GNuggies: A Framework for Hosting Resilient Decentralized Services using Untrusted Nodes

Abhishek Modi
akmodi2@*

Harshit Agarwal
hagarwa3@*

Evan Fabry
efabry2@*

*illinois.edu

1. Abstract

Introducing GNuggies - a distributed infrastructure as a service solution, similar to AWS or Microsoft Azure, to help people host services on public untrusted nodes, where all the hosts are untrusted volunteer nodes. Creating this revolutionary service poses certain challenges when dealing with establishing trust in the network while ensuring complete lack of liability for all hosts. We demonstrate our methods and their strengths, and show that this is truly a direction that the internet will benefit from moving towards.

2. Introduction

Infrastructure as a Service and Platform as a Service solutions are very popular today as they allow entities to host services in a relatively simple manner. Offerings range from the extremely robust like Google Cloud Platform, Microsoft Azure, and Amazon Web Services, to the extremely simple like Heroku, Docker Cloud, and Parse.

Now while these solutions are excellent for most services, we find that they often are unacceptable for certain other kinds of services. Examples of these services include protest websites such as WikiLeaks and content sharing websites such ThePirateBay[10]. We believe these are unacceptable because of social reasons rather than technological ones, for instance IaaS providers often choose not to allow TOR nodes, Torrent seedboxes. A high profile example of such a request can be seen in 2010 when AWS refused to continue hosting Wikileaks on request of the US government[1]. Most IaaS solutions require account details that the host of a protest site may not want to provide for fear of persecution. We believe that such services have the potential of creating enormous social good and we would like to afford IaaS-like solutions to these services.

[1] Reuters - Amazon stops hosting WikiLeaks website
reuters.com/article/us-wikileaks-amazon-idUSTRE6B05EK20101202

We introduce GNuggies, a framework for hosting services in a completely decentralized manner on untrusted nodes. The nodes on which GNuggies is run is sourced from crowds - this is similar to the manner in which GNutella, Torrent and Tor operate.

The goals that we set out for GNuggies were: 1)Present service hosters with an interface that is similar to common IaaS offerings. This is so that creators of a service do not have to make any changes to work with GNuggies. 2) The framework should be as resilient as possible. This is because protest sites are often attacked with the attempt to take them offline. We'd like to ensure that every service hosted on GNuggies stays up so long is there are users who want the service to exist. 3) Minimize the liability held by creators of the service as well as for anyone who helps host GNuggies. This is to help remove factors that may discourage people from creating or hosting services. This point is particularly relevant in context of a global increase in authoritarian governments and censorship. 4) Ensure that the hosted services can serve familiar protocols such as http. This is because many existing solutions such as Tor[5] and IPFS[4] serve using special protocols and thus limit the audience of the service to only those who are technically adept.

Our main contributions in this paper include the GNuggies system itself - a free hosting service that reduces liability for hosts. We also create a novel introducer that can introduce new nodes to a system without requiring the node to have prior information about the introducer. Finally, we also create an age based method for security and trust building.

3. Trusting Untrusted Nodes

As discussed in Section 4.Sourcing Compute Resources, GNuggies is designed to be run on volunteered machines and as a result is expected to run almost exclusively on untrusted nodes. GNuggies utilizes the age of nodes in the system in order to trust these untrusted nodes. We reason that malicious attackers would like to take down the system as quickly as possible. We find that we can frustrate an attacker's attempts by partitioning the nodes in the system into two sets. The first set consists of trusted nodes which we call patriarchs[8, 9], the second consists of untrusted nodes. This partition is created using information about how long nodes have been well behaved in the system (the age of the node in the system) - a constant proportion of the oldest nodes are the patriarch set whereas the remaining nodes (and consequently, the newest nodes) form the untrusted set.

By selecting the older group as the trusted group, we ensure that an attacker must provide resources to a system until the youngest trusted node and all untrusted nodes younger than that but older than the attacker have left the system. An attacker must achieve this by either patiently waiting for all such nodes to exit the system, by attacking nodes via DDoS or other means, or waiting for a single older node to exit the system then assuming the age of that node in order to go undetected but advance in the queue.

