Analyzing the Impact of Malicious Network Traffic: Final Report

Alexander M. Valentin

Kennesaw State University

CS 4632: Modeling and Simulation
Section W01
Instructor Christopher Regan

December 8, 2025

https://github.com/AMVGH/NetworkDDoSSimulation

Analyzing the Impact of Malicious Network Traffic: Final Report & Presentation

ALEXANDER M. VALENTIN, Kennesaw State University, USA

CCS Concepts: • **Networks → Network simulations**; **Network performance modeling**; *Network performance analysis*; *Network experimentation.*

Additional Key Words and Phrases: Distributed Denial of Service (DDoS), Botnet, Load-Balancing, Distributed Adaptive Routing, Bandwidth Exhaustion, Resource Depletion, Queuing Theory, Network Topology

ACM Reference Format:
Alexander M. Valentin. 2025. Analyzing the Impact of Malicious Network Traffic: Final Report & Presentation. *ACM Trans. Access. Comput.* 1, 1, Article 1 (September 2025), 31 pages.

1 Abstract

The aim of this project was to utilize a specialized discrete-event simulation in order to analyze the impact of malicious network traffic on network systems. The discrete-event simulation presented alongside this literature has been designed to analyze how Distributed Denial of Service attacks (DDoS) impact network metrics such as server response time, request throughput, and service availability. Findings indicate that since March of 2024, Distributed Denial of Service attacks are responsible for approximately 4,500 attacks per day, ranging from 100,000 to 1,000,000 packets-per-second. [10] These staggering metrics indicate a strong need for refined research surrounding the impact of malicious network traffic on enterprise service availability. The simulation approaches the problem domain by effectively modeling legitimate and malicious network traffic, servers and server responses, client networks, and network routing. In doing so, the simulation comprehensively captures how safeguards such as load-balancing contribute to reduced latency, improved throughput, and decreased server deterioration when under load. As outlined later in this literature, simulation outcomes indicate key findings surrounding relationships between attack surface area and network topology that contribute significantly to furthering understanding within the network domain. Not only are the findings outlined in this report relevant to the overall network domain and illustrate key relationships among network components, but the final simulation implementation serves as an educational tool surrounding malicious network traffic as a whole.

Author's Contact Information: Alexander M. Valentin, Kennesaw State University, Marietta, Georgia, USA.

2025. ACM 1936-7236/2025/9-ART1
https://doi.org/

2 Table of Contents

Contents

List of Figures

List of Tables

3 Introduction

To preface, the aim of this report is to effectively capture and refine all of the previously demonstrated findings surrounding the malicious network traffic simulation over the course of the semester into one comprehensive document. The motivation for exploring such a problem comes from the relevance of this area of study to the overall network system domain. Year after year, Distributed Denial of Service attacks continue to grow and target large enterprise services, with no indication of slowing down. According to Cloudflare's DDoS threat report for 2025 Q2, "Layer 3/Layer 4 (L3/4) DDoS attacks plunged 81% quarter-over-quarter to 3.2 million, while HTTP DDoS attacks rose 9% to 4.1 million. Year-over-year changes remain elevated. Overall attacks were 44% higher than 2024 Q2, with HTTP DDoS attacks seeing the largest increase of 129% YoY." [2] These findings indicate that there has not only been growth in Distributed Denial of Service attacks, but a fundamental shift in network attacking methodology among individuals who manage malicious distributed networks.

The objective of the final simulation implementation aims to effectively model network components such as legitimate and malicious network clients, networks of interconnected machines, network requests, network routers, and network servers. Furthermore, the final implementation captures core network interactions such as request generation, network traffic routing, server request processing and responses, and service availability impact under load.

The research conducted over the course of the past sixteen weeks aims to answer fundamental questions surrounding malicious network defense and overall attack traffic impact. By meeting the project implementation objectives outlined above, this enabled effective data collection highlighting key patterns among core configuration parameters and execution outcomes. Furthermore, by analyzing execution outcomes, the simulation comprehensively captures how safeguards such as load-balancing and increased network topology contribute to reduced latency, improved throughput, and decreased server deterioration under load.

Throughout this document, key information surrounding the network domain, supporting literature, simulation design and architecture, implementation decisions, simulation executions, execution results and analysis, and simulation validation and verification will be presented. In doing so, this report captures and illustrates the evolution of the project as time progressed, documents what was built versus what was proposed, presents key findings and insights surrounding simulation outcomes, and offers deep reflection on the development process as a whole. In all, this document intends to tell a complete story of the project's progression over the course of sixteen weeks, from the project's conception to final implementation and onward.

4 Background and Literature Review

Prior to beginning to implement such a simulation, core models and algorithms as well as supporting literature were established throughout Milestone 1, in order to provide a strong theoretical framework moving forward. Beginning with network routing, the academic paper *Routing Algorithms: A Review* presented by Ujjwal S., Vikas K. and Shubham K., under the guidance of Dr. Jayasheela, provided an in depth breakdown surrounding the algorithms that drive network traffic. As outlined in the paper, routing algorithms are divided into three distinct categories based on the way that they manage traffic: Adaptive Routing Algorithms (Isolated, Centralized, Distributed), Non-Adaptive Routing Algorithms (Flooding, Random Walk), and Hybrid Routing Algorithms (Link State, Distance Vector). The routing algorithms discussed throughout the paper were extremely imperative to the implementation of a successful malicious network traffic simulation, as the final implementation aimed to accurately model real-world network interactions. By utilizing academic research surrounding Adaptive Routing Algorithms, Non-Adaptive Routing Algorithms, and Hybrid Routing Algorithms, we were able to establish a strong theoretical baseline by which we could accurately-model real world traffic whilst analyzing a simulated system under load.

Moving on to attack architecture, Divya N. , Pooja W., Sanjay S. and Deepak K's academic paper, *BOTNET: Lifecycle, Architecture and Detection Model* provided significant insight surrounding the lifecycle and deployment of a Botnet. Furthermore, the writing outlined the different design architectures as well as implementations for such a network, as well as the different detection methods for malicious computer networks designed for Distributed Denial of Service attacks. According to the writing, there are five main design topologies when dealing with botnet architecture [6]:

- Star or Centralized C&C Topology: Relies on a single C&C resource to communicate with all bot agents. Each bot agent is issued new instructions directly from the central C&C point.
- Multi-Server Topology: A logical extension of the Star topology in which multiple servers are used to provide C&C instructions to bot agents, where the command systems communicate amongst each other to manage the botnet.
- Hierarchical C&C Topology: A Bot Master communicates with a central botnet C&C Server which send instructions propagating down a layered hierarchy of connected agents.
- Peer-to-Peer Topology: Each host periodically connects to its neighbor to retrieve orders from the Botmaster. The Botmaster only needs to connect to one of the Bots (peer) to send his commands all over the network.
- Unstructured or Random C&C Topology: Each bot has the ability to scan the internet in order to find another Bot.

The main purpose of this paper with regard to the final malicious network simulation was the implementation of an attacking network designed to emulate a real-world DDoS attack. By utilizing the academic paper to gather information surrounding attack network topology and architecture, we could ensure that the final simulated attack network closely resembles what is expected under real-world conditions, preserving the validity and credibility of our overall simulation approach.

