Using Chord as a Platform for Software as a Service

Andrew Rosen Brendan Benshoof Matt Erwin Anu Bourgeois
Department of Computer Science, Georgia State University
34 Peachtree St NW
Atlanta, Georgia 30303
rosen@cs.gsu.edu

***Abstract*—Distributed Hash networks, such as Chord, are an established technique for storing data in a distributed fashion. In Chord, nodes are evenly distributed and are responsible for files based on their position in the overlay network. From Chord, we introduce the concept of the Hashnet, where computers use distributed hash table as a means of distributing both data and work. Because Chord provides the backbone of this network, a Hashnet is robust, scalable, resilient to churn, fault tolerant, and automatically and evenly distributes work throughout the network.**

WHAT'S GOING ON IN THE PAPER

- The subsubsections are being combined as I write stuff
- The Introduction and related work are the sketchiest areas, as the old paper isn't really relevent to this

I. Introduction

Peer-to-peer (P2P) networks are of enormous interest to network research, as they provide an efficient way to distribute large files to millions of users without many of the limitations of the traditional client-server model. In the client-server model, the server addresses each client's request individually, creating a single point of failure due to traffic or the loss of the server. In peer-to-peer networks, each node takes on both the role of client and server. If someone wishes to introduce data into the a peer-to-peer network, peers can both request the file and distribute it to others. This takes much of the burden of file distribution off of what would have normally been an extremely busy server [18].

P2P networks have resulted in an unprecedented amount of data being purveyed through the Internet. While recent studies show that P2P accounts for 13.2% of downstream traffic, down from its peak of 40% of all Internet traffic, the actual volume of P2P traffic is at its highest yet [12]. Many companies have embraced legal uses of P2P technology, such as providing ways to stream media or provide large updates [14]. For example, Blizzard uses a modified BitTorrent protocol to distribute updates and patches to millions of users for games such as World of Warcraft and Starcraft II [4]. ¡—— PROBABLY GOING TO GET RID OF THESE PARAGRAPHS

However, the architectures that provide the backbone of many a P2P network, such as Chord [17] and Kademlia [11], have been viewed solely as a means for sharing files or routing traffic. We have developed a system, called CHRONUS, that takes the idea of evenly distributing files and abstracted it. CHRONUS is capable of evenly distributing both data and work

1) Current models of distributing work:

2) Our proposed model Hashnet: A system of computers connected together using a distributed hash table as a means for efficiently distributing both data and work.: A hashnet is not a new paradigm for distributing work; it is an extension of the established and highly tested means of distributing files and applying the scheme to do the same for work.

II. Background

A. Old Background

Not all peer-to-peer networks are equal; there are variety of protocols and methodologies that a networks could implement and this affects what kind of solutions are available to reduce the traffic on the network [8] [18].

The most basic type of network is a structured, centralized network. Peers in this network communicate with a central server to provide their files and to locate other peers that have the files they are searching for. This is structured in the sense that the layout of the overlay network is tightly controlled, in this case by the server(s). While this avoids the problems of routing, it has the same issues of scalability as a client-server layout and is not much of an improvement. An example of this network is the long defunct Napster [8] [18].

On the other side of the spectrum, there are unstructured, decentralized networks. These networks create overlay links between nodes in a random manner. This leads to a very unstructured overlay, but it is one that is very easily constructed. No single node is responsible for the whole of the network; files are located by sending out requests to neighboring peers, which in turn request from their neighbors and so on. Should a file become suddenly popular, the flood of traffic generated by requests can easily bring some peers to their knees [9], making them a poor choice for implementation [18].

Modern P2P implementations are hard to classify easily, due to the variety of methods used to create a working network that avoids the weaknesses of the two above network types. Many networks today use a decentralized structured approach to distribute files, where the topology of the overlay is constructed and controlled by the protocol and the information about the network is distributed among the peers. This distribution is typically accomplished by a distributed hash table (DHT). Networks that use a DHT choose specific peers in which to place information about how to find particular files or data. These peers are chosen so that the peer's ID in the network

corresponds to the file or data's ID, typically by hashing both ID's and comparing them [8].