3.1 Assigning New Nodes to the Patriarch Set

When nodes join the GNuggues system, they are part of the untrusted set of nodes. In order to join the trusted/Patriarch set, nodes must be well behaved for a certain amount of time. We monitor their behavior using them as audit nodes. Within the Bitcoin[1] space this is achieved via systems like TLSNotary, which forces nodes to behave consistently with the published AWS EC2 image via features of AWS[9]. We don't adopt this approach in part because we don't wish to limit the system to nodes provided from AWS.

The audit works by first selecting a stateless service S hosted on GNuggues at random. Let's say there are k nodes that are allocated to S. We then pick M sets of untrusted nodes grouped by relative age, each of which contain k untrusted nodes. These are used to set up M mirrors of the service S. We will replicate each request to S on each of the M mirrors. The outputs from S and from the M mirrors will be used to check each other for two purposes:

1. Ensuring that untrusted nodes are behaving well by comparing the output of the mirror to that of the service S.
2. Ensuring that trusted nodes are continuing to behave well by comparing the output of service S to the absolute consensus output of the mirrors.

In the case of output mismatch, the untrusted node that output the mismatched result is kicked from the system. Alternatively, if all mirror nodes show an absolute consensus output which does not match the output of service S, the input from that test is ran against other trusted nodes to determine if S need be kicked from the system. If S is shown to be correct than the mass number of incorrect mirror nodes are kicked.

Rather than select random nodes within each mirror, we select nodes based upon relative age. This ensures that an adversary who provides many nodes to the system in a civil attack will lose many nodes at once as soon as one dishonest node is detected. This frustrates the attack in which an adversary provides nodes that initially provide correct information and randomly lie at a rate that is proportional to the amount of time since entering the system to avoid early detection but maximize damage once the node becomes trusted. Alternatively, one might opt to kick individual nodes as kicking the whole mirror advances all subsequent nodes by k. Further analysis of pros and cons of this will be delegated to future work.

3.2 Wait Time before Nodes Become Trusted

The size of the Patriarch set is kept small by only assigning a fixed proportion of the total number of nodes in a service to the Patriarch set. This ensures that an attacker has to wait for O(N) nodes to die to attack an N-node service. The amount of time this takes is modelled in Figure 3.1 under the assumption of a Poisson distributed number of exit events from the system at each time interval. This isn't very promising as the wait time for N-k, where k << N nodes, to exit the system appears to scale logarithmically w.r.t. the number of nodes in the service as evidenced by Figure 3.1. This doesn't capture the varied nature of nodes provided to the system as some nodes will be laptops on wireless internet connections with very low average mean time to failure, others may be desktops that are rebooted every few days on average, and others may be VPS instances hosted from the likes of AWS and Digital Ocean that stay up for weeks to months or years. Figure 3.2 shows the results of simulating with nodes of differing expected times-to-exit. This demonstrates that the average wait time is primarily determined by the number of nodes within the system as opposed to the number of long-lived nodes as long as there is at least some low portion of nodes that enter the system that are long lived. It is highly unlikely for more than 30% of nodes that enter the system to originate from VPS instances, so we do not model more than that.