Botnets Multiply and Level Up is a DDoS threat intelligence report authored by Chris Conrad and published by NETSCOUT SYSTEMS, INC. in early 2022. The report provided details regarding recent growth in the botnet landscape, the impact this technology has had on organizations, and provided information about prevalent botnets. The report lists botnets such as Mirai, Meris, Dvinis, and Killnet, and lists important information such as their deployment date, notable targets, attack capabilities, attack vectors, attack surface, and node size. This information was extremely important in terms of designing the malicious network traffic simulation, as it provided a reference point in terms of the capabilities and size of real-world attack networks. According to the writing, at its height Meris was estimated to contain an estimated 250,000 compromised devices, allowing the botnet to execute attacks consisting of 2 million requests per second. By establishing a quantitative comparative baseline by which real attack networks operate, we were able to capture and accurately model the capabilities for the attacking network within the simulation implementation without worry of either under or overestimating its surface area or attack capabilities.

Beyond network routing, attack architecture, and Botnet attack capabilities, William Stalling's *Queuing Analysis* is an academic report detailing the different queuing models utilized throughout computer and network analysis. Specifically, Stalling makes mention of Single-Server Queues, Multiserver Queues, networks of Queues, as well as other Queuing models. This resource was of significant assistance over the course of initial development, as it helped determine how to come to an accurate algorithm in terms of estimating average response time and utilization of servers within the simulation. Furthermore, Stalling's research played a significant role in guiding the implementation for request processing and ultimately laid the groundwork for the initial network server design.

While the supporting literature outlined above provided a strong theoretical foundation whilst transitioning into initial development, *DDoS Attack and Detection Methods in Internet-Enabled Networks* by Kazeem A., Adnan

A., and Anish M. detailed the taxonomy and architecture of both centralized and decentralized DDoS attack, as well as the core mathematical models utilized to calculate the following:

Probability of Bandwidth Exhaustion:

$$P_B = \frac{(a^c)}{\sum_{i=0}^{e} \frac{a^i}{i!}} \tag{1}$$

Probability of Depletion of Victim Resources:

$$P_{TA} = 1 - (1 - P_B)(1 - P_M) \tag{2}$$

Probability of Successful Attack:

$$P_{wa} = \begin{cases} P = \lim_{t \to \infty} \frac{X_t}{t \times C} \\ P_{wa} = \overline{P} \end{cases} \tag{3}$$

These concepts and mathematical models were imperative to the successful implementation of a malicious network traffic simulation, as the writing contained details pertaining to proper attack architecture that was relevant to modeling how a real-world attack is performed.

In addition, the following mathematical models outlined in the Milestone 4 guidelines were utilized whilst analyzing simulation outcomes in order to comprehensively assess parameter sensitivity and outcome confidence:

Parameter Sensitivity:

$$\text{Parameter Sensitivity} = \frac{\text{\% Change in Output}}{\text{\% Change in Input}} \tag{4}$$

Outcome Confidence:

$$\text{Confidence Interval} = \bar{x} \pm 1.96 \times \frac{s}{\sqrt{n}} \tag{5}$$

These pieces of literature alongside the mathematical formulas provided a strong source of justification for the final implementation approach, as the architecture, core mathematical models, core algorithms, and key interactions illustrated in the final simulation were directly built upon and extrapolated from the findings outlined in the supporting literature. In all, the academic literature outlined above provided an extremely strong theoretical foundation whilst transitioning into initial development.

5 Model Design and Architecture

5.1 Conceptual Model Explanation

Key models of this simulation as well as the core architectural decisions behind their conceptual design are as follows:

- *SimulationExecutive:* Class designed to house all the entities of the simulation and run the simulation. The implementation remains almost identical to the implementation outlined in the proposal, with the only deviation being simulationDuration is now one of the parameters included in config.py.
- *DataHandler (Renamed from DataCollector):* Class designed to be an all encompassing body for data collection, calculation, manipulation, and exporting, as opposed to the single collection device outlined in Milestone 1. The class attributes captures all the behavior outlined by the attributes in Milestone 1, and expands the functionality of the class substantially. The final implementation added new attributes to allocate functionality for collecting time-series metrics, data calculation, data manipulation, and data exporting.

- *BaseNetworkClient:* Base class that LegitimateNetworkClient and MaliciousNetworkClient derive from. The class successfully captures the core functionality and behavior outlined in the initial proposal. The only deviation from the initial UML Class Diagram is the addition of traffic_type to house the type of traffic that will be sent from the client. Furthermore, the following fields have been added for client-level request tracking: requests_sent, successful_response, and no_response.
- *LegitimateNetworkClient:* Inherits everything from BaseNetworkClient, no differences exist between implementation and proposal.
- *MaliciousNetworkClient:* Inherits everything from BaseNetworkClient, isEnabled and the associated Getters and Setters outlined in the initial proposal have been discarded in the final implementation.
- *BaseNetworkModel (New Class):* Base class that both Botnet and LegitimateTrafficNetwork derive from. At their core Botnet and LegitimateTrafficNetwork exhibit the same behavior, so there was no need to rewrite the same function multiple times in two separate classes. Properties in the base class are as follows: env, network_type, client_count, request_rate, target_network, load_size_lower, load_size_upper, and network_clients. The attributes attackStart, attackDuration, and attackProcess have been removed from the final implementation, while the base class properly captures the behavior of what was functionally described for the two child classes, there was no need for these fields due to the nature of the SimPy library and how processes are managed.
- *Botnet:* Class designed to model an attacking network, with infected machines inside the attacking network that will be send requests to a predefined victim network. Botnet inherits everything from the parent BaseNetworkModel class.
- *LegitimateTrafficNetwork:* Class designed to model a network of interconnected, uninfected machines. Legitimate machines inside the network will send requests to a predefined target network at a fixed rate. LegitimateTrafficNetwork inherits everything from the parent BaseNetworkModel class.
- *Network:* Class designed to model a network of servers. The class is extremely similar to the design outlined in the UML Class Diagram provided in initial proposal except for the addition of two new fields, dropped_no_server_available and incoming_request_count. These fields were added in order to keep a count of how many requests hit the network and how many requests were dropped from the network. This was done in order to resolve issues with duplicate request_ids, as when the request_id was generated by the client there were duplicate Request IDs (RequestID: 1, 2, 3, ...) with different origins. Now the network "stamps" each request_id with the number in incoming_request_count in order to resolve any issues with conflicting ids. Furthermore, the dropped_no_server_available field was added to the Network class to better model how real world networks track dropped requests and to avoid confusion when assessing simulation results. Now, as opposed to seeing 30,000 dropped requests each from 10 different servers during an outage (in a simulation run with only 60,000 requests total), the dropped requests no longer propagate through all the offline servers, and instead there is one value for if a request is dropped as a result of a server/servers being offline. These architectural changes better capture what actually occurred during the simulation run.
- *NetworkRouter:* Class designed to model a network router to route incoming traffic to servers within the network. No changes were made from the proposal besides the removal of the currentAlgorithm field. Since only one refined algorithm for request routing was implemented, there was no need to keep this field included in the class.
- *NetworkServer:* Class designed to model a network server and process incoming requests, the processDelay field has been removed by virtue of how requests are handled using the SimPy library. In the final implementation, servers utilize timeouts according to the request load size and server utilization in order to calculate processing time. Furthermore, the resourceCapacity field has been changed to processing_power to better capture its usage in the context of the simulation. Many fields for collecting

server metrics and driving simulation logic have been added in order to help facilitate data collection and to make the simulation more accurate and robust. These fields include: max_requests_concurrent, max_request_queue_len, cpu_utilization, process_queue_utilization, server_health, time_spent_offline, is_server_online, increased_utilization, high_utilization, critical_utilization, request_process_worker, request_queue, total_requests_received, total_requests_processed, dropped_request_queue_full, dropped_requests_timeout, and dropped_requests_high_load.

