, a boolean return value will determine whether a validation error is present or the normal execution of the route method is to proceed.

With the help of Postman, testing validation and error maps are made easy. The responses to the requests are formatted nicely for inspection. Obviously, testing an error path requires setting up a request with intentional missing or invalid parameters, such as special characters or HTML. All data that's received from the client side is encoded into HTML entities to prevent cross-site scripting attacks of rendering HTML or evaluating scripts embed inside the review snippets. A state diagram is included as Figure X in the Management Dashboard section to indicate all possible code paths including the results of both this and the latter components.

6.1.2 Route Map

The following routes are defined to with an assumption that we are hosting the DRS application at *domain.com.*

Resources	
domain.com Will return the dashboard application index page	GET
domain.com/somefile.extension Will return the resource if exists or else 404	GET
Account API	
domain.com/api/account/new Create a new account. Require an identification and password string. Returns a success or fail.	POST
domain.com/api/account/update Update an account. Require an account model (Appendix B). Returns the ID of the success or fail account.	PUT
domain.com/api/account/login Verify username password that's posted to the server.	POST
domain.com/api/account/fetch Fetches account detail by email. Returns a user model.	GET
domain.com/api/account/upload Uploads a file in association to ann email that's passed in. The following request uses multi-part form data to stream the file over to the server. Returns success or failure.	POST
Metric API	
domain.com/api/metric/all Returns all the data that will be displayed on the analytics page.	GET
Review API	
domain.com/api/review/new/{productName} Receives a complete review model (appendix A) and a product name, when a save is complete, a full review model is passed back with the IDs.	PUT
domain.com/api/upvote/{locationId}/{contentId} Upvotes a review based on barcode (locationId) and review content (contentId) and returns the ID of the review if successful.	PUT
domain.com/api/downvote/{locationId}/{contentId} Same as upvote just renders a downvote.	PUT

Table 1: Route Table for the REST API

domain.com/api/review/embed/{locationId}	GET	Returns a compiled HTML iframe for at most five reviews based on a barcode locationId
domain.com/api/review/update/{locationId}	PUT	Requires a full review model in JSON to update, then returns the updated JSON when successful.
domain.com/api/review/accept/{locationId}	PUT	Requires a full review model in JSON in acceptance domain, then the review will be published and returning the full review.
domain.com/api/review/deny/{locationId}	PUT	Same as accept, only it will delete the acceptance review from the acceptance domain.
domain.com/api/review/get/{barcode}	GET	searches the network for the barcode related to the request and passes back a list of review models. This URL can be attached with queryparams relating to the page as well as step, which determines the amount of reviews to return from that page.
domain.com/api/review/approval	GET	Returns a list of review models, currently in acceptance, and are tracked by the current node.
Product API		
domain.com/api/product/all	GET	Returns a list of products along with all reviews for that product. This query will search within the space of all tracked product reviews on the current node. This query can be controlled but a limit query param for page and step.
domain.com/api/product/search	GET	Takes a query param for a string query and does a matching for different aspects of the review set.

Table 2: Route Table for the REST API

6.1.3 Management Dashboard

I utilized a built-in module that comes with Ember.js called PhantomJS to test the dashboard component. PhantomJS is built on top of WebKit making it a headless client which means it is essentially a browser without the GUI. It's a powerful test and automation framework as it's able to make any requests just like the browser in addition to being programmable. I can set up test to make requests for a particular page and compare return values of properties or features in the response to the expected values. When errors do happen, PhantomJS can take a screenshot of what the error state on the browser looked like and save it to disk for later inspection. This benefits of this testing framework drives many popular projects like Bootstrap, YUI3, jQuery, and Modernizr to have adopted this technology.

Ember is one of the early adopters and the framework so it became tightly integrated into it's the Ember CLI. Ember development is different than the traditional process as it adopts the Ruby on Rails philosophy of program-generated code. The CLI is very helpful as it can generate every possible class, model, or Ember component by providing you barebones JavaScript template for you. In some cases, it may even edit application-level configurations, for example, if you created a route, the CLI will automatically write code for you to enabled the router to handle your new code. Not surprisingly, it also generates primitive tests for each file that's created with the CLI.

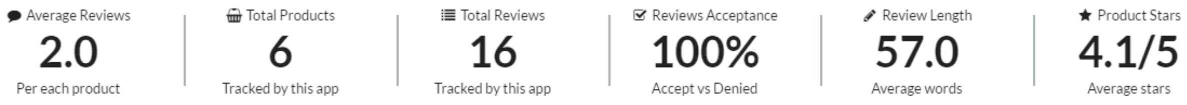
As I've mentioned in the "Implementation" section that Gradle is used to manage my application build process. I've integrated JavaScript PhantomJS testing alongside my Java JAR compilation. Each time the DRS system is built, it triggers Ember to build for the production environment which invokes 150 test cases I've set up. Production Ember is resource minimized to decreasing the size of the request payload which can help improve browser response times on the first load. The production build step will combine all sparse JavaScript and stylesheets into one, then it will try to shorten all long variable names to one or two characters, thereby saving space. Resource minification is useful in HTTP 1.1 since a request for a single resource such as an HTML doc, JavaScript file, or stylesheet requires a new port to be opened on the computer for TCP transmission. However, with HTTP 2.0 spec fully approved, this file concatenation step can essentially be avoided due to a new feature on HTTP 2.0 compliant webservers. SPDY, an HTTP 2.0 compliant server built by Google, can multiplex multiple files through the same TCP connection maintained with the web server. Aside from breaking down the dashboard with unit tests, manual user testing was very important to cover the entire scope of the application.

6.2 Functional Breakdown

This section is dedicated to showing off the DRS software, it's functionalities and user interface. Care has been taken to ensure functional requirements are paired with visual elements of the final product.

6.2.1 Management Dashboard Analytics

With full utilization of the DRS DHT, review records can be fetched from across the network through the REST API service and into the frontend Ember application. The JSON result is then beautifully rendered by the Semantic UX toolkit. Aside from the standard search and approval functionality, the dashboard has a simple set of analytics that are used to provide the admin with some usage statistics on the home page of the dashboard. The full page can be view on appendix C. The data for which these statistics are based on are aggregated usage data of DRS then post processed into the display of the first-page load. Due to the cost in computing the entire set of analytic data set, the results are cached for fifteen minutes before resetting. There are thirteen metrics that are featured on the front page and this section provides a survey on the implementation on each metric tracked.



Average Reviews	Total Products	Total Reviews	Reviews Acceptance	Review Length	Product Stars
2.0	6	16	100%	57.0	4.1/5
Per each product	Tracked by this app	Tracked by this app	Accept vs Denied	Average words	Average stars

Figure 3: Top 6 analytic fields

1. **Average review per product**
   A simple aggregate function that counts the average reviews each product has. This metric can show how well distributed the review are for each tracked products. An administrator would effectively attempt to improve this metric over the course of DRS runtime.

2. **Total number of products tracked**
   Aggregated number of products in the system, which also refers to the subset of products that are tracked by the current DRS node amongst the entire DRS system.

3. **Total number of reviews tracked**
   Aggregated of total reviews that are amongst the products that are tracked by the current DRS node.

4. **Acceptance rate**
   A ratio of accepted to denied reviews for the current DRS node. This ratio can be used as an indicator on the quality of reviews, such as finding too much spam, hosted by the commerce site.

5. **Average length of review**
   A second indicator on the quality using word count as the metric to evaluate good from bad reviews. An admin can use this indicator to set up restrictions to on their product pages to enforce higher quality reviews.

6. **Average product stars**
   A sentiment metric of an aggregated stars for each product that the current DRS is tracking. Provides the admin high-level overview on their e-commerce site on consumer opinion.

