OPENBAZAAR: AN ONLINE PEER-TO-PEER MARKET

Kevin Chen '15 Aaron Taylor '16

1 Introduction

In October 2013, the FBI was able to shut down the Silk Road, an online black market, with the seizure of a single server. To circumvent the problem of a single point of failure, the winners of a hackathon recently unveiled a proof-of-concept for a peer-to-peer version of the Silk Road.

Known as OPENBAZAAR (formerly the Dark-Market), this service incorporates many of the same user experience paradigms as its predecessor: private communication between buyers and sellers, HTML pages to view sellers' wares, a reputation system for ratings and reviews, and an arbiter-escrow feature that ensures a "fair" outcome when deals go sour. Each piece of the system is covered with layers of encryption, based on the bitcoin protocols and elliptic curve cryptography, to ensure that all communications are secure, all transactions are properly mediated, and all identities are correctly verified. Still nascent, OPENBAZAAR has yet to implement several features that would enable the service to enter production mode and be delivered to users.

As the current system currently stands, in order for a seller to maintain a page, they basically need to maintain an active server with a fixed IP address that other users of the network can connect to in order to view their product offerings and initiate transactions. This is less than ideal for a number of reasons.

What has led to the sucess of other online marketplace systems – such as eBay, Craigslist, and even the Silk Road – is the ease with which users can post items for sale and order items from other sellers. The current system has a large amount of setup due to the direct peer-to-peer nature of the system.

We want to emulate the convenience and simplicity of an environment hosted on a central server within the peer-to-peer network. We work on achieving this by replicating user sites across the network using a distributed hash table (DHT).

In order for this system to work, the data for a seller's marketplace and ratings has to satisfy the following conditions: it must replicated across several nodes, it must maintain its integrity and be mutable only by the seller/publisher, and it must be quickly accessible by the users of the system.

We intend to implement a system of replication that brings two main advantages. First, it allows users the convenience of not having to keep their server up constantly if they want buyers to see their wares. Second, it increases availability: in the event that a seller's machine or otherwise needs to go offline, his or her marketplace is being hosted by other nodes in the P2P network.

Replication also has two downsides. First, we must ensure that when Bob requests data for Alice's market, he receives the most up-to-date version of her market. We address the challenge of synchronization later. Second, replication disincentivizes users from maintaining their servers. That is, what is to stop Bob from powering down or disconnecting his machine with great frequency if Alice is hosting his marketplace for him? We address this problem by giving users with greater hosting uptime in the network certain perks and community incentives, such as paying smaller commissions to the third party arbiter during transactions, and positive or negative indicators in the reviews section attached to each user.

2 How it works

A user downloads the market software, which runs a daemon in the background and allows a user to become a node in a distributed network with a P2P library known as ZeroMQ, or ØMQ.

For Alice to become a seller, she can edit an HTML file designated as her seller page. Then a buyer like Bob can browse the market, which he runs via web browser. Bob can search for a user's nickname or click on another user's node.[1]

If Bob sees something he wants to buy, he sends a message to Alice. If Alice wishes to engage in trade, an arbiter must be selected to settle disputes in the event that the deal falls through. An arbiter may be selected from overlap between Alice's and Bob's list of approved arbiters, or randomly if their lists don't intersect.

Having selected an arbiter, the system creates a new "multisignature" Bitcoin address from the three users' public encryption keys. During the duration of the transaction, this Bitcoin address holds the buyer's money in escrow. The transaction is complete when two out of the three parties agree on where the money should go. If everything goes smoothly and the product is shipped to the buyer, both buyer and seller sign a transaction to move the bitcoins out of escrow and into the seller's account. If the product is defective or never ships, the arbiter must step in.

Finally, each participant rates all other participants. Ratings are cryptographically signed with the

[1]At the moment, there is no functionality for searching for user nicknames, and there is a lack of anonymity since Bob must click on Alice's bare IP address if he wishes to view her wares.

rater's private key, which prevents users from forging reviews. Sellers with better ratings will naturally attract more buyers.