- *Request:* Class designed to represent a request sent to the target network. The class still maintains fields outlined in the initial proposal such as source_id, traffic_type, load_size, arrival_time, and served_time. As stated previously, request_id was removed from the request class due to a bug with duplicate IDs. Additional fields have been added to the class to help support simulation functionality such as: is_served, is_routed, seen_by, and an on_completion callback.
- *ConfigValidator (New Class):* Validates the parameters provided in config.py to ensure that all user-configured parameters are reasonable and will not cause errors.
- *GenericEnums(New Class):* Houses miscellaneous values such as non-configurable weights and ranges in order to reduce the presence of 'magic' numbers throughout the solution and make the simulation more robust.
- *DataPlotter (NewClass):* Houses all data visualization mechanisms and builds graphics concerning simulation outcome metrics.
- *ProbabilityEngine (NewClass):* Implements the probability algorithms and relevant calculations for Probability of Bandwidth Exhaustion, Probability of Depletion of Victim Resources, and Probability of Successful Attack.
- *TestingEngine (NewClass):* Houses all relevant testing functionality.

5.2 UML Diagrams

As illustrated in the previous section, many core architectural and design changes have occurred between initial conceptual ideation and final implementation, as such, the following UML diagrams reflect the core architectural changes that have occurred since the initial proposal:

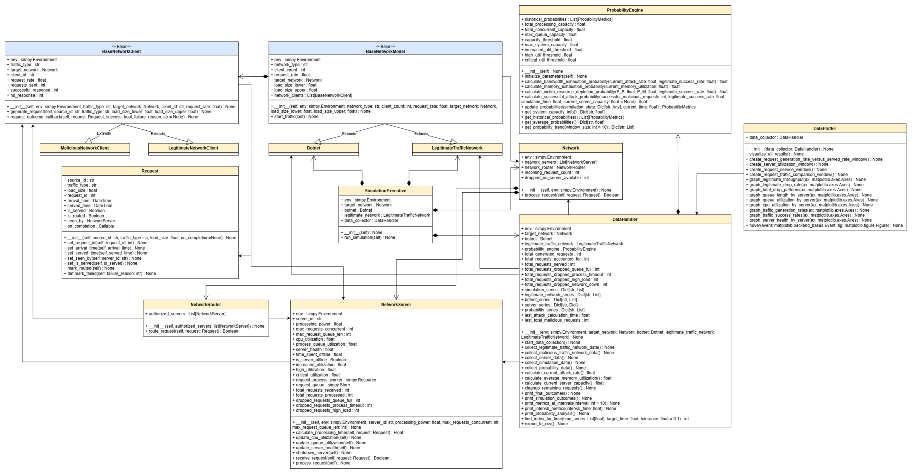


Fig. 1. UML Class Diagram visualizing the main network simulation classes and relationships alongside key attributes, methods, inheritance, and composition.

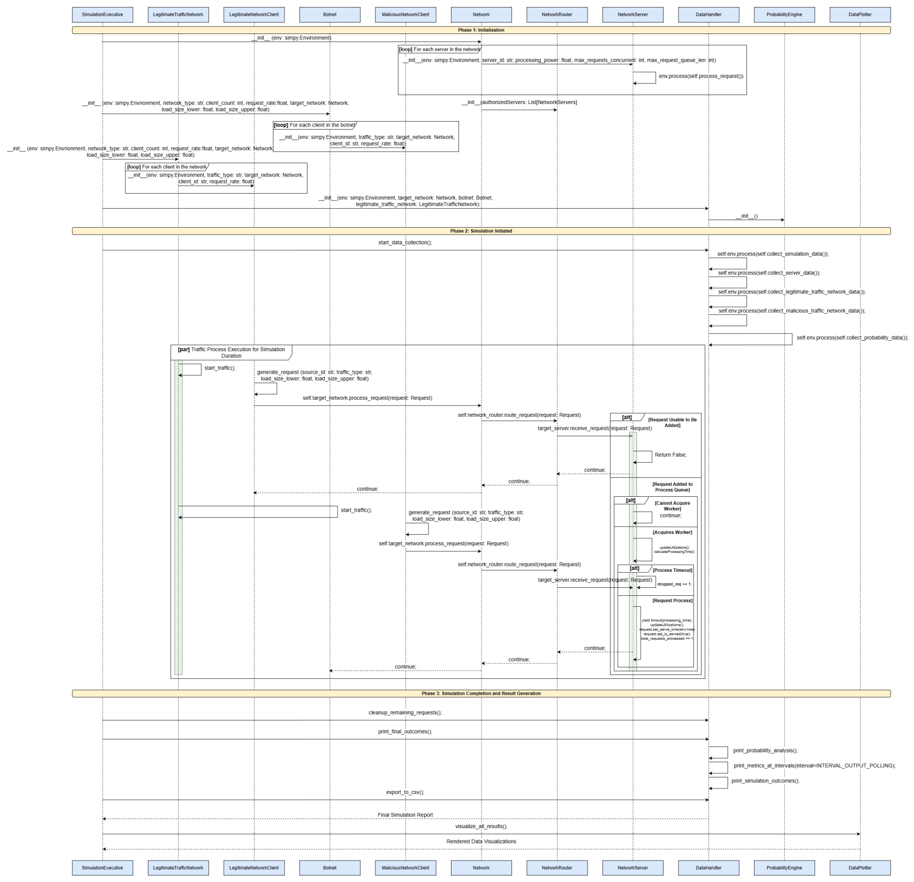

SimulationExecutive
LegitimateTrafficNetwork
LegitimateNetworkClient
Botnet
MaliciousNetworkClient
Network
NetworkRouter
NetworkServer
DataHandler
ProbabilityEngine
DataPlotter
Phase 1: Initialization
__init__ (env: simpy.Environment)
[loop] For each server in the network
__init__(env: simpy.Environment, server_id: str, processing_power: float, max_requests_concurrent: int, max_request_queue_len: int)
env.process(self.process_request())
__init__ (env: simpy.Envrionment, network_type: str, client_count: int, request_rate:float, target_network: Network, load_size_lower: float, load_size_upper: float)
__init__(authorizedServers: List[NetworkServers]
[loop] For each client in the botnet
__init__(env: simpy.Environment, traffic_type: str, target_network: Network, client_id: str, request_rate: float)
__init__ (env: simpy.Envrionment, network_type: str, client_count: int, request_rate:float, target_network: Network, load_size_lower: float, load_size_upper: float)
[loop] For each client in the network
__init__(env: simpy.Environment, traffic_type: str, target_network: Network, client_id: str, request_rate: float)
__init__(env: simpy.Environment, target_network: Network, botnet: Botnet, legitimate_traffic_network: LegitimateTrafficNetwork):
__init__()
Phase 2: Simulation Initiated
start_data_collection();
self.env.process(self.collect_simulation_data());
self.env.process(self.collect_server_data());
self.env.process(self.collect_legitimate_traffic_network_data());
self.env.process(self.collect_malicious_traffic_network_data());
self.env.process(self.collect_probability_data());
[par] Traffic Process Execution for Simulation Duration
start_traffic();
generate_request (source_id: str, traffic_type: str, load_size_lower: float, load_size_upper: float)
self.target_network.process_request(request: Request)
self.network_router.route_request(request: Request)
[alt]
[Request Unable to Be Added]
target_server.receive_request(request: Request)
Return False;
continue;
continue;
continue;
[Request Added to Process Queue]
[alt]
[Cannot Acquire Worker]
continue;
start_traffic();
generate_request (source_id: str, traffic_type: str, load_size_lower: float, load_size_upper: float)
[Acquires Worker]
updateUtilizations();
calculateProcessingTime();
self.target_network.process_request(request: Request)
self.network_router.route_request(request: Request)
[alt]
[Process Timeout]
target_server.receive_request(request: Request)
dropped_req += 1;
[Request Process]
yield timout(processing_time);
updateUtilizations();
request.set_serve_time(env.now)
request.set_is_served(true);
total_requests_processed += 1
continue;
continue;
continue;
Phase 3: Simulation Completion and Result Generation
cleanup_remaining_requests();
print_final_outcomes();
print_probability_analysis();
print_metrics_at_intervals(interval=INTERVAL_OUTPUT_POLLING);
print_simulation_outcomes();
export_to_csv();
Final Simulation Report
visualize_all_results();
Rendered Data Visualizations