In addition, the network topology is distributed among various peers. Each peer has a table consisting of other peers in the network and the means of communicating with them. The contents of this table are also controlled by the protocol. The table handles the routing of requests from one node to another; when a peer receives a request it cannot fulfill, such as information about where to find a particular file, it directs the the request to the node that is "closest" to the destination of the request. How this works is determined by each protocol [17]. It should be noted that closeness is relative to the algorithm; depending on what identifiers are assigned, a node in New York City might be "close" to a node in Russia, but "far" from a node physically a few miles away. Examples of protocols that use these techniques are trackerless BitTorrent [7], Chord [17], and Kademlia [11]. As our work is implemented using Chord, our implementation of the protocol is covered in the following section.

B. CHORD

The Chord protocol [17] takes in some key and returns the identity (ID) of the node responsible for that key. These keys are generated by hashing a value of the node, such as the IP address, or by hashing the filename of a file. The hashing process creates a m-bit hash identifier.

The nodes are then arranged in a ring from the lowest hash-value to highest. Chord then takes the hashed files and places each in the node that has the same hashed identifier as it. If no such node exists, the node with the first identifier that follows this value. This node responsible for the key κ is called the $successor$ of κ, or $successor(\kappa)$. Since the overlay is a circle, this assignment is computed in module 2^m space. For example, if there were some portion of the network with nodes 20, 25, and 27, node 25 could be responsible for the files with the keys (21,22,23,24,25). If node 25 were to decide to leave the network, it would inform node 27, who would then be responsible for all the keys node 25 was covering. An example Chord network is drawn in in Figure **??**.

With this scheme, we can reliably find the node responsible for some key by asking the next node in the circle for the information, who would then pass the request through the circle until the successor was found. We can then proceed to directly connect with the successor to retrieve the file. This naive approach is largely inefficient, and is a simplification of the lookup process, but it is the basis of how Chord theoretically works.

To speed up the lookup time, each node builds and maintains a *finger table*. The *finger table* contains the locations of up to m other nodes in the ring. The ith entry of node n's *finger table* corresponds to the node that is the $successor(n + 2^{i-1})$ mod 2^m. Because hash values won't be perfectly distributed, it is perfectly acceptable to have duplicate entries in the *finger table*.

When a node n is told to find some key, n looks to see if the key is between n and $successor(n)$ and return $successor(n)$'s information to the requester. If not, it looks for the entry in the finger table for the closest preceding node n' it knows and asks n' to find the successor. This allows each step in the to skip up to half the nodes in the network, giving a $\log_2(n)$ lookup time. Because nodes can constantly join and leave the network, each entry in the table is periodically checked and updated.

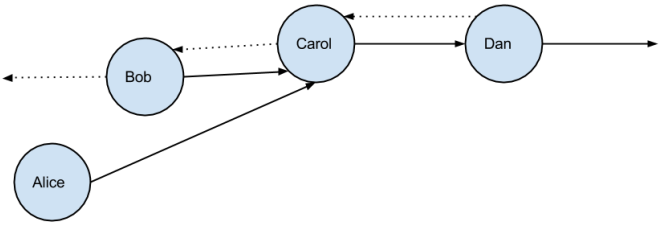

Fig. 1: Alice has incorrectly determined that Carol is her appropriate successor. When Alice stabilizes, Carol will let her know about Bob.

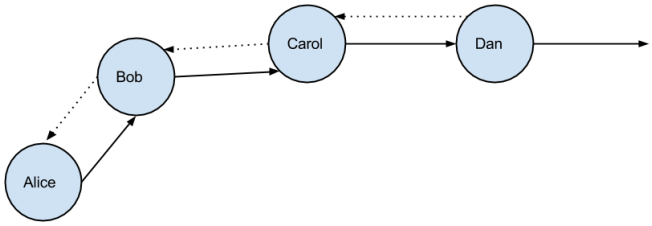

Fig. 2: After completing stabilize, Alice makes Bob her successor and notifies him. Bob then made Alice as his predecessor.