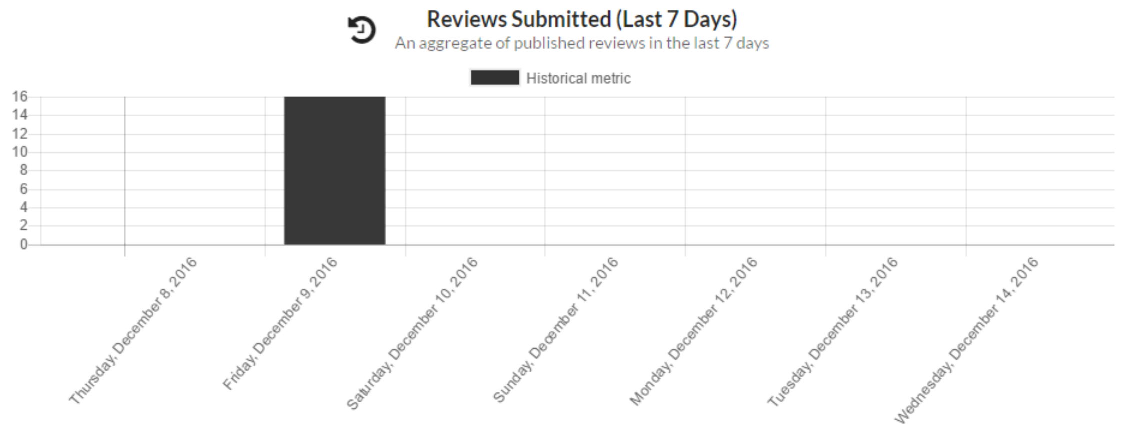
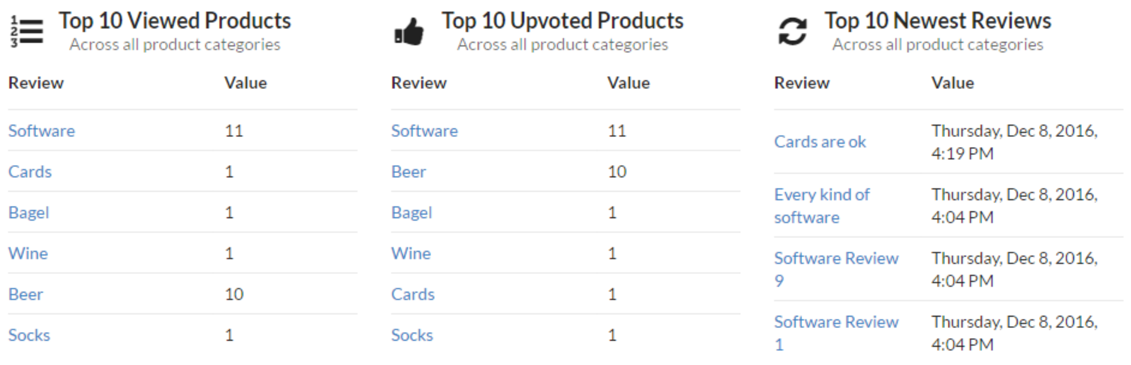


Figure 4: Number of reviews submitted and accepted in the past week

7. **Number of reviews submitted and accepted in the past week**
   Reports a history of the review system for the past seven days, presented in a bar chart with each bar referring to the number of approved reviews per day.

8. **Top 10 most viewed reviews**
   Each review that is queried and sent over the wire is considered viewed. This metric



Top 10 Viewed Products
Across all product categories

Review	Value
Software	11
Cards	1
Bagel	1
Wine	1
Beer	10
Socks	1

Top 10 Upvoted Products
Across all product categories

Review	Value
Software	11
Beer	10
Bagel	1
Wine	1
Cards	1
Socks	1

Top 10 Newest Reviews
Across all product categories

Review	Value
Cards are ok	Thursday, Dec 8, 2016, 4:19 PM
Every kind of software	Thursday, Dec 8, 2016, 4:04 PM
Software Review 9	Thursday, Dec 8, 2016, 4:04 PM
Software Review 1	Thursday, Dec 8, 2016, 4:04 PM

Figure 5: Top 10 s Analytics

records the top 10 reviews that have been fetched by a client. An admin can use this metric is pinpoint which products are getting the most traffic which can help identify which products to optimize.

9. **Top 10 most upvoted products**

10. **Top 10 newest approved reviews**

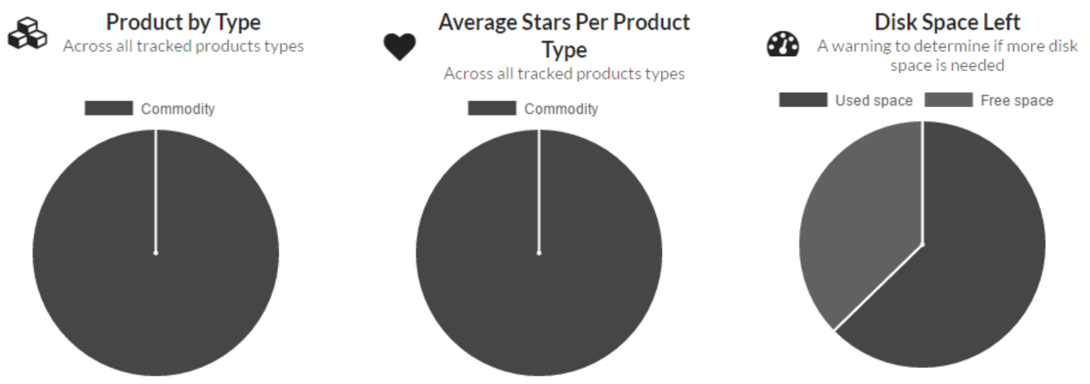


Figure 6: Extra Analytic Fields

11. **Product type ratios**

    Some commerce sites may sell more than one type of products. In DRS, reviews are separated by their types: commodity, service, hotels, etc. This metrics helps map out the ratios of these product types among all offerings that are currently tracked on the DRS node.

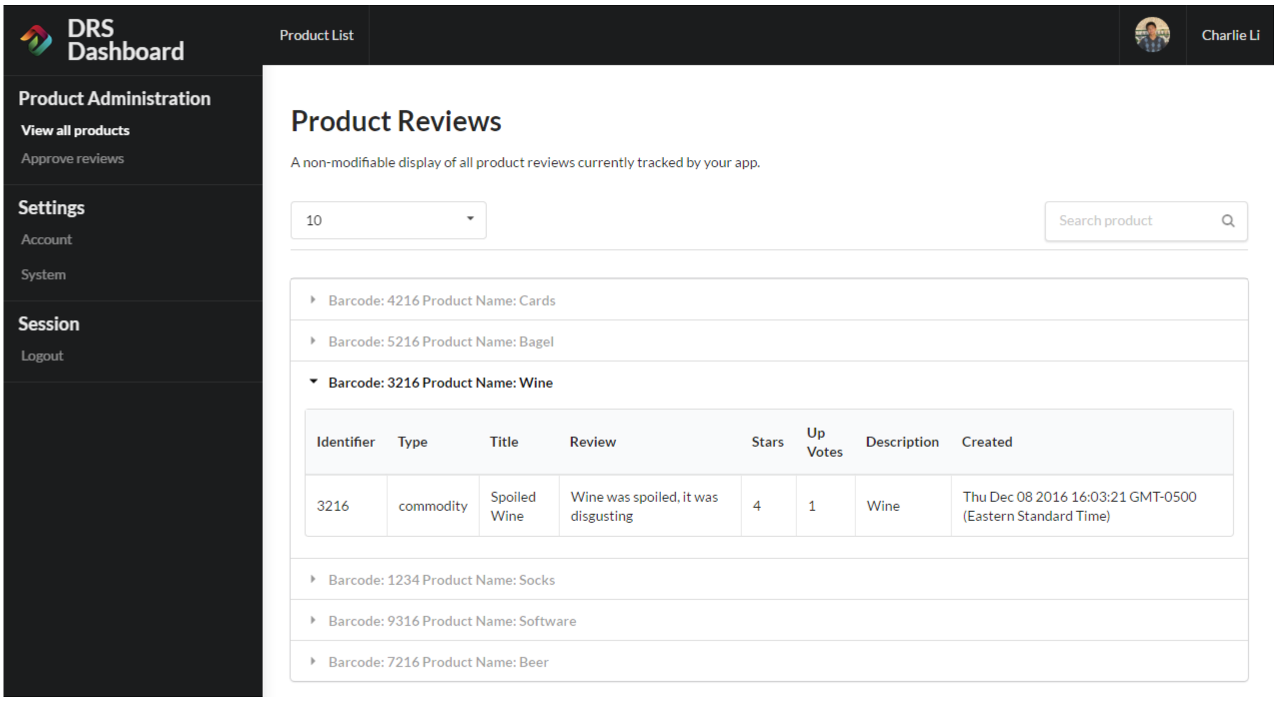
12. **Average number of stars per product type**

Like the metric above, this indicator gives a slightly more granular outlook on the average stars per product type, than the all-inclusive metric 6.

