Anthill Network

Ross Grundy and Ryan Novitski
Bowdoin College '24 and '25
16 December 2023
Brunswick, ME 04011
rgrundy@bowdoin.edu; rnovitsk@bowdoin.edu

Distributed Systems 3325 Final Project
Professor: Sean Barker
Bowdoin College, Department of Computer Science
sbarker@bowdoin.edu

***Abstract*—This paper introduces Anthill, a decentralized distributed system designed to anonymize application-layer network traffic when sending HTTP requests. Utilizing a scalable peer-to-peer architecture, Anthill is implemented in Java with the Apache Library and XML-RPC Framework. This paper will go through Anthill's design, implementation, evaluation process, related works, and possible future adjustments.**

I. Introduction

For as long as two machines have been able to connect to one another, there has existed the vulnerability for an attack to occur on any data that is being stored or sent within that connection. Anthill looks to directly reduce the risk and danger involved with sending requests across the Internet. Using the decentralized nature of a peer-to-peer ring architecture, Anthill provides the ability to send and receive HTTP requests without the worry of your data being traced back to your specific machine.

Although anonymity is at the forefront of Anthill's design decisions, there are still some necessary connections that need to be stored in each node of the system to create request paths as well as dynamically add or remove nodes. These nodes are stored in a list of IP addresses called the colony table, which holds a fixed number of addresses from different locations around the ring. Each colony table begins with the node's first successor(i.e. the node directly next to it within the ring structure), and makes exponential increments for the remaining nodes(2, 4, 8, etc.). This specific layout of the colony table is crucial to a variety of algorithms that are used in the system, including the dynamic joining and disconnecting of nodes, as well as the request path setup. The feed-forward implementation of the ring structure provides partial anonymity from other users within the system. Since each colony table begins with each node's first successor, it cannot be determined if an incoming request to a particular node originated from the previous node, or if it is simply being passed along.

A key feature within our Anthill system is the ability for any node to act as a bootstrap or exit node. If the original bootstrap node becomes unresponsive, it is swiftly replaced with another responsive node. This design decision eliminates the possibility of the bootstrap node being a single-point of failure. Although having a dynamic exit node also eliminates it as a potential point of failure, it also serves a greater purpose of evenly spreading out traffic on the network. This has the potential to maximize efficiency as well as make it more difficult to track where requests are specifically coming from.

Java was the main language used to implement Anthill, along with bash scripting for ease of use setting up connections between nodes. Additionally, the Apache and XML-RPC frameworks and libraries were used to send messages between nodes in the server.

II. Design/Architecture

This section of the paper features an in-depth analysis of the different design decisions and functionalities that provide the Anthill system its unique properties.

A. Peer-to-Peer(P2P) Ring Approach

Anthill's primary structure consists of an interconnected ring of different nodes in the system called a *peer-to-peer(P2P) architecture*. As a core concept, a peer-to-peer system uses a linked list traversal approach, storing only a certain number of nodes within each machine. When a node leaves, it does not affect the functionality of the rest of the system as a whole. This is purposefully designed to eliminate any potential single points of failure that could be caused by one node leaving or crashing.

Instead, every node holds the same level of importance as any other node; if a machine that is providing a certain functionality to the system goes down, a new node will take its place as soon as its unresponsiveness is detected.

In addition to providing a significant boost to fault tolerance, the peer-to-peer ring structure also helps achieve anonymity within the system. As described in the introduction, the Anthill ring is designed to be feed-forward; a particular machine does not have access to or information about any of the nodes behind it. This provides a blanket level of security to the sender since its successors have no knowledge of where the request originated from.

B. Drones

Each machine that connects to the Anthill architecture is called a *Drone*. Each drone operates independently from one another, and once connected into the system they do not rely on any particular node to stay connected. Instead, any Drone has the capability to perform the jobs of any other node. These functionalities include sending XML-RPC requests between machines in the system, adding new machines into the system

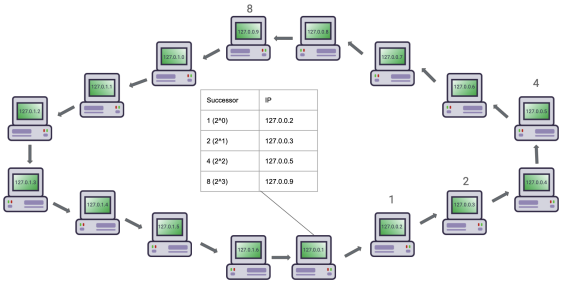

Fig. 1. Trivial example of connections between drones

once it is already running, and sending HTTP requests from the system out to the Internet. The specific functionalities of each Drone will be discussed further in-depth later on in this section.