To join the network, node n first asks n' to find $successor(n)$ for it. Node n uses the information to set his successor, but the other nodes in the ring will not acknowledge n's presence yet. Node n relies on the stabilize routine to fully integrate into the ring.

The stabilize routine helps the network integrate new nodes and route around nodes who have left the networls. Each node periodically checks to see who their successor's predecessor is. In the case of a static network, this would be the checking node. However, if the checking node gets back a different node, it looks at that returned node's hash value and changes their successor if needed. Regardless of whether the checking node changes its successor, that node then notifies the (possibly) new successor, essentially telling the successor "based on the information I have, I'm your predecessor. Check to see if you need to update your predecessor information," to which the successor obliges. A more concrete example:

Suppose Alice, Bob, Carol, and Dan are members of the ring and everyone happens to be ordered alphabetically (Figure 1). Alice is quite sure that Carol is her successor. Alice asks Carol who her predecessor is and Carol says Bob is. Since Bob is closer than Carol, Alice changes her successor to Bob and notifies him.

When Bob sees that notification, he can see Alice is closer than whoever his previous predecessor is and sets Alice to be his predecessor. During the next stabilization cycle, Alice will see that she is still Bob's predecessor and notify him that she's still there (Figure 2).

One of the major design choices for Chord implementation is not figuring which node is responsible for a given key, but figuring out who decides which node is responsible for a given key. In our implementation, a node n is responsible for the keys ($predecessor(n)$, n]. In other words, when n gets a message, it considers itself the intended destination for the message if the message's destination hash is between $predecessor(n)$ and n. A node that does not have a predecessor assigns itself as its own predecessor and considers itself responsible for all messages it receives. COUNTER WHY THIS IS A BAD IDEA. THIS IS A VERY DEFINITE CRITICISM

When a node n changes his successor, n asks if the successor is holding any data n should be responsible for. The successor looks at all the data n is better suited to hold onto, packages it up, and sends it along to n. The successor does not have to delete this data. If fact, keeping this data as a backup is beneficial to the network as a whole, as n could decide to leave the network at any point.

Due to the potentially volatile nature of a peer-to-peer network, Chord has to be able to handle (or at the very least, tolerate) an arbitrary amount of churn. We already detailed how Chord gradually guides nodes into their correct locations after they join the network. The same is true for when a node leaves the network; the stabilize procedure will guide nodes to their correct successors and predecessors. However, we can exert more control over how to handle nodes leaving the network

A node can leave the ring in one of two ways. A node can either suddenly drop out of existence, or a node can tell the network he is about to leave, letting his successor and predecessor immediately perform the needed changes.

When a node politely quits, he informs both his successor and predecessor and gives them all the information they need to fill the gap that would be left over. He also sends all of the data he is responsible for to his successor, who would become responsible for that data when that node left. Fingers that pointed to that node would be corrected during the finger maintenence period. This allows for the network to adjust to the change with a minimum of fuss.

Unfortunately, it is impossible that every time a node leave the network it will do so politely. If a node suddenly quits, the data it had stored goes with it. To prevent data from becoming irretrievable, a node can periodically send backups to its successor. So as not to overwhelm the ring with a cascade of backups of backups, the node only passes along what it considers itself responsible for, which changes as nodes enter and leave the network. If the backup leaves, he send his stuff to his successor, since the backup's successor would be the one responsible for the info now.

C. Extensions of Chord

The Cooperative File System (CFS) is an anonymous, distributed file sharing system built on top of Chord [5]. In CFS, rather than storing an entire file at a single node, the file is split up into multiple chunks around 10 kilbytes in size. These chunks are each assigned a hash and stored in nodes corresponding to their hash in the same way that whole files are. The node that would normally store the whole file instead stores a *key block*, which holds the hash address of the chunks of the file.

The chunking allows for numerous advantages. First, it promotes load balancing. Each piece of the overall file would (ideally) be stored in a different node, each with a different backup or backups. This would prevent any single node from becoming overwhelmed from fulfilling multiple requests for a large file. It would also prevent retrieval from being bottlenecked by a node with a relatively low bandwidth. Finally, when Chord uses some sort of caching scheme like that described in CFS [5], caching chunks as opposed to the entire file resulted in about 1000 times less storage overhead.