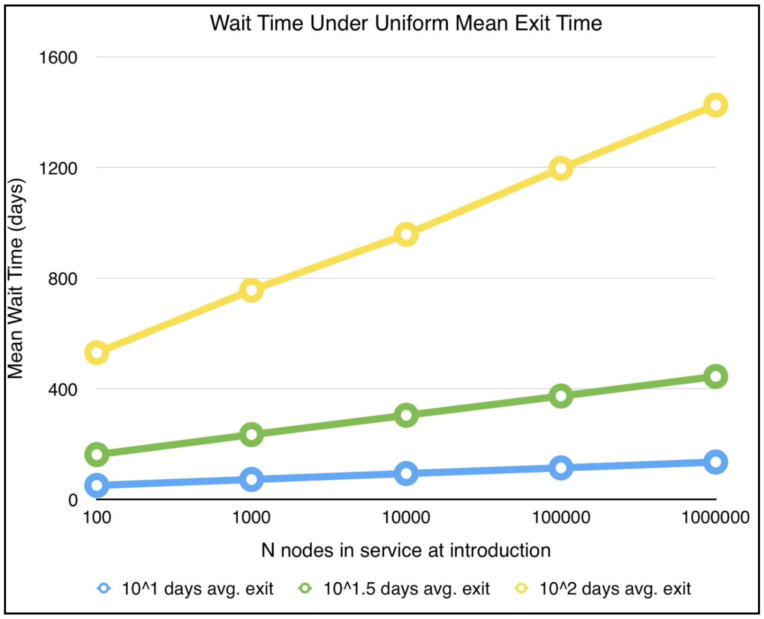

Figure 3.1: Modeled Average wait time for a node to become trusted.

Figure 3.1 demonstrates the average wait time for a node to become trusted under the assumption that all nodes have a Poisson distributed average exit time from the service. The condition to become trusted is that N - k + 1 nodes that are older than the new node leave the system. The x-axis is log scaled, demonstrating that the wait time for a node to become trusted O(log N) wait w.r.t. the size of the system. Each point is the average wait time across 100 simulations.

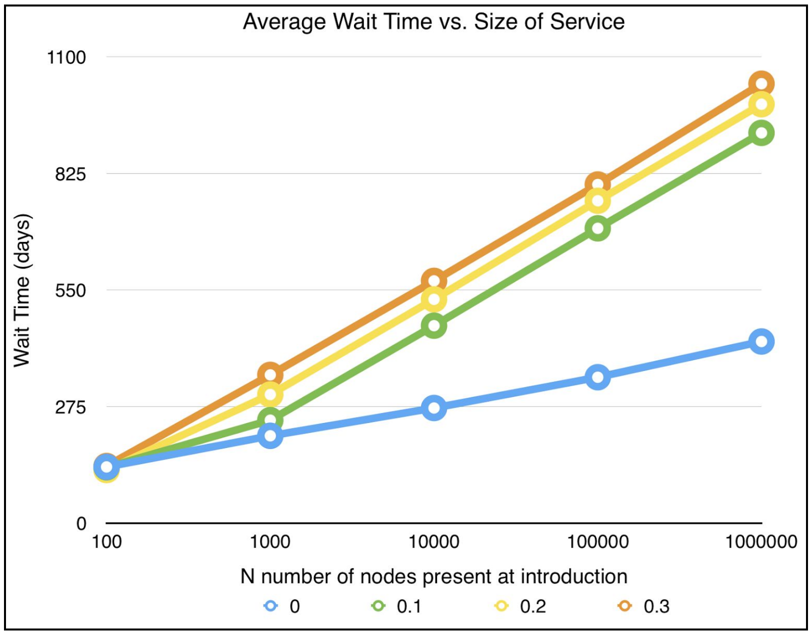

Figure 3.2: Simulation of wait time for a node to become trusted after entry under the assumption that nodes within a service are short, medium, or long lived with Poisson distributed average times to exit of 10^1, $10^{1.5}$, and 10^2 days respectively.