C. Colony Table

The primary method to maintain communication and connection between drones in the system is referred to as a *colony table*. The colony table stores a fixed number of IP addresses from machines within the network to help effectively traverse the ring structure, provide fault tolerance, and form paths for each request made by a client.

1) Structure: The colony table holds a fixed number of successors spread out across the ring. Each successor stored is 2^n nodes away from the previous node in the colony table. This special relationship between nodes in the colony table is crucial for the algorithms that are used to operate many of the other functionalities within Anthill.

2) Effective Traversal: Because of the exponential relationship between the position of nodes in each colony table, it makes it much easier for a particular message to get sent around the ring without being localized to a particular area. In contrast, if the colony tables were stored with nodes being direct successors of each other, the chances of the message to stay close to the originating node grows significantly higher. Instead, a request in a 16 machine system and 4-IP colony table could theoretically traverse the entire ring with 2 passes.

3) Fault Tolerance: The colony table also lets the system dynamically fix itself if there are any nodes that join or leave while it is running. Since each node has a list of other potential nodes in the system, it makes it possible to communicate with each to update its table whenever a machine becomes unresponsive. The algorithms that are utilized to update the colony tables while still maintaining the exponential sequencing of successors will be explained further in the paper.

D. Initializing Network

The Anthill system begins by initializing the first server to connect as the bootstrap server. This server has the ability to connect any other machine that requests to join with its IP address. The first bootstrap node initializes with a complete list of current members inside of the system to combat any potential errors finding the first few connections when the system is small. Once an entire colony table has been filled with unique IP addresses, the node ceases to keep track of the entire list of connected machines. Right upon initialization since there are no other machines in the system yet, the bootstrap's colony table is initially filled with just itself.

E. Joining Network

Once the first bootstrap server is ready to make new connections, Anthill can begin to connect other machines into its system and update colony tables dynamically.

1) Propagating Changes: When the bootstrap server receives a request from a new machine to join, it begins by placing the new machine's IP address into the last index of its own colony table. This places the new node 2^n nodes away from the bootstrap and sets it into position to call the function that recursively adds the node into the correct place for the rest of the machines affected by the new placement.

2) Adding New Nodes: Once the bootstrap node begins the execution, each node recursively replaces every value in its colony table from the dead node's index(i) down with the colony table from its $i-1[st]$ index's table. Starting this from the bootstrap node and propagating it through upwards will allow it to correctly update every colony table's values with the new positions of each node.

It should be noted there is a exponential trade-off between the size of the colony table and the time it takes to propagate these changes to the entire system. Since each table entry corresponds to a value $2[i]$ nodes away, there are $2[n]$ tables that need to be adjusted every time(with n being the size of the colony table). So, for a colony table with a size of 4, there are 16 updates to make whenever the replacement algorithm is called.

F. Creating a Request Path

A new path is dynamically created for every request that is sent within the system. When a node would like to send out a request, it calls a function that specifies the path length, requested URL, and the operation of the actual request. The path length corresponds to the number of machines it must pass through before it actually gets sent out; in order to ensure anonymity between machines in the system, it is recommended to have a path length of at least 3. The path length does not have to be lower than the number of machines in the system; the request method allows for the message to be passed around the ring multiple times before it is actually sent out.

Each time a node receives an outgoing request, it has the option to either execute it or pass it along to a random node in their colony table until the path length is reached. However, the path length is not concrete; there is also a random probability the message will be passed regardless of if the path length has been reached. This provides another layer of security and ensures it is impossible to track the exact origin of any particular request being sent out.

G. Sending/Receiving a Request

Once the request reaches its final destination and a node is designated to execute it, the HTTP request is ready to be

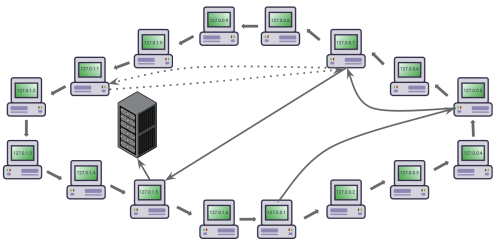

Fig. 2. Figure depicting example of random request path

sent out. The request is processed using a request builder that was created using a simple Apache HTTP framework. This request builder is capable of handling get, post, head, delete, put, and patch requests and returns back to the exit node with the response data. Once returned, the output is stored in a customized Response class that holds all of the relevant data that was received.