13. **Amount of disk space used**

    Space on disk can become an issue since the what's saved on the DRS node does not necessarily have to have been generated on the site that it's serving. Space needs to be monitored to help predict when it's time to expand memory and disk space on each node.

6.2.2 Viewing Tracked Reviews



Product Reviews

A non-modifiable display of all product reviews currently tracked by your app.

Barcode: 4216 Product Name: Cards

Barcode: 5216 Product Name: Bagel

Barcode: 3216 Product Name: Wine

Identifier	Type	Title	Review	Stars	Up Votes	Description	Created
3216	commodity	Spoiled Wine	Wine was spoiled, it was disgusting	4	1	Wine	Thu Dec 08 2016 16:03:21 GMT-0500 (Eastern Standard Time)

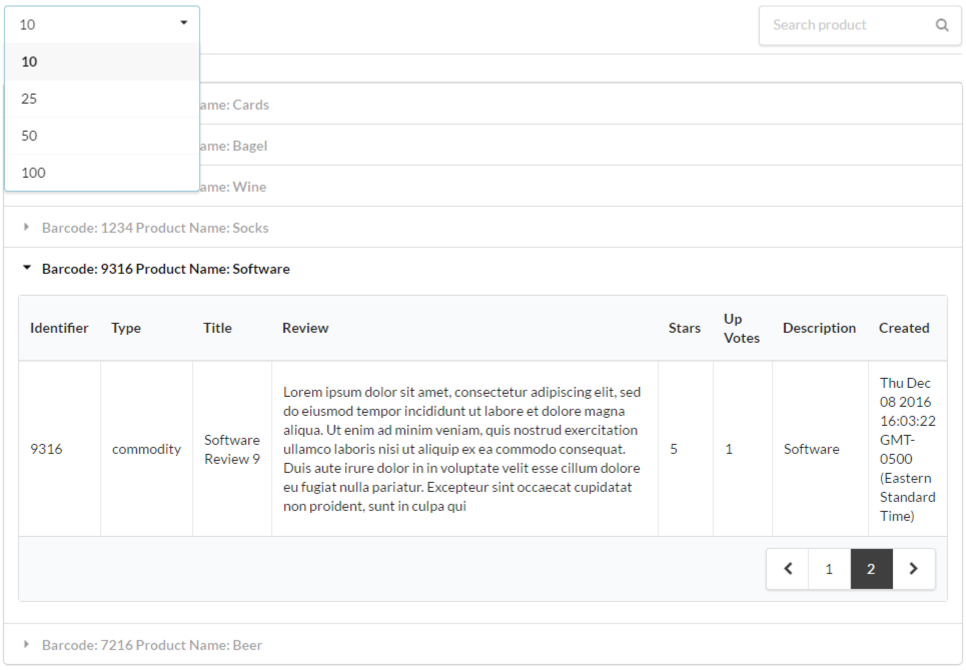
Barcode: 1234 Product Name: Socks

Barcode: 9316 Product Name: Software

Barcode: 7216 Product Name: Beer

Figure 7: Tracked reviews page

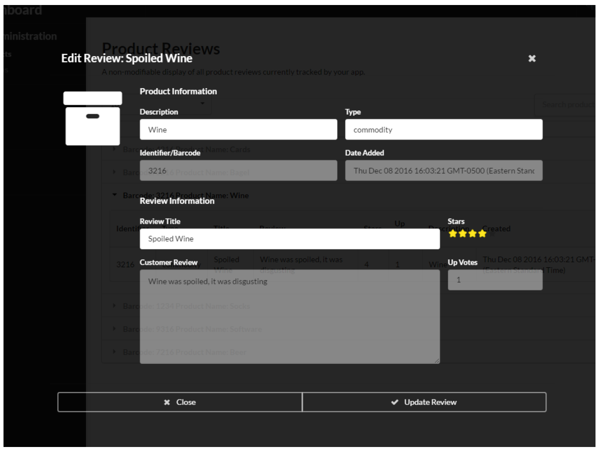
Tracked reviews are product IDs that have been used by the client. To add reviews to the tracked set, the client only needs to request for an existing product ID in the network. The tracked reviews list is also part of the management dashboard and has a separate section to browse each of the reviews for each tracked per product. From this interface, the user is provided with read-only permissions which align with DRS's "no-editing after published state" philosophy. Since the number of reviews can become quite large, the page has implemented pagination on the records. The use gains direct control to how many records are displayed per page on the product and review level. The user can adjust the number of products to show per page as well as the number of reviews visible per product.


10
10
25
50
100
Search product
ame: Cards
ame: Bagel
ame: Wine
Barcode: 1234 Product Name: Socks
Barcode: 9316 Product Name: Software
Barcode: 7216 Product Name: Beer
1
2

Identifier	Type	Title	Review	Stars	Up Votes	Description	Created
9316	commodity	Software Review 9	Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui	5	1	Software	Thu Dec 08 2016 16:03:22 GMT-0500 (Eastern Standard Time)

Figure 8: Product pagination element

Some colleagues I've consulted have raised an issue with me that the tabular display of the records is boring. That got me to think of another approach to display singular records through a pop-up (modal) window display. I believe the effects and animations of the popup invocation look nice as the background of the page is dimmed down to focus on the review. Building this effect was not difficult either, Ember has very modular reusable components and I was able to find alternative areas of the application to display the review details on, for example, the search and approval functionalities.