Fig. 2. UML Sequence Diagram illustrating core interactions within the final implementation of the malicious network traffic simulation.

While this document has been submitted as a PDF in order to preserve image clarity, if there are any issues visualizing the figures please visit https://github.com/AMVGH/FinalReportSupportingImages for the full resolution images.

5.3 Key Design Decisions and Implementation Details

The biggest challenge faced throughout the course of implementing the final network simulation came in the form of architectural design. Unfortunately, there were instances over the course of development where I had realized that the architectural design I had initially outlined in the UML Class Diagram in Milestone 1 was either ineffective in capturing the behavior I wished to emulate, or could be further refined and was not optimal for what I was trying to achieve.

To illustrate, the initial server design outlined in Milestone 1, whilst conceptualized around academic literature, still contained inaccuracies in terms of processing and architecture. Specifically, real-world network environments don't utilize a formalized 'resourceCapacity' attribute in order to determine processing ability. These issues, however, did not impede my ability to implement the system I wished to emulate, and instead caused me to reevaluate design choices and improve the class models as a whole. In order to ensure that the final implementation was as refined and as accurate as possible, extensive additional research was conducted throughout the course of development surrounding network design and server processing to assist in making architectural decisions.

Continuing on the server's initial design, additional research led to the understanding that the processing capabilities of a network server were directly attributed to things such as the Central Processing Unit's architecture and functional attributes (Clock Speed, Number of Cores, etc.), Amount of Random Access Memory, Number of Storage Drives, Motherboard Architecture, Power Supply Unit Capabilities, Network Interface Card Availability, as well as other technical considerations [4]. While the final implementation does not implement some of the technical considerations in their entirety due to simulation limitations (As referenced in Section 9 of this literature), it is clear that the final implementation is much more derivative of real world network infrastructure than previously outlined. Ultimately, it was the recognition of possible lapses in initial design conceptualization that led to additional research for design and implementation decisions.

6 Implementation Details

6.1 Technologies Used

The final implementation of the malicious network traffic simulation is built using Python Version 3.x due to familiarity and previous experience with the language. Furthermore, Python offers extensive support for numerous libraries tailored to discrete-event simulations, allowing for robust and finely tuned test simulations that truly capture the nuances surrounding network design.

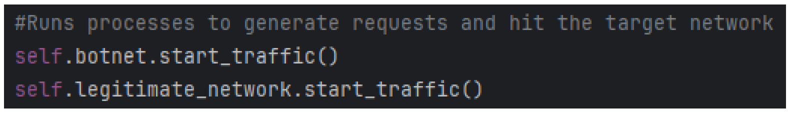
The following tools, libraries, and frameworks have been utilized in order to implement all of the necessary functionality seen in the final implementation:

- Development Environment: PyCharm
- Source Control: Git, GitHub
- Simulation Tooling: SimPy [5]
- Data Visualization: Matplotlib [11]
- Testing: PyTest [7]

6.2 Core Algorithms

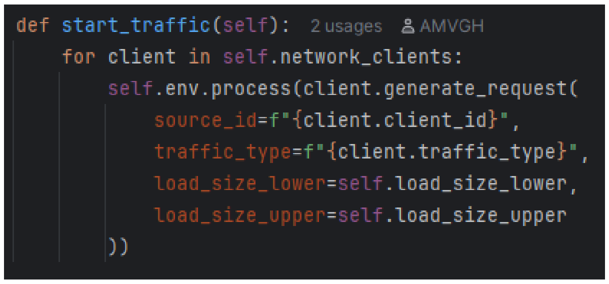
Despite numerous architectural and design changes occurring over the course of the simulation's development, the system dynamics and key component interactions outlined in Section 2.2 of Milestone 1 have not changed. As such, the core algorithms of the final implementation are those that answer targeted research questions as well as capture the initial component behaviors outlined in the project's proposal.

The first important core algorithm in this simulation is the traffic generation algorithm utilized to model targeted and legitimate network traffic flow. This algorithm is housed in the BaseNetworkModel class, the parent class of both the LegitimateTrafficNetwork and the Botnet classes. The idea behind this design, as communicated in Milestone 1, is that instructions would be issued to both the LegitimateTrafficNetwork and the Botnet defining the target network, request rate, and load size boundaries. Upon traffic being initiated by the housing networks, a SimPy process is started in which the network client begins to generate requests according to the instructions. Attached are excerpts from the code base that capture this component interaction.



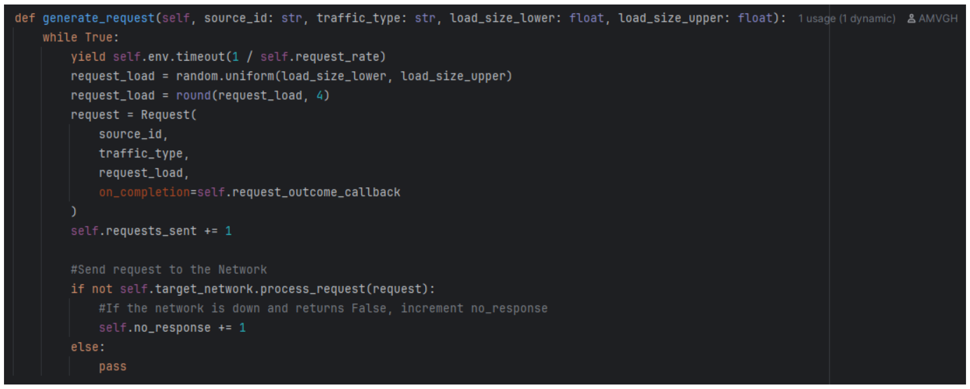
```
#Runs processes to generate requests and hit the target network
self.botnet.start_traffic()
self.legitimate_network.start_traffic()
```