3.3 Tracking Node Age

We can cryptographically force nodes to be truthful in communication of their relative age, identity via use of public and private keys. When a node is introduced to the system, it generates a public and private key, sends its public key to the introducer service, which forwards that key to every trusted node for the service the node is joining. Each trusted node then encrypts the tuple of the public key of the new node and a 64 bit unsigned integer representing the number of entries that have occurred to the service. The number is both the relative age of and a unique ID for the node. The first node to a system is also the first trusted node of the service with unique ID and age zero, the next is one, and so on. All messages between nodes include a tuple of the public key of a trusted node and the encrypted tuple given to the node upon introduction. The node then encrypts that tuple and the message with its private key and sends this outermost encrypted object with its public key. This ensures not only that a node can not forge its identity or age but that it also can not forge a message from another node. This can be leveraged for replay of messages.

```
New node:
def request_intro(self):
    self.public_key, self.private_key = generate_keys()
    introducer_public_key, encrypted_age, callback = request_introduction(public_key)
    encrypted_message = private_key.encrypt((public_key, introducer_public_key,
encrypted_age))
    success = callback((public_key, encrypted_message))

Introducer:
def accept_intro_request(self, requester_public_key):
    self.intro_count += 1
    encrypted_section = self.private_key.encrypt((self.plublic_key, self.intro_count))
    return self.plublic_key, encrypted_section, self.complete_introduction

def complete_introduction(self, requester_public_key, requester_encrypted_section):
    received_key, _, _ = requester_public_key.decrypt(requester_encrypted_section)
    if received_key == requester_public_key:
        self.propagate_introduction_information(requester_public_key,
requester_encrypted_section)
        return True
    else:
        return False
```

Figure 3.3: Pseudo Code for the introduction and encryption process.

4. Sourcing Volunteered Nodes

GNuggies is designed with the intention that it will run entirely on volunteered (and therefore untrusted) machines. In this section, we talk about how these nodes are sourced. In order to make the flow easy to understand, we will look at this from the point of view of a user who is volunteering their system.

4.1 GNuggies Client

The user installs the GNuggies Client. The Client consists of two processes: the first one is the GNuggies Communication Module, and the second is a Docker environment. The communication module is responsible for GNuggies-specific communications such as relaying heartbeats to maintain the membership list. It's main client-side purpose however, is receiving a Docker image and deploying it on the client's machine. These images are created by people who want to host their service on GNuggies; it contains the code to run their service. The Communication Module also provides basic telemetry of the Docker image to the GNuggies Allocator (see section 5.1).

4.2 GNuggies Introducer and Membership List

Once the GNuggies Client is set up, it makes a request to the GNuggies Introducer. The Introducer creates and shares the key that is used for tracking the age of the node. The client also shares details such as the amount of compute power available on the Client machine. This information will be used by the GNuggies Allocator. The Introducer then informs the membership list of the existence of this new node by sharing keys to identify the Client. The Client then sends a heartbeat every HB_DURATION seconds (which is set in configs).

It is important to note that the Introducer and Membership List are services that run on GNuggies. The Introducer is a stateless service and the Membership List is a stateful service with a simple load balancer and whose state is replicated across all of the nodes that run it. This is to ensure that these systems are fault tolerant and are very simple to recover in the case of a crash.

4.3 GNuggies DNS

One important question to consider is how the Client finds the Introducer given that the IP address of the introducer is likely to change often. This is done by simply accessing a known domain name that has the IP address information in its records. To facilitate this, GNuggies hosts a Name Server. Service owners must purchase a domain name to use with their GNuggies service and make sure that the NS records of this point to the GNUggies Name Server and that the C-Name record contains the UID of the service. Once this is done, the GNuggies Name Server will be updated every time the Client machines that run the Service are changed.

5. Allocating Nodes to Services

GNuggies is designed with the intention that it will run entirely on untrusted volunteered machines. This is so that the system is not under the control of any individual and instead, intelligently decides which volunteered machine runs which service. This helps reduce liability of both, owners of GNuggies Services and owners of the GNuggies nodes.

5.1 Fairness and the GNuggies Allocator

The GNuggies Allocator uses information about the history of utilization of various services in order to allocate nodes to services. It obtains this information from the Communication Modules in the

GNuggies Client processes. It then uses this information along with information about the compute power of each node in to make an allocation using the Max-Min Fairness algorithm.

Allocations are re computed every time a patriarch goes down or is kicked out due to byzantine failures, as well as every 24 hours. This ensures that bandwidth usage for the GNuggies hosts is not too high, while keeping the change frequent enough to reduce the impact of malicious nodes. During each allocation refresh, we make sure that only half the nodes (subject to math.ceil) are changed for every service. This is done to ensure that the service doesn't have downtime every time that allocation is changed and to ensure that a particular service is difficult to single out and attack as the nodes that are hosting it keep changing.