Product Information

Description: Wine

Type: commodity

Identifier/Barcode: 3216

Date Added: Thu Dec 08 2016 16:03:21 GMT-0500 (Eastern Stanc

Review Information

Review Title: Spoiled Wine

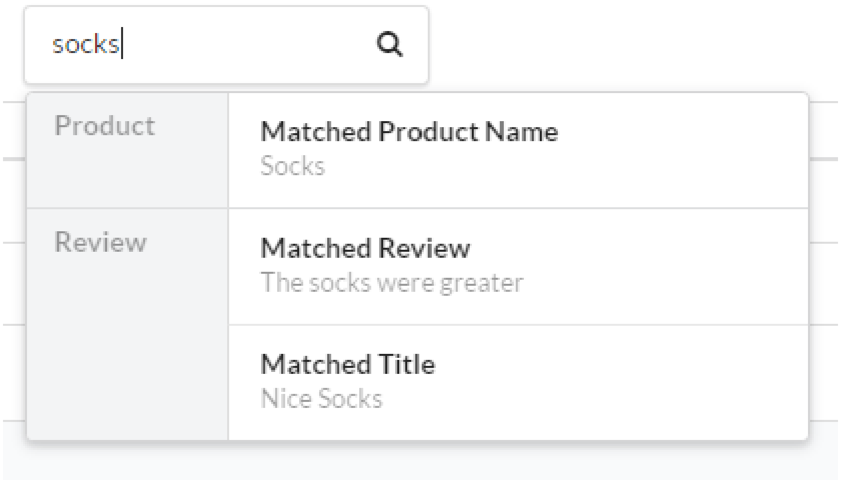
Stars: ★★★★

Customer Review: Wine was spoiled, it was disgusting

Up Votes: 1

Figure 9: Product inspection modal

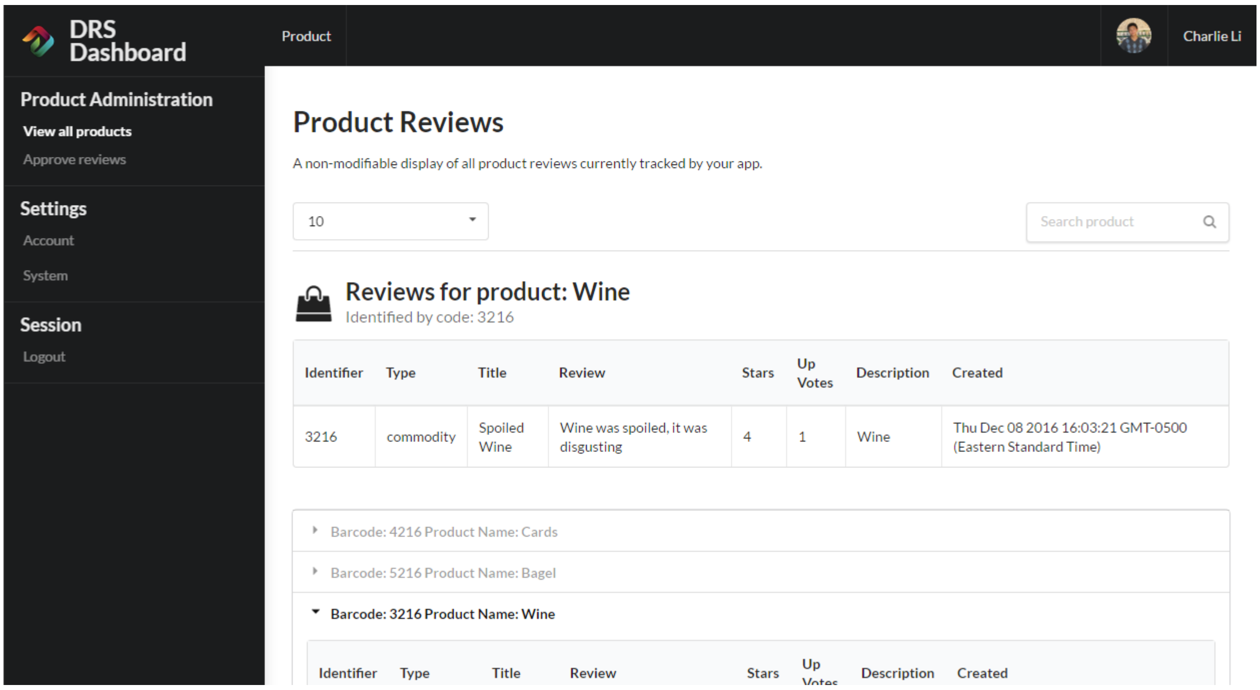
Next up, the search functionality is shown, one of the coolest widgets in my opinion. The searching module implements an AJAX call to the REST API to fetch for matches on the query string in multiple dimensions. The search functionality is set up in a way to search based on categories. What this mean is the results are divided up between matchings on product categories, review categories, and general identifiers. The search will run string matchings on the product model for the product name, barcode, product types, and all other product attributes. It will check the review model (see appendix A) for any matching in the review text, review title, and other review model properties. It will also search for common identifiers such as the hash table location, domain, and content keys for advanced users that want to run a more granular search. The search space will examine only a subset of the entire review system that only consists of tracked reviews by the current node. These tracked reviews are cached either in DRS memory or disk so the functionality does not require network hops and improves the response rate. Each search category can be fired in parallel to further improve response times when the search space becomes large. When the search is completed, the results are serialized into a JSON response object and returned to the dashboard to look something like the image below.


socks

Product	Matched Product Name Socks
Review	Matched Review The socks were greater
	Matched Title Nice Socks

Figure 10: Product search functionality

The contents of the search results above are clickable. Since they are divided by categories, clicking a result of each category will bring up the detailed component onto the page. That is, clicking on product category will bring all the reviews of that product onto the page and transition the user to the correct section where the product result is displayed. Clicking on a review category will pull the exact detailed review on the current page for viewing. This review view is recycled in many parts of the site.



Product Reviews

A non-modifiable display of all product reviews currently tracked by your app.

10

Search product

Reviews for product: Wine
Identified by code: 3216

Identifier	Type	Title	Review	Stars	Up Votes	Description	Created
3216	commodity	Spoiled Wine	Wine was spoiled, it was disgusting	4	1	Wine	Thu Dec 08 2016 16:03:21 GMT-0500 (Eastern Standard Time)

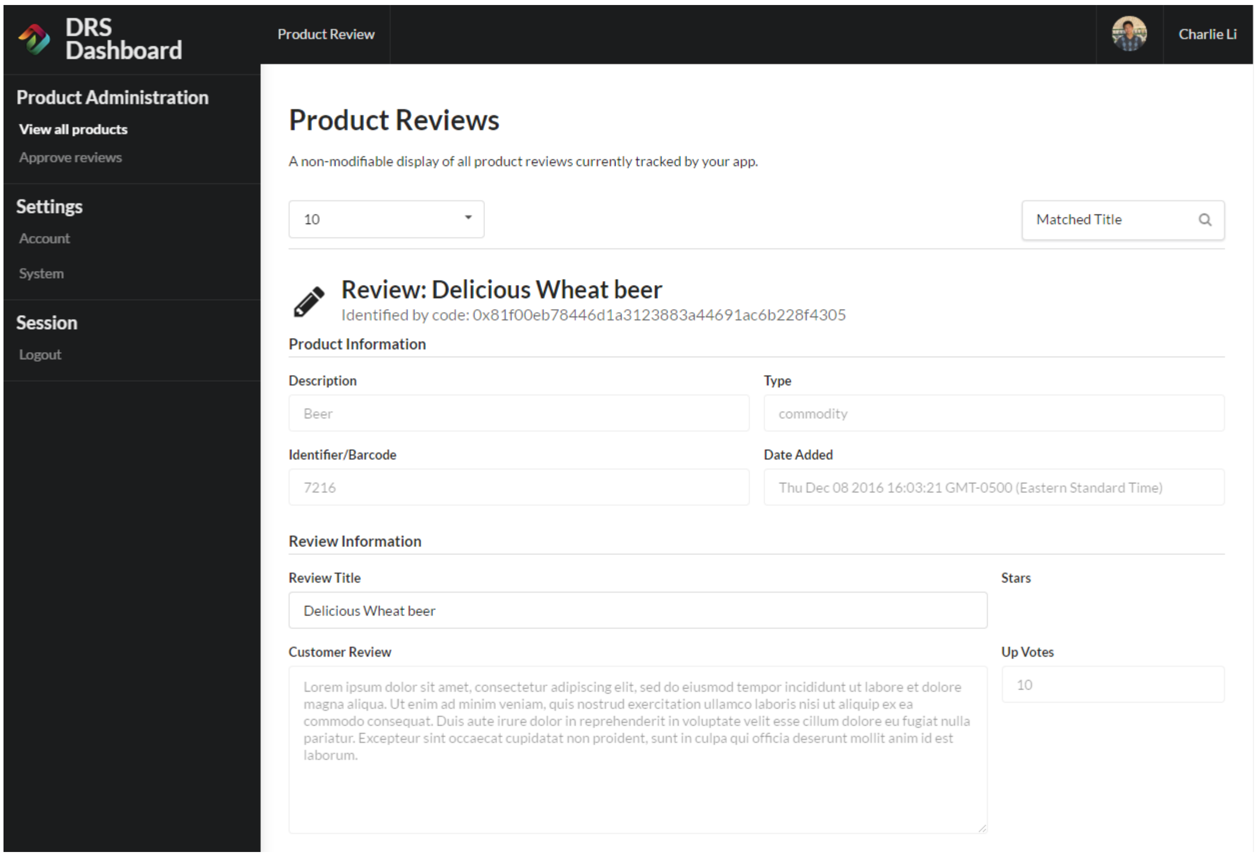
Barcode: 4216 Product Name: Cards

Barcode: 5216 Product Name: Bagel

Barcode: 3216 Product Name: Wine

Identifier	Type	Title	Review	Stars	Up Votes	Description	Created

Figure 11: Product search result



Product Reviews

A non-modifiable display of all product reviews currently tracked by your app.