At this point in time, the previous nodes in the path before the exit node are waiting to send the response backwards through the path it had originally been sent through. This is the only time in the system where any information or communication is sent backwards through the ring. To achieve this, each node in the path temporarily stores the node that initially called them in order to eventually propagate the response data back to the original sender. Once each node returns its call backwards and the sender gets its desired information, the path is closed and won't be used again unless randomly generated a second time.

H. Dynamic Joins

While still the same concept as the initial joining of nodes, the Anthill system also supports dynamically joining the system once it has already begun. Following the same algorithmic steps as the initial joining sequence, a new node is first placed in the last index of the bootstrap's colony table.

I. Dealing with Failures

Every system with a lot of moving parts has the potential to run into some failures along the way. In the case of a peer-to-peer architecture, there is a high probability of encountering churn when running the system at any given time. With that being said, it is very important to have a recovery system in place for whenever a node fails that allows the structure of the ring and each colony table to remain intact. This is done within Anthill via a very similar algorithmic process to the one utilized in the dynamic join method.

1) Recovery Process: Every node in the ring incrementally performs a ping to each of its successors every thirty seconds. If any of these pings are not successful, the pinging node will take action according to the unresponsive node's position in its colony table and the number of times that the node has been marked as unresponsive. If the down node is in the last index of the pinging node's colony table, the pinging node will execute the same algorithm that is used when a new node is joining the system.

When the down node is in any other position in the pinging node's colony table, the pinging node checks to see how many times it has marked that node as down. If the node was marked down *less than the size of the colony table subtracted by the down node's index*, its down counter will be incremented and the pinging node will skip over it when forming any paths until the executing node finishes propagating its changes to the rest of the system.

However, if the node was marked down by at least the size of the colony table subtracted by the down node's index, the replacement algorithm will be executed by that pinging node instead. This is due to an edge case in the sweeping nature of the algorithm. Under normal conditions, every machine ignores the unresponsive node until it is replaced by the pinging node that has the down node in its last index. However, in the case where a node that has its own IP as its last index goes down(a possibility for a 4-IP colony table system running exactly 8 servers), the other machines would have no way of removing the down node from their tables. To combat this, if a node is marked down more than once by the second to last node, it will trigger the replacement algorithm to run. After another 30 seconds, if the node is still not replaced, the third to last node will trigger the replacement algorithm. This continues until the first node in the colony table is reached.

III. Implementation

This section of the paper describes the specific implementation details that were chosen for this project. There are some potential improvements that could be made to the system as a whole by experimenting with different libraries, languages, and frameworks. These improvements will be discussed within a later section.

A. Language and Frameworks

1) Languages: The main language that was used during this implementation of the Anthill system was Java 1.8. Java was chosen due to its high level of support for complex server operations without the added overhead of a language like C. Additionally, the object-oriented nature of Java was helpful in creating the Response objects that are sent back from the HTTP requests, as well as the ability to create a Logger class and thread all output into one place for easy debugging purposes.

Additionally, Bash scripts were used to create and set up various servers and server information, including cloning the code onto each server, pulling current data onto each machine, initializing the bootstrap and setting up specifically-sized networks, and killing all active processes when needed.

2) Frameworks: XML-RPC was chosen as the primary communication method between servers in the system via the Apache XML-RPC library. XML-RPC provides a simple way to send encrypted HTTP messages, while still being adaptable to different objects and return values.

HTTP was chosen as the primary framework to send requests out to the Internet and back from the exit nodes. In a future implementation, using HTTPS has the potential to

provide added security to messages being sent out from the network, strengthening the system's overall anonymity.

B. User Interface

The user interface is a simple text-based command-line interface. When the user's client opens, they are met with the choice to either enter "Send", "Info", or "Quit". The "Info" command outputs the current colony table information, while the "Send" command prompts the user for their request. For each request, the user has the ability to specify the minimum path length, the URL, the HTTP method they would like to execute, as well as any parameters that may be needed for the requested method. Once a valid input has been given, the request is built, sent through the randomized path, executed, and sent back to the user with the response.

IV. Evaluation

A. Transfer Speed and Latency

1) Path Length: The path length is allowed to vary by request. It allows the user to choose the minimum number of nodes the request/response must travel through in order to complete the transaction. With each increase in path, comes a significant increase in overhead to complete transaction. The effect is striking and can be seen in Fig 3,4. As the path length increases, the transfer speed begins to decrease exponentially. This makes paths over 10 nodes long almost entirely infeasible.