Fig. 3. Code excerpt illustrating how housing networks initiate traffic generation.



```
def start_traffic(self):  2 usages  AMVGH
    for client in self.network_clients:
        self.env.process(client.generate_request(
            source_id=f"{client.client_id}",
            traffic_type=f"{client.traffic_type}",
            load_size_lower=self.load_size_lower,
            load_size_upper=self.load_size_upper
        ))
```

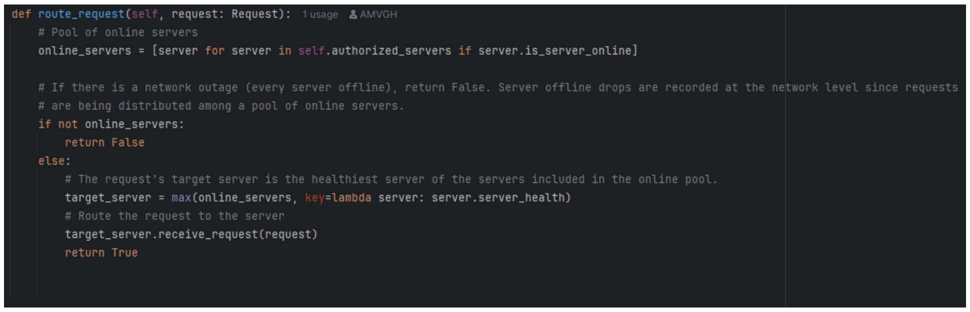
Fig. 4. Code excerpt demonstrating how the client request generation process is initiated.



```
def generate_request(self, source_id: str, traffic_type: str, load_size_lower: float, load_size_upper: float):  1 usage (1 dynamic)  AMVGH
    while True:
        yield self.env.timeout(1 / self.request_rate)
        request_load = random.uniform(load_size_lower, load_size_upper)
        request_load = round(request_load, 4)
        request = Request(
            source_id,
            traffic_type,
            request_load,
            on_completion=self.request_outcome_callback
        )
        self.requests_sent += 1

        #Send request to the Network
        if not self.target_network.process_request(request):
            #If the network is down and returns False, increment no_response
            self.no_response += 1
        else:
            pass
```

Fig. 5. Code excerpt illustrating how network clients generate and send requests to the target network.

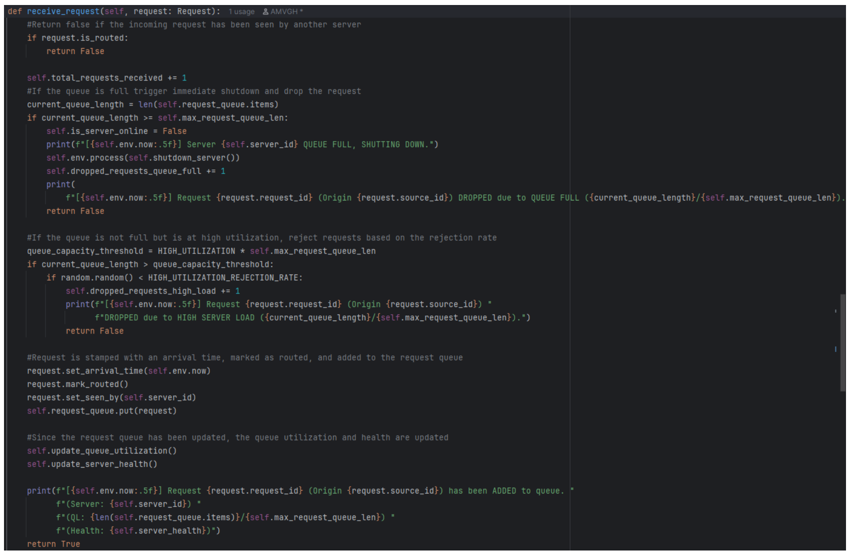
As communicated in the System Dynamics portion of Milestone 1, once requests arrive at the target network, they are directed to a network router that then determines how client requests are directed to available servers in the network. This routing algorithm is another core algorithm that captures system dynamics and drives research questions, as the algorithm not only routes traffic to corresponding network servers, but effectively balances traffic among the available servers using network health metrics.



```
def route_request(self, request: Request):  1 usage  AMVGH
    # Pool of online servers
    online_servers = [server for server in self.authorized_servers if server.is_server_online]

    # If there is a network outage (every server offline), return False. Server offline drops are recorded at the network level since requests
    # are being distributed among a pool of online servers.
    if not online_servers:
        return False
    else:
        # The request's target server is the healthiest server of the servers included in the online pool.
        target_server = max(online_servers, key=lambda server: server.server_health)
        # Route the request to the server
        target_server.receive_request(request)
        return True
```

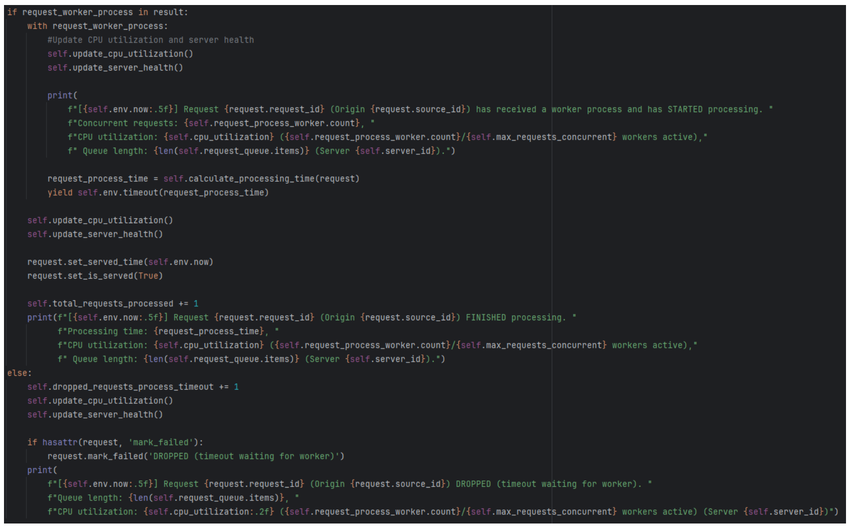
Fig. 6. Code excerpt demonstrating how the network router balances and routes network traffic.

Once a request arrives at a server, the request is added to a processing queue where different process workers will retrieve pending requests, service them, and return the expected results to the client that requested the resource. This is the heart of the simulation and one of the most important component interactions within the simulation, as this behavior is directly involved in servicing and managing the overall throughput of incoming requests. This algorithmic interaction of receiving and processing requests can be observed as follows.



```
def receive_request(self, request: Request):  1 usage  AMVGH *
    #Return false if the incoming request has been seen by another server
    if request.is_routed:
        return False

    self.total_requests_received += 1
    #If the queue is full trigger immediate shutdown and drop the request
    current_queue_length = len(self.request_queue.items)
    if current_queue_length >= self.max_request_queue_len:
        self.is_server_online = False
        print(f"[{self.env.now:.5f}] Server {self.server_id} QUEUE FULL, SHUTTING DOWN.")
        self.env.process(self.shutdown_server())
        self.dropped_requests_queue_full += 1
        print(
            f"[{self.env.now:.5f}] Request {request.request_id} (Origin {request.source_id}) DROPPED due to QUEUE FULL ({current_queue_length}/{self.max_request_queue_len}).
        return False

    #If the queue is not full but is at high utilization, reject requests based on the rejection rate
    queue_capacity_threshold = HIGH_UTILIZATION * self.max_request_queue_len
    if current_queue_length > queue_capacity_threshold:
        if random.random() < HIGH_UTILIZATION_REJECTION_RATE:
            self.dropped_requests_high_load += 1
            print(f"[{self.env.now:.5f}] Request {request.request_id} (Origin {request.source_id}) "
                  f"DROPPED due to HIGH SERVER LOAD ({current_queue_length}/{self.max_request_queue_len}).")
            return False

    #Request is stamped with an arrival time, marked as routed, and added to the request queue
    request.set_arrival_time(self.env.now)
    request.mark_routed()
    request.set_seen_by(self.server_id)
    self.request_queue.put(request)

    #Since the request queue has been updated, the queue utilization and health are updated
    self.update_queue_utilization()
    self.update_server_health()

    print(f"[{self.env.now:.5f}] Request {request.request_id} (Origin {request.source_id}) has been ADDED to queue. "
          f"(Server: {self.server_id}) "
          f"(QL: {len(self.request_queue.items)}/{self.max_request_queue_len}) "
          f"(Health: {self.server_health})")
    return True
```