3 Architectural Overview

The OPENBAZAAR repository is incipient but constantly being updated. Currently, the only supported features are the ability to connect to the distributed marketplace and the ability to view your market in the browser; transactions, ratings, and arbitrations are not possible. Furthermore, the data for the marketplace is stored persistently with a user-created MongoDB database. However, we would like to ensure the availability of every user's marketplace by spreading out data. This is done with a distributed hash table (DHT), which provides fault tolerance, scalability, and decentralized autonomy.

3.1 What is a DHT?

A DHT is just a distributed dictionary; that is, hash table buckets are nodes across a network. Just like any other lookup service, it maps keys to values. In this case, the keys are hashed files or hashed file locations, and the values are the files themselves. That is, given some key $k = hash(filename)$, the DHT should be able to figure out which node hosts *filename*, thereby allowing other nodes to request that specific file. This is shown in Figure 1.

So how do we map $hash(filename)$ to an actual node? In most DHTs, each node has a 160-bit identifier that determines which potential keys each node owns.[2] Specifically, a node n owns a key k if n is the closest node to k. "Nearness" is computed by a distance function $\delta(k, n)$, which varies depending on the DHT.

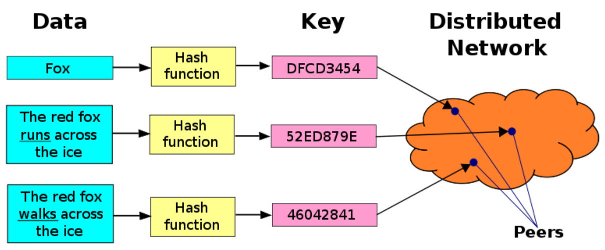

Figure 1:

When deploying a distance function, one of the main considerations is load balancing. Ideally, the keyspace is partitioned evenly between all the nodes in the network, so one node doesn't store more keys than another. If this is not the case, then that node can get bogged down, and the entire system will suffer in performance.

This form of consistent hashing ensures that when a node is added or removed from a system, only the keys owned by adjacent nodes are changed. When a node joins the system, it takes responsibility for some of its neighbor's keys. When a node crashes or leaves the system, the keys mapping to the data it held must be allocated to new nodes. Moreover, the more nodes are in a system, the fewer

[2]Keys and node IDs are 160 bits for consistency with the SHA-1 hash function. Nothing stops us from using a different hash function.

keys have to be remapped on every node entry or exit. This is yet another advantage a DHT holds over a traditional hash table, which remaps its entire keyspace when a bucket is added or removed.

3.2 Why Kademlia?

3.2.1 Unique Features

There are many different kinds of DHTs out there. They differ in the size of their address space (128- or 160-bit keys?), in the type of hash function is used (SHA-1, SHA-2, etc.), in what gets hashed to produce keys (file names, file contents, etc.), in redundancy and reliability (nodes can dynamically agree to store the same key, just in case one them crashes), in the way traffic is routed, and in many other ways.

We opt for the KADEMLIA DHT, which has several unique features. First, it requires minimal configuration. When a node wants to join a network, little else is required of it than to perform a self-lookup. By performing a query, the requester becomes aware of all the nodes that exist between it and the request's intended recipient.

Second, this frequently updated awareness affords nodes the ability to route their queries through low-latency paths. And although knowledge spreads like the plague, there is a natural defense against denial of service attacks, since a node can only know about a limited number of other nodes.

Third, not only can KADEMLIA nodes select low-latency routes, but they can also send parallel asynchronous queries to avoid timeout delays.

Fourth, KADEMLIA features unidirectional routing, as opposed to bidirectional routing. All node lookups and value lookups are guaranteed to converge on the same path, regardless of the requester. Thus, caching $\langle$key,value$\rangle$ pairs along a path will reduce latency. Unidirectionality is a consequence of using a symmetric distance function, namely *bitwise exclusive or* (XOR). That is, the distance function is denoted by $\delta(x, y) = x \oplus y$, and its property of symmetry is denoted by $\delta(x, y) = \delta(y, x)$.[3]

3.2.2 Node State

Each KADEMLIA node stores a list for every bit in the address space, in this 160. Think of these lists as a node's contacts or peers, and each element in the list stores a peer's IP, port, and node ID. For $0 \leq i < 160$, list i stores peers whose distance falls within the interval $[2^i, 2^{i+1})$. That is, the 0^{th} list stores peers that are 1 away; the 1^{st} list, peers that are 2 or 3 away; the 2^{nd} list, peers that are 4, 5, 6, or 7 away; and so on.