10

Matched Title

Review: Delicious Wheat beer
Identified by code: 0x81f00eb78446d1a3123883a44691ac6b228f4305

Product Information

Description: Beer	Type: commodity
Identifier/Barcode: 7216	Date Added: Thu Dec 08 2016 16:03:21 GMT-0500 (Eastern Standard Time)

Review Information

Review Title: Delicious Wheat beer

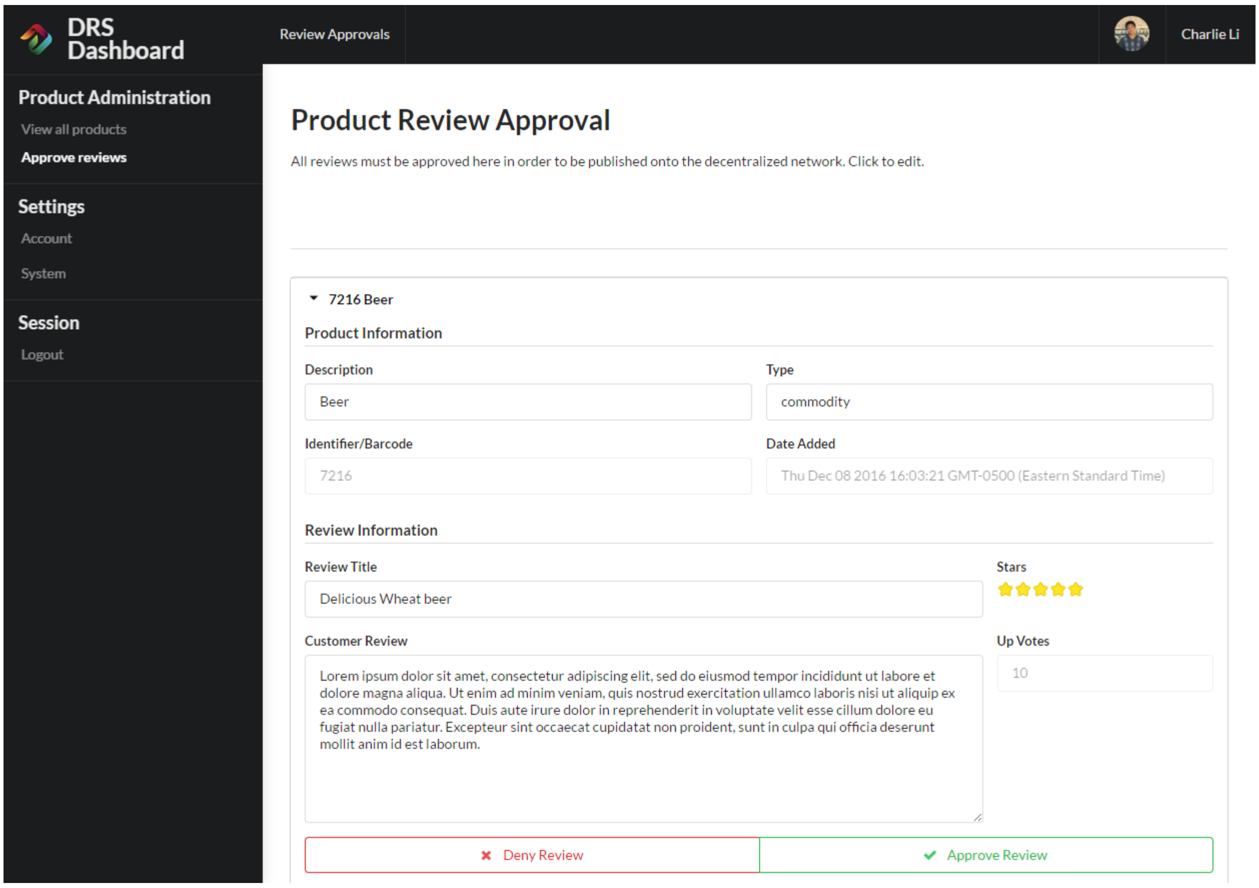
Stars:

Customer Review: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Up Votes: 10

Figure 12: Review search result

6.2.3 Approving New Reviews



Product Review Approval

All reviews must be approved here in order to be published onto the decentralized network. Click to edit.

▾ 7216 Beer

Product Information

Description: Beer

Type: commodity

Identifier/Barcode: 7216

Date Added: Thu Dec 08 2016 16:03:21 GMT-0500 (Eastern Standard Time)

Review Information

Review Title: Delicious Wheat beer

Stars: ★★★★★

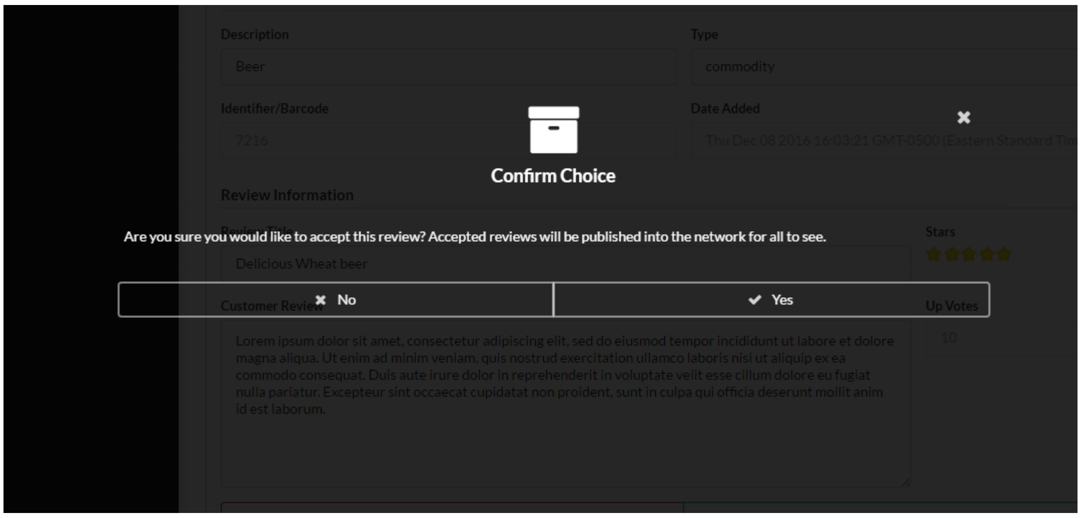
Customer Review: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Up Votes: 10