Chunking also opens up the options for implementing additional redundancy such as erasure codes[13]. With erasure codes, redundant chunks are created but any combination of a particular number of chunks is sufficient to recreate the file. For example, a file that would normally be split into 10 chunks might be split into 15 encoded chunks. The retreival of any 10 of those 15 chunks is enough to recreate the file. Implementing erasure codes would presumably make the network more fault tolerant, but that is an exercise left for future work.

D. MapReduce

1) History: At its core, MapReduce [6] is a system for division of labor. One very popular implementation of MapReduce is Apache's Hadoop. [3].

2) Example: A task gets sent to a master node, who then divides that task among slave nodes, which may further divide the task. This task has two distinct parts: Map and Reduce. Map performs some operation on a set of data and then reduces a result for each map operation. This intermediate data is then reduced by other slave nodes then returned to the user.

The classic example given for MapReduce is counting the word in a collection of documents. The master node splits up the documents into multiple chunks and sends them off to workers [1]. Each worker then goes thru each chunk and creates a small word frequency list. These lists are then used by other workers, who combine them into larger and larger lists, until the master node is left with a word frequency list of all the words in the documents.

3) Map Reduce advantages and disadvantages: this is a good survey [**?**].

III. Related Work

A. Volunteer Computing

In recent years, there has been a trend of crowd-sourcing large and complicated tasks among willing participants. Fold-

[1]Workers can further split up the chunks among other workers.

ing@home is a distributed program for simulating the way proteins put themselves together, a process called folding [2]. [2]. The program has been a huge boon to the research of various diseases, such as Alzheimer's Disease, cystic fibrosis, and Huntington's Disease. These diseases and others are believed to be tied to proteins misfolding, or failing to assemble properly. Folding@home simulates the molecular dynamics of proteins during the folding process, with the aim to understand how misfolds occur[2]. A fast processor can simulate about 20 nanoseconds of behavior a day, but protein interactions occur at the millisecond and second scale, which necessitated a distributed approach.

This concerted effort has been immensely successful, providing data for 109 peer-reviewed publications as of August 2013 [**?**][3].

In a similar vein, SETI@home is a concerted effort by millions of users to analyze collected radio signals for signs of intelligent life [**?**][4].

Great Internet Mersenne Prime Search, or GIMPS, is a large scale distributed computing software to find Mersenne Primes. Mersenne primes are prime numbers of format $M_p = 2^p - 1$, where p is a prime number. The largest known prime number is a Mersenne Prime found using GIMPS.

Our program would provide an excellent platform for users to write their own program for volunteer work.

B. P2P-MapReduce Systems

P2P-MapReduce [10], is similar to our work, but looks only at MapReduce; MapRecude is only one of the services CHRONUS provides. It consumed more network resources than the traditional centralized implementation, but was much more tolerant to churn and lost less time when nodes?jobs failed[5]. P2P-MapReduce was not implemented on a large scale; the test results for larger networks were derived from simulations.

Closest to our work is Lee et al.'s work *Paarallel Processing Framework on a P2P System Using Map and Reduce Primitives* [**?**]. Their work, like ours, draws attention to the fact that a P2P network can be much more than a way to distribute files and demonstrates how to accomplish different tasks using map and reduce functions.

1) Underlying protocol: Chord vs Symphony: [**?**][6] is implemented on top of BruNet[**?**], which itself is an implantation of the DHT protocol Symphony [**?**]. Symphony and Chord share a great deal of symetry. Both protocol create an overlay in the shape of a ring, both use a hash to assign files to a node that corresponds to that hash, and both use a finger table[7] to create shortcuts across the ring.

The difference is that Symphony seeks to exploit the small world phenomena [**?**], where the fingers are chosen at random along a probability distribution function. The further away a node is, the less likely it will be chosen as a finger [8]. Like Chord, messages travel along the paths that bring them closest to their target destination, which is the node responsible the destination's hash value. In a network with N nodes, each with k fingers, a message will take on average $O(\frac{1}{k}\log^2(N))$ hops to reach its destination. In comparison, the average lookup time in Chord is $\frac{1}{2}\log(N)$ [17].