5.2 Using a GNuggies Clusters

Once Gnuggies Clients have been assigned to a Service, we will refer to them as GNuggies Clusters. Each node in the cluster receives the Docker image containing the code for the service and deploys it. The owner of the service specifies if the service is a stateless or a stateful one. In the case of a stateless service, the Docker image is deployed on each node except for on the leader (elected based on node age) which becomes a simple random load balancer.

In the case of a stateful service, the Docker image is deployed, however it is entirely up to the service owner to orchestrate their own Cluster using a simple python API on the communications module that gives information such as the address of nodes in the cluster, their age, and their compute power. They may also need to make sure that the leader of the cluster is the gateway to their service as this is the node that the GNuggies Name Servers will be pointing to. We realize that this setup is suboptimal, for stateful services however we did not have the time to figure out a better solution.

6. Evaluation

6.1 Time Taken to Cycle A Service

The time taken to cycle any individual service among different nodes, based off the Max-Min fairness algorithm, is dependent on both, the individual GNuggies nodes and the service itself. The time taken to cycle services impacts how quickly a service can be scaled in or scaled out, and also is representative of how much bandwidth is used by the GNuggies system in general. The size of the Docker container for the service as well as connection speeds on the GNuggies node are primary factors for this metric. Outside of varying download speeds, setting up a Docker container on the GNuggies node takes a few seconds once the image is compiled into a container. We ran experiments with multiple popular Docker images, adding scripts on each of them and testing time taken for startup and shutdown. Varying based on available system resources and the scripts tested, actual deployment time in each case always ended up being under 15 seconds.

6.2 Perceived Downtime Due to DNS Updates

Every time the nodes that are hosting a particular service are updated, the DNS records associated with the service must be updated in GNuggies DNS. This may cause perceived downtime in the service as it takes some time for DNS changes to propagate. DNS propagation was tested below.

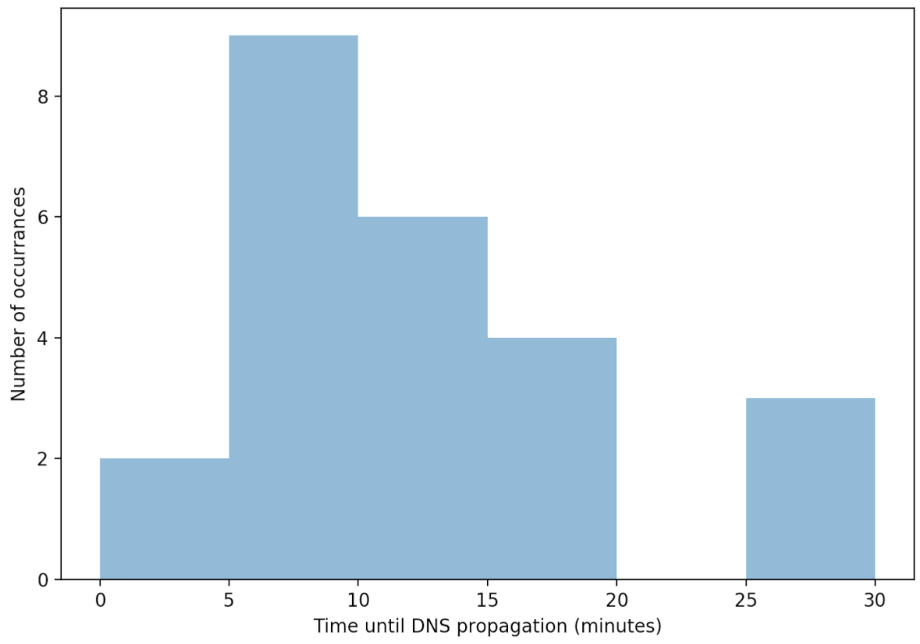

Figure 6.1

An experiment was conducted on Virtual Private Servers in many different geographical locations. We picked popular ISPs in that location and changed the DNS settings in the VPSs to query the DNS servers of those ISPs. We did this with 25 VPSs. Each VPS ran 10 trials and returned the mean propagation time. TTL was consistently set to 60 seconds. This shows the amount of time taken for a change in a DNS record be propagated. This represents the current downtime that a user may perceive if the leader node in a given service's cluster is changed.