Fig. 7. Code excerpt visualizing how network servers receive incoming requests.



```
if request_worker_process in result:
    with request_worker_process:
        #Update CPU utilization and server health
        self.update_cpu_utilization()
        self.update_server_health()

        print(
            f"[{self.env.now:.5f}] Request {request.request_id} (Origin {request.source_id}) has received a worker process and has STARTED processing. "
            f"Concurrent requests: {self.request_process_worker.count}, "
            f"CPU utilization: {self.cpu_utilization} ({self.request_process_worker.count}/{self.max_requests_concurrent} workers active),"
            f" Queue length: {len(self.request_queue.items)} (Server {self.server_id}).")

        request_process_time = self.calculate_processing_time(request)
        yield self.env.timeout(request_process_time)

    self.update_cpu_utilization()
    self.update_server_health()

    request.set_served_time(self.env.now)
    request.set_is_served(True)

    self.total_requests_processed += 1
    print(f"[{self.env.now:.5f}] Request {request.request_id} (Origin {request.source_id}) FINISHED processing. "
          f"Processing time: {request_process_time}, "
          f"CPU utilization: {self.cpu_utilization} ({self.request_process_worker.count}/{self.max_requests_concurrent} workers active),"
          f" Queue length: {len(self.request_queue.items)} (Server {self.server_id}).")
else:
    self.dropped_requests_process_timeout += 1
    self.update_cpu_utilization()
    self.update_server_health()

    if hasattr(request, 'mark_failed'):
        request.mark_failed('DROPPED (timeout waiting for worker)')
    print(
        f"[{self.env.now:.5f}] Request {request.request_id} (Origin {request.source_id}) DROPPED (timeout waiting for worker). "
        f"Queue length: {len(self.request_queue.items)}, "
        f"CPU utilization: {self.cpu_utilization:.2f} ({self.request_process_worker.count}/{self.max_requests_concurrent} workers active) (Server {self.server_id})")
```

Fig. 8. Code excerpt demonstrating once a worker is afforded, how a request is processed.

Capturing these core component interactions was important as they allow for emergent behaviors to arise over time as the system undergoes continued load from the attacking network. Since the attacking network aims to exhaust resources from the network servers, by algorithmically capturing these core network behaviors the simulation successfully illustrates how component interactions impede the server's ability to process legitimate requests.

6.3 Data Structures

The final simulation implementation is heavily characterized by the use of lists as means of storage and tailored dictionaries for comprehensive data management. This is observed throughout the entire project solution, as lists are utilized to store network clients, servers within the target network, authorized network routing locations, and are even used in intermediate data calculations. Given the functional requirements of the system outlined in Milestone 1, it was not necessary to use other advanced means of storage, as the biggest consideration when storing data elements throughout the project solution was ease of iteration. By virtue of how the SimPy library works, I also found that lists worked extremely well for managing processes, as the details of an individual process worker were not important since the final simulation is designed to run for a fixed duration uninterrupted. In addition to list utilization, dictionaries are heavily utilized throughout the project solution, specifically with regards to data management. Given the level of granularity execution outcomes were required to achieve, as well as the multiple dimensions of data that were necessary for analysis, dictionaries seemed like the most robust solution for storing and managing execution outcomes.

6.4 Challenges Overcome

Although I had chosen Python due to my previous familiarity with the language and expansive support for libraries tailored for discrete-event simulations, the biggest challenge was ultimately utilizing these libraries effectively in order to capture the behavior I desired. While I was familiar with Python, I was unfamiliar with tools such as SimPy, PyTest, and Matplotlib. A lot of frustration came from designating a lot of time for a core feature dependent on these tools, only to run into fatal errors and unexpected behaviors. While I am very pleased with the final simulation implementation and have learned a lot over the course of its development, I recognize that I was likely very inefficient in doing-so as a large part of my time was spent reading supporting documentation or browsing forums in order to better understand the packages and assist in debugging.

7 Experimental Setup

7.1 Simulation Parameters

The final implementation takes an extensive list of parameters, allowing the user to change and configure core simulation pipeline values, network infrastructure values, server health and utilization weights, and data output granularity. By implementing such an expansive list of configurable values, the user is afforded complete and total control over the simulated environment. The simulation parameters are as follows:

- SIMULATION_DURATION
- INTERVAL_DATA_POLLING
- INTERVAL_OUTPUT_POLLING
- NUM_SERVERS
- REQUEST_TIMEOUT
- SERVER_TIMEOUT
- PROCESSING_POWER
- MAX_REQUESTS_CONCURRENT
- MAX_REQUEST_QUEUE_LENGTH
- CPU_UTILIZATION_HEALTH_WEIGHT
- QUEUE_UTILIZATION_HEALTH_WEIGHT
- OFFLINE_CLEAR_THRESHOLD
- HIGH_UTILIZATION_REJECTION_RATE
- INCREASED_UTILIZATION
- HIGH_UTILIZATION
- CRITICAL_UTILIZATION

- LEGITIMATE_TRAFFIC_RATE
- LEGITIMATE_CLIENT_COUNT
- LEGITIMATE_LOAD_SIZE_LOWER
- LEGITIMATE_LOAD_SIZE_UPPER
- MALICIOUS_TRAFFIC_RATE
- MALICIOUS_CLIENT_COUNT
- MALICIOUS_LOAD_SIZE_LOWER
- MALICIOUS_LOAD_SIZE_UPPER

7.2 Scenarios Tested

Over the course of Milestones 3 and 4, a total of twenty-four unique simulation scenarios were tested in order to capture nuanced aspects of system performance capability, parameter analysis, and real-world network behavioral comparison. To illustrate, over the course of Milestone 3, ten distinct simulation executions were run to specifically capture and test the system under distinct usage scenarios. Across these scenarios, user configurable parameters such as CPU utilization health weight (0.4), queue utilization health weight (0.6), increased utilization threshold (0.70), high utilization threshold (0.85), critical utilization threshold (0.95), simulation duration (900), interval data polling (5), and interval output polling (30), were all kept the same in order to preserve the same level of data granularity as well as maintain consistent calculation weights. Scenarios outlined in Milestone 3 include:

Run 1: Baseline Configuration

- NUM_SERVERS: 8
- SERVER_TIMEOUT: 2
- PROCESSING_POWER: 250
- REQUEST_TIMEOUT: 3
- MAX_REQUESTS_CONCURRENT: 25
- MAX_REQUEST_QUEUE_LENGTH: 400
- OFFLINE_CLEAR_THRESHOLD: 0.5
- HIGH_UTILIZATION_REJECTION_RATE: 0.15
- LEGITIMATE_TRAFFIC_RATE: 2
- LEGITIMATE_CLIENT_COUNT: 400
- LEGITIMATE_LOAD_SIZE_LOWER: 1
- LEGITIMATE_LOAD_SIZE_UPPER: 3
- MALICIOUS_TRAFFIC_RATE: 15
- MALICIOUS_CLIENT_COUNT: 80
- MALICIOUS_LOAD_SIZE_LOWER: 12
- MALICIOUS_LOAD_SIZE_UPPER: 20

Run 2: Intense HTTP Flood

- NUM_SERVERS: 8
- SERVER_TIMEOUT: 2
- PROCESSING_POWER: 250
- REQUEST_TIMEOUT: 8
- MAX_REQUESTS_CONCURRENT: 25
- MAX_REQUEST_QUEUE_LENGTH: 400
- OFFLINE_CLEAR_THRESHOLD: 0.5
- HIGH_UTILIZATION_REJECTION_RATE: 0.15
- LEGITIMATE_TRAFFIC_RATE: 2
- LEGITIMATE_CLIENT_COUNT: 400
- LEGITIMATE_LOAD_SIZE_LOWER: 1
- LEGITIMATE_LOAD_SIZE_UPPER: 3
- MALICIOUS_TRAFFIC_RATE: 40
- MALICIOUS_CLIENT_COUNT: 150
- MALICIOUS_LOAD_SIZE_LOWER: 12
- MALICIOUS_LOAD_SIZE_UPPER: 20

Run 3: High Legitimate Traffic

- NUM_SERVERS: 8
- SERVER_TIMEOUT: 2
- PROCESSING_POWER: 250
- REQUEST_TIMEOUT: 3
- MAX_REQUESTS_CONCURRENT: 25
- MAX_REQUEST_QUEUE_LENGTH: 400
- OFFLINE_CLEAR_THRESHOLD: 0.5
- HIGH_UTILIZATION_REJECTION_RATE: 0.15
- LEGITIMATE_TRAFFIC_RATE: 4
- LEGITIMATE_CLIENT_COUNT: 1000
- LEGITIMATE_LOAD_SIZE_LOWER: 1
- LEGITIMATE_LOAD_SIZE_UPPER: 3
- MALICIOUS_TRAFFIC_RATE: 15
- MALICIOUS_CLIENT_COUNT: 80
- MALICIOUS_LOAD_SIZE_LOWER: 12
- MALICIOUS_LOAD_SIZE_UPPER: 20

Run 4: Mixed Workload Stress

- NUM_SERVERS: 8
- SERVER_TIMEOUT: 2
- PROCESSING_POWER: 250
- REQUEST_TIMEOUT: 3
- MAX_REQUESTS_CONCURRENT: 25
- MAX_REQUEST_QUEUE_LENGTH: 400
- OFFLINE_CLEAR_THRESHOLD: 0.5
- HIGH_UTILIZATION_REJECTION_RATE: 0.15
- LEGITIMATE_TRAFFIC_RATE: 3
- LEGITIMATE_CLIENT_COUNT: 700
- LEGITIMATE_LOAD_SIZE_LOWER: 1
- LEGITIMATE_LOAD_SIZE_UPPER: 3
- MALICIOUS_TRAFFIC_RATE: 25
- MALICIOUS_CLIENT_COUNT: 120
- MALICIOUS_LOAD_SIZE_LOWER: 12
- MALICIOUS_LOAD_SIZE_UPPER: 20

Run 5: Load Balancing Pressure

- NUM_SERVERS: 12
- SERVER_TIMEOUT: 2
- PROCESSING_POWER: 300
- REQUEST_TIMEOUT: 3
- MAX_REQUESTS_CONCURRENT: 35
- MAX_REQUEST_QUEUE_LENGTH: 600
- OFFLINE_CLEAR_THRESHOLD: 0.5
- HIGH_UTILIZATION_REJECTION_RATE: 0.15
- LEGITIMATE_TRAFFIC_RATE: 2
- LEGITIMATE_CLIENT_COUNT: 400
- LEGITIMATE_LOAD_SIZE_LOWER: 1
- LEGITIMATE_LOAD_SIZE_UPPER: 3
- MALICIOUS_TRAFFIC_RATE: 15
- MALICIOUS_CLIENT_COUNT: 80
- MALICIOUS_LOAD_SIZE_LOWER: 12
- MALICIOUS_LOAD_SIZE_UPPER: 20

Run 6: High Server Outage

- NUM_SERVERS: 5
- SERVER_TIMEOUT: 2
- PROCESSING_POWER: 180
- REQUEST_TIMEOUT: 3
- MAX_REQUESTS_CONCURRENT: 20
- MAX_REQUEST_QUEUE_LENGTH: 300
- OFFLINE_CLEAR_THRESHOLD: 0.5
- HIGH_UTILIZATION_REJECTION_RATE: 0.15
- LEGITIMATE_TRAFFIC_RATE: 2
- LEGITIMATE_CLIENT_COUNT: 400
- LEGITIMATE_LOAD_SIZE_LOWER: 1
- LEGITIMATE_LOAD_SIZE_UPPER: 3
- MALICIOUS_TRAFFIC_RATE: 15
- MALICIOUS_CLIENT_COUNT: 80
- MALICIOUS_LOAD_SIZE_LOWER: 12
- MALICIOUS_LOAD_SIZE_UPPER: 20

Run 7: Moderate Server Downtime

- NUM_SERVERS: 6
- SERVER_TIMEOUT: 2
- PROCESSING_POWER: 200
- REQUEST_TIMEOUT: 3
- MAX_REQUESTS_CONCURRENT: 22
- MAX_REQUEST_QUEUE_LENGTH: 350
- OFFLINE_CLEAR_THRESHOLD: 0.5
- HIGH_UTILIZATION_REJECTION_RATE: 0.15
- LEGITIMATE_TRAFFIC_RATE: 2
- LEGITIMATE_CLIENT_COUNT: 400
- LEGITIMATE_LOAD_SIZE_LOWER: 1
- LEGITIMATE_LOAD_SIZE_UPPER: 3
- MALICIOUS_TRAFFIC_RATE: 15
- MALICIOUS_CLIENT_COUNT: 80
- MALICIOUS_LOAD_SIZE_LOWER: 12
- MALICIOUS_LOAD_SIZE_UPPER: 20

Run 8: Heavy Attack Payload

- NUM_SERVERS: 8
- SERVER_TIMEOUT: 2
- PROCESSING_POWER: 200
- REQUEST_TIMEOUT: 3
- MAX_REQUESTS_CONCURRENT: 25
- MAX_REQUEST_QUEUE_LENGTH: 400
- OFFLINE_CLEAR_THRESHOLD: 0.5
- HIGH_UTILIZATION_REJECTION_RATE: 0.15
- LEGITIMATE_TRAFFIC_RATE: 2
- LEGITIMATE_CLIENT_COUNT: 400
- LEGITIMATE_LOAD_SIZE_LOWER: 1
- LEGITIMATE_LOAD_SIZE_UPPER: 3
- MALICIOUS_TRAFFIC_RATE: 15
- MALICIOUS_CLIENT_COUNT: 80
- MALICIOUS_LOAD_SIZE_LOWER: 25
- MALICIOUS_LOAD_SIZE_UPPER: 40