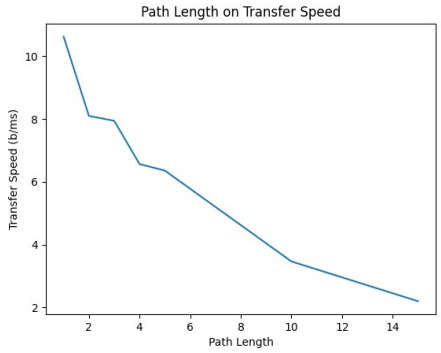

Fig. 3. Path Length vs Transfer Speed

2) File Size: As most of the latency of this application comes from the overhead of establishing the path, larger files appear to have significantly higher data transfer rates. The overhead of the network remains constant, but the data transfer rate changes based on the file size. This in turn correlates to larger files being viewed preferentially in the lens of file transfer speed.

B. Randomized Request Paths

1) Network Distribution: Since the goal of this program is to distribute network traffic over a wider array of computers, one of the measures of its success is its distribution of requests throughout the entire network. Figure 7 shows that while network traffic is spread over the entirety of the network, it is not equally distributed. There is still a high concentration of requests being performed locally especially when there are a low number of requests being filled. However, Figure 7 also shows that there is an upward trend in requests served to nodes further from the bootstrap.

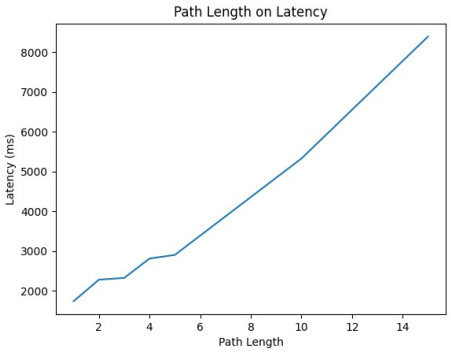

Fig. 4. Path Length vs Latency

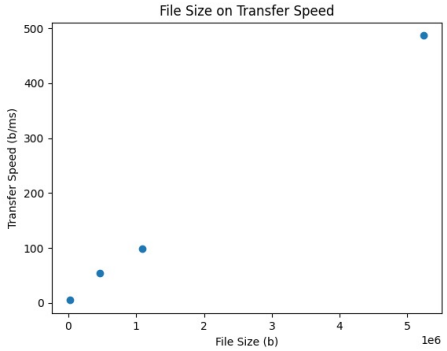

Fig. 5. File Size vs. Transfer Speed

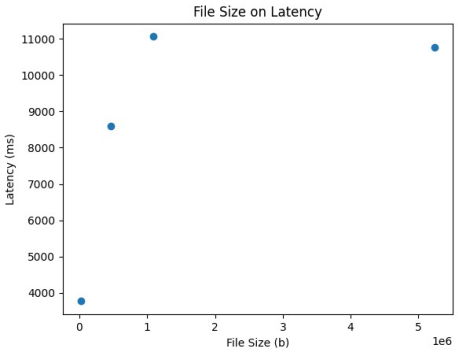

Fig. 6. File Size vs. Latency

V. Related Works

In the realm of anonymizing network traffic, two significant systems stand out as predecessors to Anthill: Tor and Tarzan. All three systems, while distinct in their approaches, share the

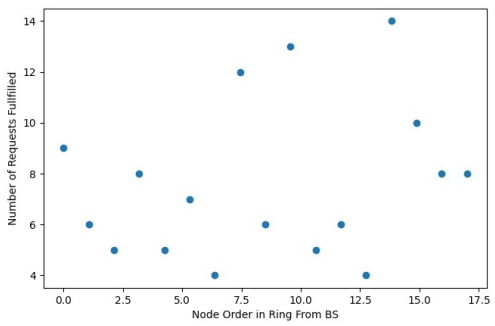

Fig. 7. Chart showing random nature of paths

common goal of preserving user anonymity and security in digital communications.

1) Tor: Tor, also recognized as 'the Onion Router', presents a complex approach to anonymizing network traffic. Tor utilizes a worldwide network of volunteer-run servers to relay and encrypt user traffic multiple times. The cornerstone of Tor's architecture lies in its layered encryption mechanism, which incrementally peels away layers of encryption at each checkpoint similar to the layers of an onion.

The parallels between Tor and Anthill are evident in their mutual commitment to preserving anonymity. However, Anthill diverges with its peer-to-peer ring structure, which inherently decentralizes the network topology contrasting Tor's reliance on a more static network of volunteer nodes. Additionally, Anthill's colony table, dynamic node allocation and replacement strategies offer a unique approach to network resilience and fault tolerance, differentiating it from Tor's methodology.