✖ Deny Review ✔ Approve Review

Figure 13: Review approval process

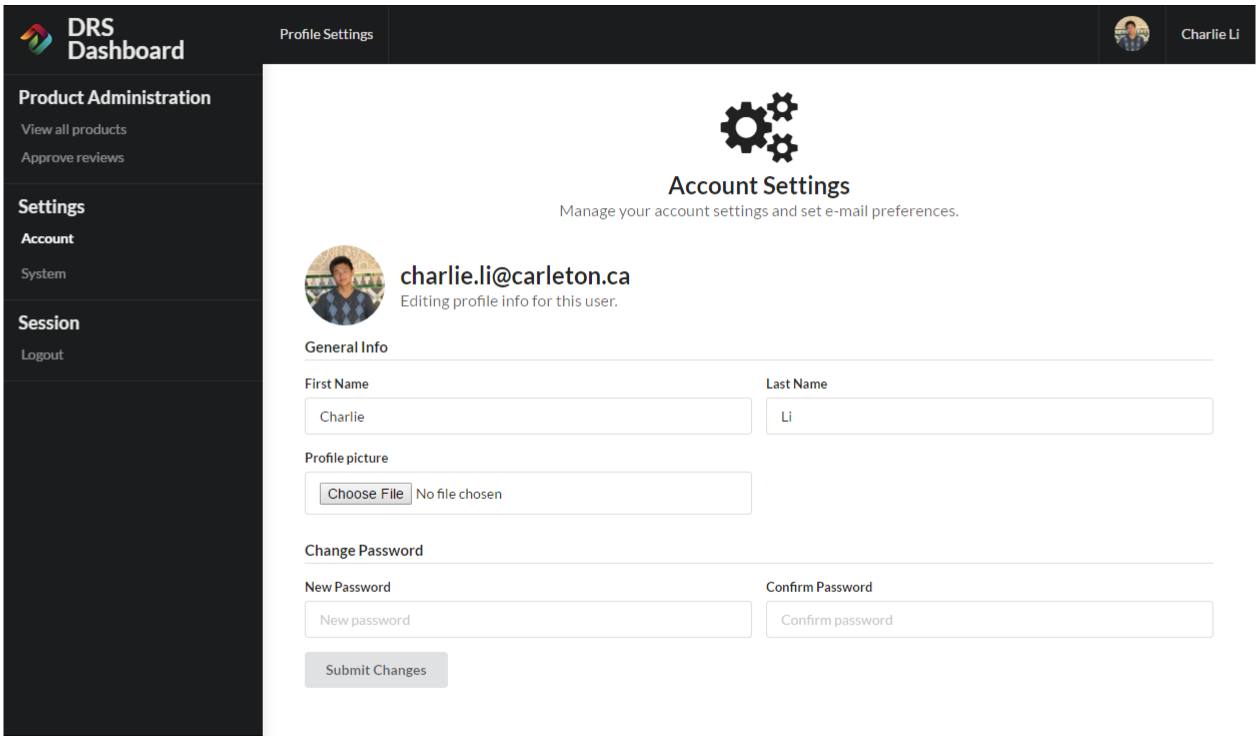
Since the purpose of the dashboard is to take the administrator through the process of approving reviews, all newly submitted reviews that are in the set of tracked product IDs are displayed on this page. Having similar layout as the product view, the difference between them is ability to edit the review before it moves into a published state. Only administrators can edit the title and the formal review text but not the star ratings the review is given. We hope that administrators use this functionality to correct spelling errors before the review is fully submitted to the network since sometimes spam or profanity is submitted on the behalf of bots. The administrator is granted the power of denying the review from reaching the network. The unpublished review is simply discarded. Below, are images of the approval page and the prompts associated to approve or deny a review.



Are you sure you would like to accept this review? Accepted reviews will be published into the network for all to see.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 14: Review approval confirm process



Manage your account settings and set e-mail preferences.

charlie.li@carleton.ca
Editing profile info for this user.

First Name: Charlie
Last Name: Li
Profile picture: No file chosen
New Password: New password
Confirm Password: Confirm password

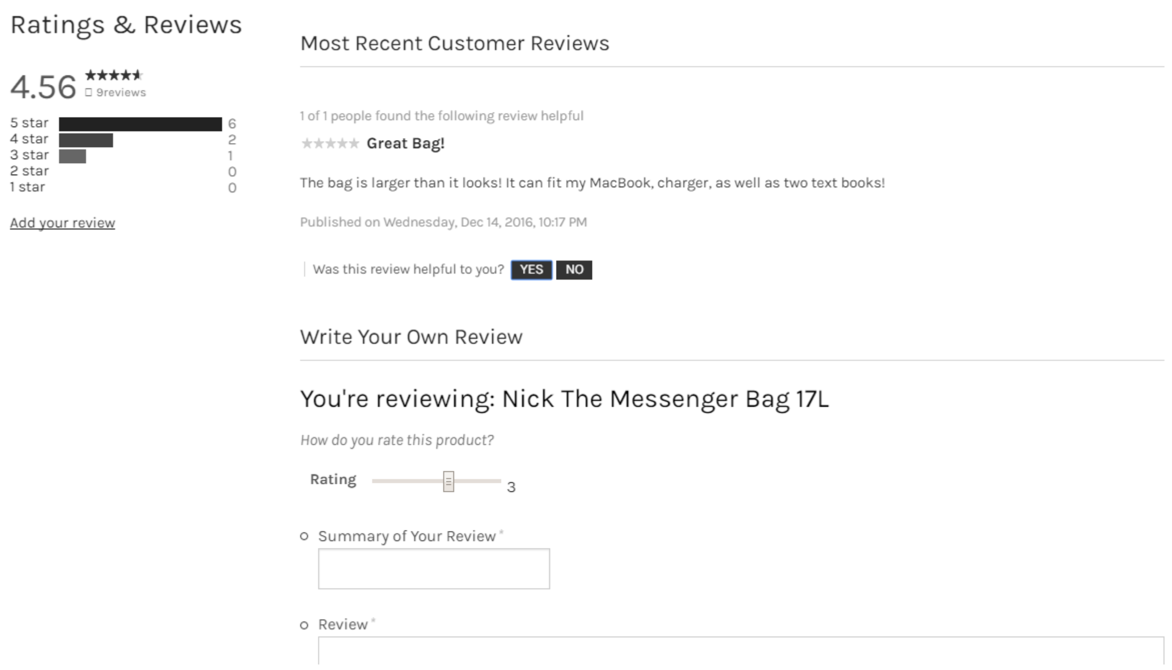
Figure 15: View profile interface

6.2.4 User Profile

Since each admin has a profile of their own, I thought it would be nice to personalize the dashboard for the logged in user. I've allowed some basic human info such as first name and last name to be saved to the server so the dashboard can greet the newly logged in administrator. I've even provided the functionality to attach a profile picture of the admin which becomes displayed on the top of the page. The profile images are saved on the server's static resource folder as mentioned in the implementation section.

Lastly, the profile control allows the logged in user to change their password directly on the dashboard. A screenshot of this page is included above.

6.2.5 Server Side Rendering


Ratings & Reviews

4.56 ★★★★★ 9reviews

5 star	6
4 star	2
3 star	1
2 star	0
1 star	0

Add your review

Most Recent Customer Reviews

1 of 1 people found the following review helpful

★★★★★ Great Bag!

The bag is larger than it looks! It can fit my MacBook, charger, as well as two text books!

Published on Wednesday, Dec 14, 2016, 10:17 PM

Was this review helpful to you? YES NO

Write Your Own Review

You're reviewing: Nick The Messenger Bag 17L

How do you rate this product?

Rating 3

Summary of Your Review *

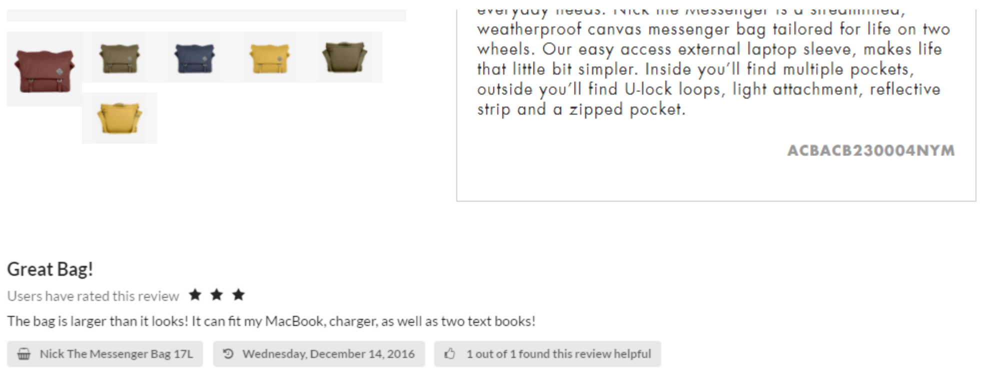
Review *

Figure 16: Server side rendering, look and feel

To showcase the server side rendering functionality, I have built a separate NodeJs application to the serve a static e-commerce site for which I scrapped off the web. I call this application the server side rendering demo (SSRD). The application's main route is programmed to compile an HTML template in DustJs format and respond with HTML. SSRD can make an HTTP call to the backend DRS for reviews using the REST API. With the JSON that's returned, it can inject the results into the template and use the custom CSS of SSRD to define the look and feel of the review section of the page. The demo application is served with Nginx through a reverse proxy to port 443, the only left egress port left on my VM. An image is included that depicts the final HTML, rendered on the browser.