To speed up routing in Symphony, the fingers are bidirectional, rather than unidirectional (does this mean when k is 4 there's effectively 8 fingers? I think it does according to the # tcp connection). Symphony also has nodes maintain a 1-lookahead list for each finger [9].

In the simulation of a 2^15 node network in [**?**], this allowed the Symphony to perform at nearly identical speeds with Chord while utilizing only 4 bidirectional fingers and a 1-lookahead. With 27 bidriectional finger and 1-lookahead, Symphony's lookups took half the time that Chord did.

However, for large numbers of fingers, keeping a lookahead list becomes expensive. Unless their metrics for Symphony are actually log base 10, then it's unbelievable amazing looking. I have my doubts there.

2) Maintenence: Symphony is extremely effective in a (relatively) smaller network because of the simplicity of the maintenance.

It's okay to have $< k$ fingers or different k for a finger.

3) Fault Tolerence: At a glance Symphony and Chord seem to handle churn the same way, backing up nodes for the data.

Nodes entering the a Symphony ring choose their address uniformly at random, do they hash it? If not, then nodes will basically be unable to resume their place in the network.

4) Implementation of map reduce:

5) Important thing is the paradigm shift:

6) Does Brunet address security?: Doesn't look like it, so we should especially talk about our encrypted chat later and make mention of it heare

7) We're looking at a giant computer, a distributed operating system.: Our work handles issues of fault tolerance and reassigning lost jobs. We also don't explicitly form a hierarchical structure.

IV. CHRONUS

A. Implementation

Paragraph here about the bidding nature of the amazon market. age of the inherent ring structure.

[2]Get this entire paragraph reviewed by a bio guy

[3]Check the format of this citation

[4]Check the sizes, I think seti is bigger

[5]Fix this, actual quote was "In summary, the experimental results show that even if the P2P-MapReduce system consumes in most cases more network resources than a centralized implementation of MapReduce, it is far more efficient in job management since it minimizes the lost computing time due to jobs failures."

[6]I need to find a better way to reference this; it doesn't have a catchy name.

[7]For simplicity, we use the Chord terminology in discussing Symphony concepts. The *long-distance links* are analogous to Chord's fingers and the *short-distance links* correspond to the predecessor and successors of a node.

[8]Check this with brendan.

[9]these speedups could be implemented on chord.

1) Code Details: We implemented the Chord based on the psuedocode in the Stoica's paper [17], using Python instead of C++. We also sent messages instead of performing remote procedure calls.

B. Distributed MapReduce

1) Operation: In our implementation of a distributed map reduce, each node takes on responsibilities of both a worker and master, much in the same way that a node in a p2p file-sharing service will act as both a client and a server. To start a job, the user contacts a node a specified hash address. The node he contacts to be the stager may be his own computer, and this is preferable when the job involves dividing up a large pile of data.

The job of this stager is to take the work and divide it into *data atoms*, which are the smallest individual units that work can be done on. This might a line of text in a document, the result of a summation for a particular intermediate value, or a subset of items to be sorted. The specifics of how to divide the work are defined by the user in a *stage* function. These *data atoms* are then given a random hash and sent to that hash address, guaranteeing an even distribution of the data atoms throughout the network. The *data atoms* also contain the map function and reduce function, as defined by the user.

Nodes which receive data atoms apply the map function to the data to create intermediate data atoms, which are sent to other BLANK. DESCRIBE HOW TWO INTERMEDIATE ATOMS GET TO THE SAME PLACE. Nodes that receive at least two intermediate values merge them into one data atom using the reduce function. The atoms are continually merged until only one remains at the hash address of the stager.

Once the reductions are finished, the user gets his results from the node at the stager's address.

Once the stager has sent all the data atoms, his job is done. The stager does not need to collect the results work, since the work is sent to the stager's hash address, rather than the stager itself. Thus, the stager could quit the network after staging, and both the user and the network would be unaffected by the change.