7. Future Work

With regards to detection of untrusted nodes, it was mentioned that there is a trade off between removing individual adversarial nodes detected within a system and the entire mirror. The advantage of removing the whole group is that it penalizes attackers who provide many nodes to the system at once, however it can potentially shorten the wait time for other nodes to become trusted. Research into selection of the number of nodes per mirror, k, as well as the possibility of booting similar nodes based upon clusterings of nodes by absolute time in UTC that they enter the system and upon IP address might be valuable.

As mentioned in the evaluation in section 6, there is at least one case of need for an exception to the Min-Max Fairness allocation approach. For DNS, the propagation of IP addresses when DNS updates is very slow as ISPs routinely disregard TTL preferences. Future work should address these concerns as one can not quickly scale out the DNS service, which all other services within the system rely on.

This work does not address how to detect if nodes come from distinct machines or if several are provided by the same VM. Scaling up the number of nodes provided to the system by leveraging VMs would allow an attacker to reduce the cost of leveraging a civil attack requiring the attacker to gain a quorum of trusted nodes. Future work could address how to detect and reject these nodes, either based upon common IP addresses or other means.

The question of how to support and test nodes within stateful services is also not addressed. The design decision to test the same index of each mirror at once was intentional such that if nodes within different mirrors are mapped to the same roles, they can be directly compared. There is more work to be done in terms of ensuring consistency while testing stateful services across mirrors, ensuring each mirror is consistent at the time of test without alerting individual nodes to the presence of a test, etc.

Future work might also more thoroughly address the auditing of already trusted nodes to prevent an attacker from taking a majority of trusted nodes. In addition, sleeper cell nodes that wait to gain trust and then become malicious can be more easily frustrated if they are unable to determine if they are trusted nodes.

The scalability of the audit itself may be a question as a constant number of audit resources can only audit each node a number of times inverse to the number of nodes within the system. As it takes a logarithmic amount of time for nodes to become trusted w.r.t. the size of the system, this presents a security issue at scale if the scalability of the audit is not addressed as each untrusted node would be audited fewer times before becoming trusted.

8. Conclusion

We have created a system that is designed to operate almost entirely on untrusted nodes, that hosts services in a manner similar to IaaS solutions in such a manner that it minimizes liability for both, the owner of the service as well as for the owner of the nodes on which it runs. We have shown that this system is robust to faults as well as to malicious attacks. We have also done so in a manner that is completely self contained - all of the support services and the infrastructure for our system is hosted as part of the system itself.

References

[1] Satoshi Nakamoto “Bitcoin: A Peer-to-Peer Electronic Cash System”

[2] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. ”SETI@home: An experiment in public resource computing”. Communications of the ACM, Nov. 2002, Vol. 45 No. 11, pp. 56-61.

[3] James C. Corbett, Jeffrey Dean, et al. “Spanner: Google's Globally-Distributed Database”. Proceedings of OSDI'12: Tenth Symposium on Operating System Design and Implementation

[4] Juan Benet “IPFS - Content Addressed, Versioned, P2P File System”

[5] Roger Dingledine, Nick Mathewson, Paul Syverson “Tor: The Second-Generation Onion Router”

[6] Joao Neto, VinhTao, Vianney Rancurel “Managing Object Versioning in Geo-Distributed Object Storage Systems” Proceedings of the ACM 7th Workshop on Scientific Cloud Computing

[7] Bassam Zantout, Ramzi Haraty “I2P Data Communication System”. In the Proceedings of ICN 2011, The Tenth International Conference on Networks, St. Maarten, The Netherlands Antilles, January 2011.

[8] Oraclize (http://oraclize.it)

[9] Adam Gibson “TLSNotary” (https://github.com/tlsnotary/tlsnotary)

[10] Bram Cohen "BitTorrent-a new P2P app." *Yahoo eGroups* (2001).