6.2.6 Client Side Rendering

Lastly, I have also build Node app to show the effects of the client side rendering technique. This Node app renders a different static e-commerce site which I also scraped off the web.



weatherproof canvas messenger bag tailored for life on two wheels. Our easy access external laptop sleeve, makes life that little bit simpler. Inside you'll find multiple pockets, outside you'll find U-lock loops, light attachment, reflective strip and a zipped pocket.

ACBACB230004NYM

Great Bag!

Users have rated this review ★ ★ ★

The bag is larger than it looks! It can fit my MacBook, charger, as well as two text books!

Nick The Messenger Bag 17L | Wednesday, December 14, 2016 | 1 out of 1 found this review helpful

Figure 17: Client side rendering, look and feel

I will call this application the client side rendering demo (CSRD. The application's main route serves a static HTML, no templates this time, the magic happens on the DOM. A simple "div" element must be included on the page that looks something like this:

```
<div id="drs-review" data-barcode="861123656411"

style="width:100%;height:500px;"></div>

<script async src="http://drs-server.com/embed.js"
type="text/javascript"></script>
```

In this snippet, we created a section for the reviews to be populated. The width and size of the iframe can be controlled by the outermost "div" element. A script tag is included after to call and evaluate an external script located in on the DRS server. When the script arrives, it will execute by finding the DOM element with ID of "drs-review". It can gather the metadata such as "data-barcode" on this element before making the request for reviews. The response will be injected into the "div" and seamlessly integrated into the web page.

7 Discussion

7.1 Limitations

One of the limitations of the DRS system is the memory hungriness of the back-end. Scaling the system to millions of data items will exemplify a fight between DRS and

Redis on who uses more memory. DRS is designed to cache a large portion of the key space, however, some of these keys are not tracked because they act as the B-Trackers and are redundant backup and query time optimizers. Redis caches the entire tracked set of product reviews database into memory. Based a ballpark of the memory usage based on Redis documentation, a million reviews stored on a node will require 300mb of RAM. DRS can use more since it's running a JVM and can also expect very volatile memory swings since it's also a web server. Therefore, I see memory as the biggest limitation of the system when it scales upwards to millions of reviews on each node.

7.2 Future Work

Although DRS has contained a plentiful set of features, these features were coded quickly with less security in mind. In future iterations, a focus on securing the REST API is scheduled. Since the API opens the ability to interface with the network, the aim is to allow only authorized users to make modifications while tracking their changes so unintended commits can be undone. Aside from the security aspects of the backend, an effort may be launched to provide a more diverse set of reviews that extend the classifications of the commodity genre. Lastly, since DRS was tested and developed in a controlled environment, I would like to see how the system works with real networks and handle traffic from geographically separated regions.

8 Conclusion

DRS has been a success across the board. The project provided a decentralized solution to the data ownership problems experienced by the large data banks that run centralized systems today. DRS capitalized on the scalability, flexibility, and reliability properties of P2P systems by effectively utilizing modern tools, frameworks, and libraries to offer a genuine, novel approach to product review sharing. The system provided an ample solution for smaller e-commerce players to access full on reviews for the commodities they sell through a fast, flexible, and reliable means through a server side or client side rendering approach.

References

[1] Graham Charlton. Ecommerce consumer reviews: why you need them and how to use them. *Econsultancy*, Aug 1970. https://econsultancy.com/blog/9366-ecommerce-consumer-reviews-why-you-need-them-and-how-to-use-them/.

[2] Cisco. Service provider forecasts and trends. http://www.cisco.com/c/en/us/solutions/service-provider/visual-networking-index-vni/index.html.

[3] Christopher Lueg and Danyel Fisher. *From Usenet to CoWebs: interacting with social information spaces*. Springer, 2003.

[4] Navin Nagiah. In the age of big data - data is power. *The Huffington Post*. http://www.huffingtonpost.com/navin-nagiah/in-the-age-of-big-data-da_b_3279565.html.

[5] T. Manfredi. Gnutella protocol development. http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html, journal=Gnutella Protocol Development.

[6] Indranil Gupta. 3.7 pastry p2p system. *Youtube*. https://www.youtube.com/watch?v=iPVAOh_slsU.

[7] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Middleware 2001 Lecture Notes in Computer Science*, page 329–350, 2001.

[8] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. *Peer-to-Peer Systems Lecture Notes in Computer Science*, page 53–65, 2002.

[9] David Easley and Jon Kleinberg. The small-world phenomenon. *Networks, Crowds, and Markets*, page 537–566.

[10] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51, Jan 2002.

[11] Fabio V. Hecht, Thomas Bocek, and Burkhard Stiller. B-tracker: Improving load balancing and efficiency in distributed p2p trackers. *2011 IEEE International Conference on Peer-to-Peer Computing*, 2011.

[12] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, Jan 1970.

[13] Vassilis G. Papanicolaou, George E. Kokolakis, and Shahar Boneh. Asymptotics for the random coupon collector problem. *Journal of Computational and Applied Mathematics*, 93(2):95–105, 1998.

[14] Fielding Roy Thomas. Chapter 5: Representational state transfer (rest). *Architectural Styles and the Design of Network-based Software Architectures (Ph.D.)*, page Chapter 5, 200.

[15] markoJs. Templating benchmarks. *GitHub*, Nov 2016.

[16] Tony Freed. What is a virtual dom? *Medium*, June 2011.

[17] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

A Review Model

This section will take the chance to first describe what a review is composed of in the system. Figure X shows the object hierarchy diagram of a review, which can assume more specialized super classes by inheriting from the `BaseReview` class. Laying out the model this way can help developers easily extend the system to incorporate more specialized reviews such as service reviews, manufacturer reviews, restaurant reviews, hotel reviews, or attraction reviews. Here is the formal specification of the `BaseReview`. The review model is serialized into JSON by DRS and provided as outputs of the REST API. Feel free to browse through this section briefly as the later sections will touch on some components of this model.

Constructors

`BaseReview()` - Default constructor. Will make empty object

`BaseReview(String content, int stars)` - Titleless review constructor, will cascade to last constructor with default empty values.

`BaseReview(String content, String title, int stars)` - The constructor that implementors should interface with. Use this constructor to properly initialize a review object. This constructor will cascade into the one below to initialize vote to zero.

`BaseReview(String content, String title, int stars, int votes)` - This constructor initializes all values to default and also generates a timestamp for when the review was created.

Properties

Many properties have human-readable aliases to be associated with the JSON serialized object to provide the end user.

`+ m_productName: String` - alias “description” - optional - Describes the product name that the review associated to

`+ m_generalReviewType: constant default String` - alias “type” - mandatory - Describes the model type, useful for mapping to a specific Ember model.

`+ m_dhtAbsoluteKey : String` - alias "id" - mandatory - Represents the uniqueness of each review. Cannot be used to map to a product.

`+ m_content: String` - alias "review_content" - mandatory - Represents the unbounded review of the product. This is a mandatory field and is enforced upon submission by pre-processing validation.

`+ m_title: String` - alias "title" - mandatory - Represents the title of the review. This is a mandatory field and is enforced upon submission by pre-processing validation.

`+ m_createTime: long` - alias "created_at" - mandatory - Represents the timestamp of when the review was created based on UNIX epoch time. This field is autogenerated on each view submission.

`+ m_stars: int` - alias "stars" - optional - Represents the number of stars the review has received. Stars are used to judge how the product rates on a five-point scale.

`+ m_upvotes: int` - alias "upvotes" - optional - Represent the numbers viewers who found this review as helpful.

`+ m_downvotes: int` - alias "downvotes" - optional - Represents the number of viewers who found this review unhelpful.

`- m_contentId: Number160` - alias "content_id" - mandatory - Represents an SHA1 hash of the review string "m_content" used to identify unique review text and serves at one of four parts of the DHT key.

`- m_locationId: Number160` - alias "location_id" - mandatory - Represents an SHA1 hash of the product identifier used to identify unique entities in which the review is for and serves at one of four parts of the DHT key. Location ID is the first 40 bits of the 160 and helps save reviews of the same product near the same locality.

`- m_domainId: Number160` - alias "domain_id" - mandatory - Represents an SHA1 hash of the state of the review, either published or in approval.