Figure 2 shows the general structure of the network. Nodes are viewed as leaves in a binary tree. The gray ovals form the peer lists for node 6 (colored in black). The 2^{nd} peer list exclude node 3,

[3]In KADEMLIA, the distance between two nodes is computed by taking the XOR of their two node IDs.

which is not in the network.

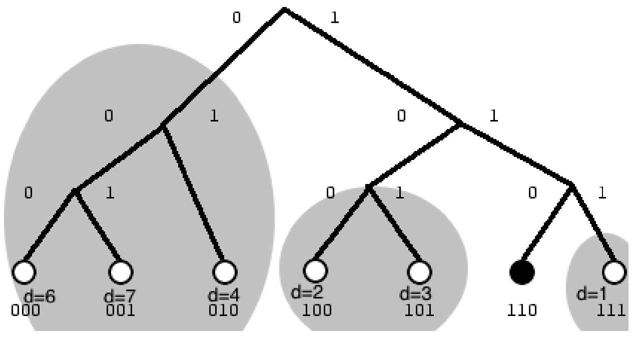

Figure 2: Nodes are leaves. k-buckets are increasingly large neighborhoods/subtrees.

These lists are known as k-buckets. The peers inside each k-bucket are ordered by how recently they were seen, with the most recent encounters at the tail. For small values of i, the k-buckets are probably going to be empty. That is, the 0^{th} k-bucket, which stores the peer that is one away, is almost certainly going to be empty; this is because the keyspace is so large that we will rarely see $k_1 \oplus k_2 = 1$. In layman terms, nodes are going to be spread out, so the k-buckets that can store a wider range of nodes are more likely to be filled.

However, there is a limit to how large a k-bucket can get: that limit is determined by a system-wide parameter, k. Every time a node receives a request or reply, it updates its k-bucket for the sender. If the sender is already in the appropriate k-bucket, it moves to the tail of the list. If there are fewer than k entries in the bucket, the sender moves to the tail as well. If the bucket is full, then the recipient node pings the least-recently seen node at the head of the list and waits for a response; if no response comes, the sender node is inserted at the list tail; but if the least-recently seen node responds, then it moves to the list tail and the sender node is discarded.

This least-recently seen eviction policy has two main advantages. First, it is a natural defense against flooding attacks, like DDoS, since the k-buckets are capped in size. Second, the preference for old contacts has been empirically shown to have availability benefits: the longer a node has been up, the more likely it is to stay up for another hour.

3.2.3 Protocol

The KADEMLIA protocol is comprised of four main RPC calls: PING, STORE, FIND_NODE, and FIND_VALUE.

The PING RPC checks to see if a node is online and, as previously mentioned, is used when a k-bucket is full and a node wants the status of the least-recently seen peer in that bucket.

STORE tells a node to store a $\langle$key,value$\rangle$ pair.

The FIND_NODE RPC takes a 160-bit node ID as an argument and returns the k nodes that are closest to the target ID. FIND_VALUE works much like FIND_NODE, except the recipient of a FIND_VALUE RPC will return a stored value if it has received a STORE RPC for the key.

Underlying these RPCs is the most important and most complex operation in KADEMLIA: node lookup. This entails locating the k nodes closest to a given node ID. For instance, node lookup is used in the STORE RPC for replication: when a $\langle$key,value$\rangle$ pair is stored at a particular target node, it is also stored at the k closest peers of that target node.

Node lookup works in the following way: The lookup initiator starts by picking $\alpha = 3$ nodes from its closest non-empty k-bucket (remember that the closest k-buckets will probably be empty) and then sends parallel, asynchronous FIND_NODE RPCs to the α nodes. Following those FIND_NODE RPCs, the initiator will become privy to new nodes and can query those. If a round of queries does not return a single node closer to the target node, then the initiator queries nodes that have not already been queried. The lookup ends when it can find nothing closer and when the k closest nodes have all responded.