2) Tarzan: Tarzan, another noteworthy system designed to anonymize network traffic, adopts a peer-to-peer approach similar to Anthill. It uses a decentralized network of nodes, or "mimics," to relay Internet traffic. The key feature of Tarzan is its ability to disguise the traffic of its users within the normal traffic flow of the Internet, making it challenging to distinguish or trace.

Anthill and Tarzan share the common ground of peer-to-peer architecture, emphasizing the importance of decentralized control and scalability. However, Anthill extends these concepts with its innovative use of the colony table for efficient network traversal and the integration of a feed-forward ring structure to enhance anonymity. Additionally, Anthill's design allows for dynamic adaptation in response to network changes, such as node failures or additions, a feature that provides an edge over Tarzan's more static approach.

3) Final Comparisons: Both Tor and Tarzan have set benchmarks in the field of anonymizing network traffic, offering insights and foundational concepts that have directly contributed to the development of Anthill. Anthill's unique contributions lie in its hybrid approach that combines the strengths of these systems while addressing their limitations. The peer-to-peer ring architecture, coupled with the colony table and dynamic node management positions Anthill as a scalable and adaptable system in the ever-evolving landscape of network security and anonymity. This innovative approach not only reinforces the system's resilience against failures and attacks but also ensures a high degree of user anonymity, giving it strong potential for a significant advancement in the field of secure and anonymous communication.

	Bad First Relay			Bad Intermediate Relay			Bad Last Relay			Bad First and Last Relay		
Protocol	OR	Tarzan	AntHill	OR	Tarzan	AntHill	OR	Tarzan	AntHill	OR	Tarzan	AntHill
Sender Activity	Yes	Maybe	No	No	Maybe	No	No	No	No	Yes	Maybe	No
Recipient Activity	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Sender Content	No	No	No	No	No	No	No	No	Yes	Yes	Maybe	Yes
Recipient Content	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes

Fig. 8. Table comparing different stats of Tor, Tarzan, and Anthill

VI. Future Improvements

Part of the original intention of this project was anonymity. While the identity of the sender is hidden in the network, if the data being transmitted has identifying information, the whole network is for naught. As part of the future improvements of this project and to fully realize its potential, a secure and reliable encryption methodology is required. Anthill's encryption takes its cues from its peer Tor, utilizing onion routing on both the request and response to ensure the safety of the message. In order to institute this, the path must be traversed twice to pass keys and then the messages. The XML-RPC protocol makes this quite difficult, as there is no persistent links where data can be passed. To remedy this the path will be stored with a unique identifier that would allow recreation of the path. This path is recreated and then discarded as to eliminate the possibility of it being traced again.

Other future improvements include garlic routing, which would allow the network to disguise not only the identity but the actual quantity of requests from a global listener. This feature clumps the requests together as they pass from node to node, masking the traffic.

A major concern going forward is misuse. While this software is useful, it is also an incredibly powerful DDOS tool. It is a non-centralized distributed system capable of scaling to hundreds of computers and thousands of requests. It could allow one person attack a target while distributing the risk amongst an army of unsuspecting bystanders. At this however, the latency of the system makes somewhat implausible.

VII. Conclusion

In summary, Anthill presents a novel approach to ensuring anonymity and security in application-layer network traffic through a decentralized, peer-to-peer ring architecture. The system, built using Java and leveraging the Apache Library and XML-RPC Framework, offers a robust and scalable solution for secure HTTP request transmission. The peer-to-peer ring structure, central to Anthill's design, eliminates single points of failure, thereby enhancing system reliability and fault tolerance. The concept of Drones and the design of the colony table

algorithms play crucial roles in maintaining system stability and efficiency, ensuring that each node contributes equally to the network's functionality.

Anthill's design balances the need for anonymity with the necessity of maintaining certain connections within each node to create request paths and manage the dynamic addition and removal of nodes. The system's ability to dynamically adjust to changes, whether in adding new nodes or managing failures, demonstrates a high degree of adaptability and resilience. The implementation of a randomized request path and the option for any node to act as a bootstrap or exit node further fortifies the system against potential security threats and enhances its capacity to distribute network traffic evenly.

Looking towards the future, Anthill's architecture lays a strong foundation for further exploration and enhancement in the realm of secure network communication. Potential improvements could include exploring alternative languages or frameworks for more efficient operation, integrating HTTPS for enhanced security, implementing onion-routing public key encryption for requests sent within the system, and refining the user interface for broader accessibility. The system's scalability and adaptability position it well for adaptation to various network sizes and types, making it a valuable tool in the continuous battle for cybersecurity and data privacy.