`- m_dhtKey: Number160` - mandatory - Represents the full 160-bit element key for Kademlia

+ `m_publishTime: long` - alias "published_at" - mandatory - Represents the UNIX epoch time for when the review was published into the main DRS network.

Methods

`fillInIds(Number160 loc, Number160 con, Number160 dom, Number640 absoluteKey): void` - Privately called when saving the identifications of the record to the object.

`abstract getIdentifier(): String` - All subclasses must implement this method to return a representation of the entity identification key. For example, this method will return a barcode string if the product type is a commodity.

`abstract getType() : String` - All subclasses must returns the type of product for review. For example, if the review if for an item, then the type would return commodity.

+ `identity() : BaseReview` - Identity function

+ `getModelId() : String` - Returns the unique identifier of the review, a hash of the review content.

+ `validate() : boolean` - Returns true or false if the review passes validation. All subclasses must implement this function to validate the subclass before calling the superclass's validate method.

B Account Model

Constructors

`BaseAccount()` - Default constructor. Will make an empty object.

`BaseReview(String email, String password)` - Creates a new

Properties

+ `m_userId : String` - alias "user_id" - optional - Describes the user identity

+ `m_email : constant default String` - alias "email" - mandatory - Describes the user email

+ `m_firstName : String` - alias "fname" - optional - Describes the user's first name

`+ m_lastName : String` - alias “lname” - optional - Describes the user’s last name

`+ m_password : String` - alias “password” - mandatory - Describes the person’s password (hashed)

`+ m_profilePic : constant default String` - alias “profile” - optional - Describes URL of the profile picture asset

`+ m_loginToken : String` - alias “token” - optional - The last login token that was generated

`+ m_prevToken : List<String>` - alias “tokens” - optional - Represents the last 10 previous login tokens used and generated. Max 10 concurrent sessions.

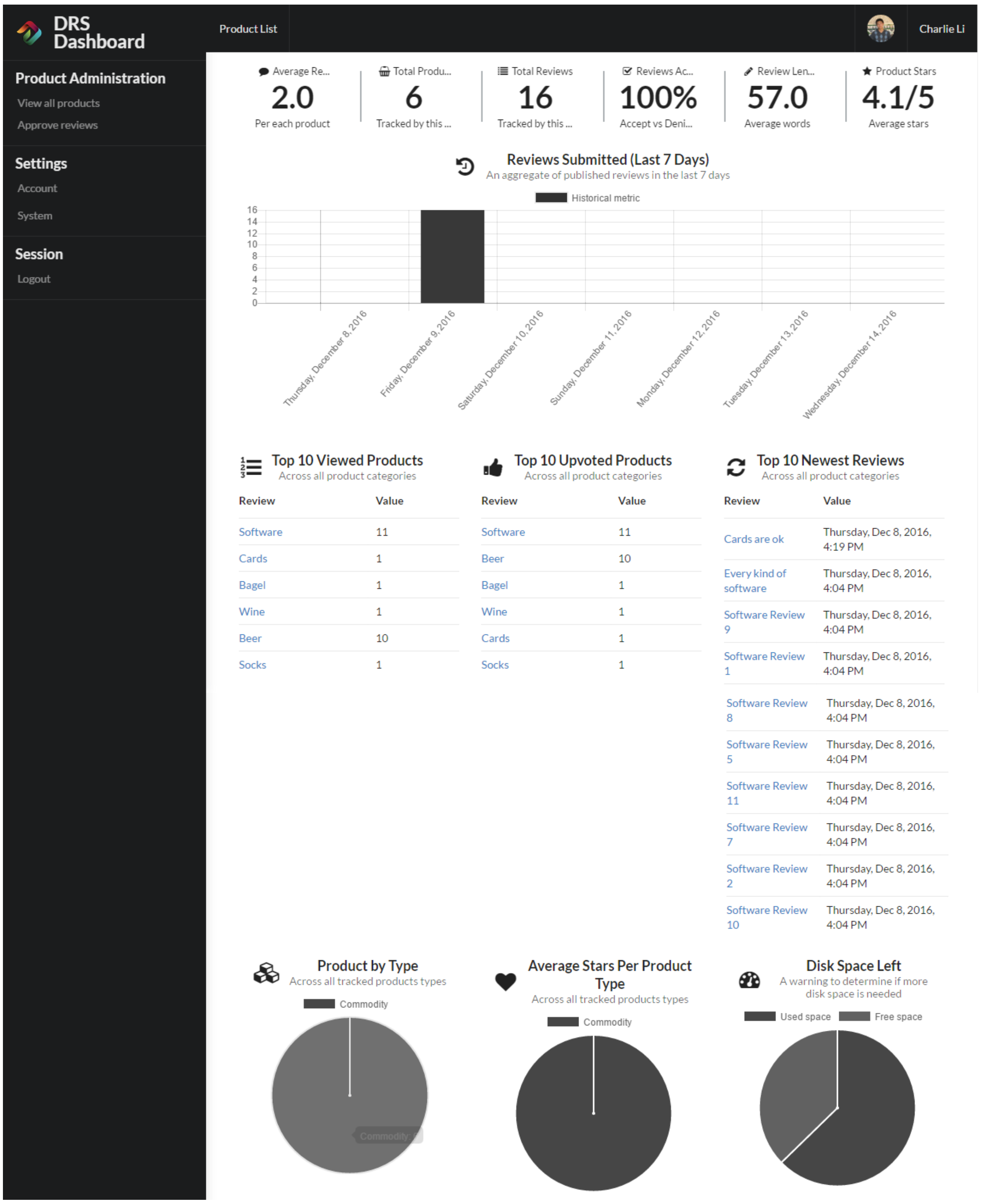
Methods

`+ addToken(token: long): void` - Add a session token to the account, with the max as 10

`+ hasToken() : boolean` - Checks if there are previous login tokens

`+ validate() : boolean` - Returns true or false if the review passes validation. All subclasses must implement this function to validate the subclass before calling the superclass’s validate method.

C Full Dashboard Screenshot view



DRS Dashboard

Product List

Charlie Li

Product Administration
- View all products
- Approve reviews

Settings
- Account
- System

Session
- Logout

Average Re...	Total Produ...	Total Reviews	Reviews Ac...	Review Len...	Product Stars
2.0	6	16	100%	57.0	4.1/5
Per each product	Tracked by this ...	Tracked by this ...	Accept vs Deni...	Average words	Average stars

Reviews Submitted (Last 7 Days)
An aggregate of published reviews in the last 7 days

Historical metric
16
14
12
10
8
6
4
2
0
Thursday, December 8, 2016
Friday, December 9, 2016
Saturday, December 10, 2016
Sunday, December 11, 2016
Monday, December 12, 2016
Tuesday, December 13, 2016
Wednesday, December 14, 2016

Top 10 Viewed Products
Across all product categories

Review	Value
Software	11
Cards	1
Bagel	1
Wine	1
Beer	10
Socks	1

Top 10 Upvoted Products
Across all product categories

Review	Value
Software	11
Beer	10
Bagel	1
Wine	1
Cards	1
Socks	1

Top 10 Newest Reviews
Across all product categories

Review	Value
Cards are ok	Thursday, Dec 8, 2016, 4:19 PM
Every kind of software	Thursday, Dec 8, 2016, 4:04 PM
Software Review 9	Thursday, Dec 8, 2016, 4:04 PM
Software Review 1	Thursday, Dec 8, 2016, 4:04 PM
Software Review 8	Thursday, Dec 8, 2016, 4:04 PM
Software Review 5	Thursday, Dec 8, 2016, 4:04 PM
Software Review 11	Thursday, Dec 8, 2016, 4:04 PM
Software Review 7	Thursday, Dec 8, 2016, 4:04 PM
Software Review 2	Thursday, Dec 8, 2016, 4:04 PM
Software Review 10	Thursday, Dec 8, 2016, 4:04 PM

Product by Type
Across all tracked products types

Average Stars Per Product Type
Across all tracked products types

Disk Space Left
A warning to determine if more disk space is needed

Figure 18: Full dashboard screenshot view