More importantly, the KADEMLIA protocol has replication built into its DNA. First, nodes must republish their $\langle$key,value$\rangle$ pairs on the hour. Second, entries will expire after 24 hours to limit stale data. Finally, when node a encounters another node b that is closer to some of a's $\langle$key,value$\rangle$ pairs, a replicates them over to b. In short, the marketplace is kept fresh and up to date.

4 Entangled API

We use ENTANGLED, a DHT written in Python that builds off of KADEMLIA.

The main object we use is ENTANGLEDNODE, which is basically a KADEMLIA node but with a few more non-standard RPCs, such as DELETE.

KADEMLIA nodes feature the four RPCs – PING, STORE, FIND_VALUE, and FIND_NODE – as well as methods to join the network and to republish and expire data. Each KADEMLIA node stores a routing table, which includes methods to add, remove, and find node contacts. Additionally, the routing table has a method to find all the k-buckets that need refreshing and a method to update the "last accessed" timestamp of the k-bucket that covers the range containing a specified key argument.

5 Design Challenges

5.1 Replication

How much replication is enough data? Suppose there are 100 users in the OPENBAZAAR network. There must be some $n < 100$ for which if we replicate Alice's data onto n other nodes, we can ensure with sufficiently high probability that Alice's marketplace will be "virtually" always be available.

[1].

Ensuring that there is adequate replication of site data between the nodes in the network is crucial to maintaining the sites as persistent state. It would be unacceptable to have a system that does not preclude the possibility of your site dissapearing with a few node failures. Fortunately, KADEMLIA itself is designed to handle this problem. It uses these mechanisms to ensure that, once published, data is maintained in the network in all but the most extreme cases. The ENTANGLED implementation of KADEMLIA mirrors this functionality, as is demonstrated provided by the example programs included with that project.

5.2 Synchronization

Another issue is

6 Implementation

Our implementation of this system began with reading through the source code and limited documentation of the various pieces of what was to be in our final system and understanding how each of the pieces worked and interacted. Running the actual server for the nodes is an implementation called TORNADO, that handles the creation of a webserver and the majority of the transport layer. The OPENBAZAAR P2P interactions run on a network library called ZeroMQ (ØMQ) that handles most of the peer-to-peer setup and network maintenance between the nodes in the system. The system is designed fairly sensibly, with the various cryptographic components separated out into seperate modules from the core market interactions. The website itself is accessed through a browser, typically at LOCALHOST:8888, and is a fairly straightforward HTML and javascript site that allows for updates to your own market and provides a GUI for interaction with other sites in the network.

6.1 Current Implementation

The current implementation is evolving rapidly with contributions from the open-source community, but the basic structure of the codebase has remained relatively the same. The majority of the market interactions are implemented in the node folder. Files within here create the various networking components, including event handlers and protocol implementations, that allow the market to function on its lowest layer.

6.2 Additions

created launch.py script to create a command-line interface. modified tornadoloop.py to handle creation of the ENTANGLED node, modified the implementation of the main eventloop to use both zeromq and ENTANGLED modified market.py to replace peer-to-peer market functionality with called to the ENTANGLED node and network on both the sending and recieving end. some code deemed unnceessary due to this change

6.3 Issues

6.3.1 Understanding the codebase

The first major issue we had was understanding a large, growing, poorly documented OPENBAZAAR codebase. Even as we were reading, new commits were being made to the repository on a daily basis. Reading every line of code would have been a hassle, so we first determined what each high-level module did, and then delved into the most important folders. We avoided in-depth readings of the low-level cryptographic implementations, focusing just on the method headers, and worked towards a summary understanding of the peer-to-peer interactions of the existing system. We spent most of our time on the market interactions where we would be implementing our desired functionality.