2) What happens if the stager goes down while staging:

3) Recap the issues of normal MapReduce and why ours is better: We present our implementation of MapReduce not as a direct competitor to MapReduce, but as proof of a more versatile system able to support many complex operations.

The big advantage of our system is the ease of development.

The developer does not need to worry about distributing work evenly, nor does he have to worry about any node in the network going down. The underlying Chord ring handles that automatically. If a node joins the ring as the MapReduce process is running, that node can be assigned work automatically. The stager does not need to keep track of the status of the network. In a node goes down while performing an operation, his successor takes over for him.

All a developer needs to do to write three functions: the staging function, map, and reduce. These define how to split up the work into manageable portions, the work to be performed on each portion to get some results, and how to combine these results into a single result.

In the next section, we show two of the implementations we ran on our ring of nodes.

C. Example implementations of mapreduce

1) Calculating Pi:

2) Word Count:

3) My computer can do way more work: In which case you boot up more instances of a node, declaring you can do that much more of the network's work.

D. In which we address possible criticism

1) If we hash a filename to get the identifier, then you can't have two same file names.:

2) Determining when you're done is expensive: As a whole yes, but merging two sorted lists is $O(n)$ time.

3) Disjoint rings: We can assign every node a *ring id*. When a node creates a ring, he randomly chooses a number for the ring id. Nodes joining that ring use that ring's id. If a node finds another node on differing ring, he compares the their ring id's and leaves if

4) The security we didn't do: We can have multiple non interacting secure file systems.

V. Experiments

A set of experiments were run on large groups of Amazon EC2 Micro Instances[1].

We ran word count on James Joyce's *Ulysses* [**?**].

VI. Results

VII. Conclusion

Our stuff [15] is awesome.

References

[1] Amazon.com. Aws documentation: Micro instances. http://goo.gl/1yWuy.

[2] Adam L Beberg, Daniel L Ensign, Guha Jayachandran, Siraj Khaliq, and Vijay S Pande. Folding@ home: Lessons from eight years of volunteer distributed computing. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[3] Dhruba Borthakur. The hadoop distributed file system: Architecture and design, 2007.

[4] WU Chehai, LU Xianliang, D. Hancong, T. Hui, Z. Xu, and Z. Zhijun. Analysis of content availability optimization in bittorrent. In *Hybrid Information Technology, 2006. ICHIT'06. International Conference on*, volume 1, pages 525–529. IEEE, 2006.

[5] Frank Dabek, M Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. *ACM SIGOPS Operating Systems Review*, 35(5):202–215, 2001.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[7] R. Jimenez, F. Osmani, and B. Knutsson. Connectivity properties of mainline bittorrent dht nodes. In *Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on*, pages 262 –270, sept. 2009.

[8] Eng Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys Tutorials, IEEE*, 7(2):72 – 93, quarter 2005.

[9] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th international conference on Supercomputing*, ICS '02, pages 84–95, New York, NY, USA, 2002. ACM.

[10] Fabrizio Marozzo, Domenico Talia, and Paolo Trunfio. P2p-mapreduce: Parallel data processing in dynamic cloud environments. *Journal of Computer and System Sciences*, 78(5):1382–1402, 2012.

[11] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric, 2002.

[12] S. Ortiz. Is peer-to-peer on the decline? *Computer*, 44(2):11 –13, feb. 2011.

[13] Luigi Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM Computer Communication Review*, 27(2):24–36, 1997.

[14] Pablo Rodriguez, See-Mong Tan, and Christos Gkantsidis. On the feasibility of commercial, legal p2p content distribution. *SIGCOMM Comput. Commun. Rev.*, 36:75–78, January 2006.

[15] Andrew Rosen, Brendan Benshoof, and Matt Erwin. Chronus. https://github.com/BrendanBenshoof/CHRONUS.

[16] Haiying Shen. Irm: Integrated file replication and consistency maintenance in p2p systems. *Parallel and Distributed Systems, IEEE Transactions on*, 21(1):100 –113, jan. 2010.

[17] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31:149–160, August 2001.

[18] Chonggang Wang and Bo Li. Peer-to-peer overlay networks: A survey. Technical report, 2003.