In this manner, we were able to make sense of a codebase tens of thousands of lines long to implement our own functionality within it. The methodology we found useful was finding a starting point, in our case the launch point for the application, and then following the method calls and initializations to understand the connections between the various pieces of the program.

6.3.2 Conflicting Systems

Once we had gotten over the hurdle of understanding how the various components of the existing codebase functioned, figuring out how to utilize the ENTANGLED APIs to create the desired functionality was not too difficult. The function calls created straightforward ways to publish and remove data from the network, and retrieve it as desired. However, out of the box the ENTANGLED implementation used a different networking engine than the rest of the OPENBAZAAR project, which created frustrating conflicts in getting that network up and running.

The root of our main problem was that two network event listening loops cannot both be running in the main thread. The OPENBAZAAR project was set up to be running a loop from the TORNADO library, which handled all its incoming requests and outgoing messages as it should. The ENTANGLED nodes relied on an instance of a module called a REACTOR from the TWISTED python library in order to handle its network interactions.

There is little information in the ENTANGLED documentation about what components of the TWISTED library are required to be running for the ENTANGLED nodes to begin functioning. In the example programs provided with the ENTANGLED source code, the only visible use of the TWISTED library is unrelated to the DHT, and involves the direct connections in the application-specific function of filesharing. Initially, I had thought that the REACTOR was unnecessary for just running the DHT, which lead me to eliminate all other possible sources of issues until I was lead back to the same line I had commented out earlier as the only deviation in my implementation. In hindsight, it makes sense that the nodes need an event listener, and overlooking that fact was due to both my relative inexperience in python and a lack of specificity in the

ENTANGLED documentation.

However, that merely brought me back to the same issue that had lead to the removal of that line of code in the first place. My initial plan was to run one of the event loops in the seperate thread. However, because of the way signals from the operating system are handled in python, both of the loops demand to be run in the main thread, and produce a variety of error messages to this effect if you try to implement them in a manner otherwise. Additionally, due to the interconnected nature of the ENTANGLED node and the market node, running one in a seperate process would just have made a bad situation worse, with extra layers of complexity necessitated for inter-process communication.

6.3.3 Conflict Resolution

As somewhat of a logjam, I dove into the sourcecode for the TORNADO and TWISTED libraries, looking for any commonality in the event loops that could be exploited. As it turns out, we were not the first to have this problem, and buried within the configuration options for TORNADO were two options for bridges between the TWISTED REACTOR event loop, and the TORNADO IOloop. One of these replaced the builtin TORNADO IOloop with the twisted REACTOR and reimplemented the TORNADO IOloop API onto of the resulting TWISTED REACTOR. This allowed for the ENTANGLED nodes to interact with the underlying REACTOR, and for the rest of the existing application to interface with the TORNADO API built ontop of it.

Happily, this solution worked, allowing the ENTANGLED nodes to connect to one another and effectively combining the two seperate nodes in the previous conception of the system into a single one, running on the same event loop in the same thread.

After this solution was found, a few details remained to be ironed out with out untested implementation of the ENTANGLED node interactions through API calls. ENTANGLED encodes the data passed into the DHT...

7 Evaluation

Using evaluation tools in launch.py

pretty graphs...

8 Outside Work

Interestingly, while we have been implementing our fork of this project, the person who has taken over primary development of project on GitHub has begun some limited work on implementing a version of the KADEMLIA DHT. The branch on GitHub is fairly limited, currently containing an implementation of the KADEMLIA Datastore and a few placeholders for where future code may go, but the similarity in our ideas is interesting.

9 Conclusion

Over the course of this work, we learned several lessons. First, working with large codebases is suprisingly approachable after a steep initial learning curve. Second, using libraries can be powerful but treacherous, as we saw with our conflict between ENTANGLED and TORNADO. Third, a DHT with a good API can provide very useful functionality in a peer-to-peer system. Fourth, our efforts have been validated by similar work in the open-source project. Fifth, we are super hackers, even more so now than before. Lastly, a huge thanks to Jeannie – you're the best.

References

[1] Edith Cohen, and Scott Shenker, *Replication strategies in unstructured peer-to-peer networks*. New York, NY, 2002.