Run 9: Large Infected Client Assault

- NUM_SERVERS: 8
- SERVER_TIMEOUT: 2
- PROCESSING_POWER: 250
- REQUEST_TIMEOUT: 3
- MAX_REQUESTS_CONCURRENT: 25
- MAX_REQUEST_QUEUE_LENGTH: 400
- OFFLINE_CLEAR_THRESHOLD: 0.5
- HIGH_UTILIZATION_REJECTION_RATE: 0.15
- LEGITIMATE_TRAFFIC_RATE: 2
- LEGITIMATE_CLIENT_COUNT: 400
- LEGITIMATE_LOAD_SIZE_LOWER: 1
- LEGITIMATE_LOAD_SIZE_UPPER: 3
- MALICIOUS_TRAFFIC_RATE: 25
- MALICIOUS_CLIENT_COUNT: 200
- MALICIOUS_LOAD_SIZE_LOWER: 12
- MALICIOUS_LOAD_SIZE_UPPER: 20

Run 10: Analyzing Network Endurance

- NUM_SERVERS: 8
- SERVER_TIMEOUT: 2
- PROCESSING_POWER: 250
- REQUEST_TIMEOUT: 5
- MAX_REQUESTS_CONCURRENT: 25
- MAX_REQUEST_QUEUE_LENGTH: 400
- OFFLINE_CLEAR_THRESHOLD: 0.5
- HIGH_UTILIZATION_REJECTION_RATE: 0.15
- LEGITIMATE_TRAFFIC_RATE: 2
- LEGITIMATE_CLIENT_COUNT: 400
- LEGITIMATE_LOAD_SIZE_LOWER: 1
- LEGITIMATE_LOAD_SIZE_UPPER: 3
- MALICIOUS_TRAFFIC_RATE: 18
- MALICIOUS_CLIENT_COUNT: 120
- MALICIOUS_LOAD_SIZE_LOWER: 12
- MALICIOUS_LOAD_SIZE_UPPER: 20

Aside from the scenarios and outcomes outlined in Milestone 3, Milestone 4 outlines a total of fourteen distinct simulation scenarios by which sensitivity analysis and real-world network behavioral comparison is performed. Of these fourteen scenarios, eleven were utilized in order to conduct sensitivity analysis, one of which being a baseline scenario in order to provide a point of comparison for later sensitivity calculations. While the full baseline configuration can be found in Section 8.1 of this literature, it is important to recognize that the following key parameters were specifically chosen in order to assess the impact network infrastructure, processing capability, and network traffic have on simulation outcomes as a whole.

- NUM_SERVERS
- PROCESSING_POWER
- MAX_REQUEST_QUEUE_LENGTH
- MALICIOUS_TRAFFIC_RATE
- LEGITIMATE_CLIENT_COUNT

The other ten scenarios utilized in the sensitivity calculations were extrapolations of the baseline scenario, with variance added to the parameters above in order to measure the change in output caused by a change in input. Lastly, the three remaining scenarios outlined in Milestone 4 were finely tuned to capture real-world network behavior in an attempt to tailor results to authentic network situations and illustrate a clear story surrounding simulation executions. These three scenarios were arguably the most important throughout the project's lifetime, as they tell a clear and actionable story surrounding the simulation outcomes and were later used to validate the overall simulation approach against real network data. To avoid redundancy, the full parameter configurations for these three scenarios can be found in Section 8.2 of this literature.

7.3 Data Collection Methods

The final simulation implementation comprehensively collects a multitude of different metrics across categories such as performance, system state, event counts and timings, resource consumption, quality and accuracy measure, and configuration parameters. The metrics collected are categorized as follows:

- **Performance**: Total Requests Received, Total Requests Processed, Success Rate, Acceptance Rate, Processing Rate, Drop Rate, Overall Success Rate, Legit Success Rate, Botnet Success Rate, Total Generated Requests, and Total Served Requests
- **System State**: Time Spent Offline, Offline Percentage, Simulation Duration, Active Workers, Health Score, Total Processing Capacity, and Capacity Threshold
- **Event Counts and Timings**: Drops Queue Full, Drops Timeout, Drops High Load, Total Drops, Legit No Response, and Botnet No Response
- **Resource Consumption**: CPU Utilization, Queue Utilization, and Queue Length
- **Quality and Accuracy Measures**: Bandwidth Exhaustion Probability, Resource Depletion Probability, and Successful Attack Probability
- **Configuration Parameters**: Number of Servers, Processing Power, Queue Size, Legitimate Clients, Malicious Clients, Legitimate Traffic Rate, Malicious Traffic Rate, Request Timeout, and Server Timeout

By collecting metrics at such a high level of granularity and detail across the network and at individual servers, this facilitated detailed analysis of system-wide behaviors and assisted in precisely identifying the impact of focused network traffic over time.

7.4 Experimental Design

Experimental design refers to the process of planning a study to test a hypothesis, where variables are manipulated in order to observe their effects on outcomes. By carefully controlling conditions, researchers can determine whether specific changes in factors can cause changes in a dependent variable. In the context of the malicious network traffic simulation, by meeting implementation objectives such as effectively modeling networks of interconnected machines, network requests, network routers, and network servers as well as capturing core network interactions such as request generation, network traffic routing, server request processing and responses, and service availability impact under load, we achieved complete and total control over the experimental design. The experimental design was carefully crafted in order to observe how safeguards such as load-balancing and increased network topology contribute to reduced latency, improved throughput, and decreased server deterioration under load. Furthermore, the experimental design intended to explore how changing topology parameters impact service availability, request response rates, and overall network health.

8 Results and Analysis

8.1 Sensitivity Analysis Findings

As outlined in the previous milestone and earlier in this literature, extensive sensitivity analysis was conducted in order to explore how changes to the configuration parameters impact the behavior and outcomes of the malicious network traffic simulation. Furthermore, this section identifies which parameters have the most influence on the system's behavior. While the simulation produces a comprehensive report of execution outcomes, the key outcome being observed is overall request success rate, as it provides a reliable means of gauging system health and service availability. Eleven finely tuned simulation executions were performed with the inclusion of one baseline execution as a point of comparison.

Baseline Configuration

- SIMULATION_DURATION = 300
- INTERVAL_DATA_POLLING = 5
- INTERVAL_OUTPUT_POLLING = 10
- NUM_SERVERS = 8
- PROCESSING_POWER = 250
- MAX_REQUEST_QUEUE_LENGTH = 400
- REQUEST_TIMEOUT = 2
- SERVER_TIMEOUT = 2
- MAX_REQUESTS_CONCURRENT = 20
- CPU_UTILIZATION_HEALTH_WEIGHT = 0.4
- QUEUE_UTILIZATION_HEALTH_WEIGHT = 0.6
- OFFLINE_CLEAR_THRESHOLD = 0.5
- HIGH_UTILIZATION_REJECTION_RATE = 0.15
- INCREASED_UTILIZATION = 0.70
- HIGH_UTILIZATION = 0.85
- CRITICAL_UTILIZATION = 0.95
- LEGITIMATE_TRAFFIC_RATE = 2
- LEGITIMATE_CLIENT_COUNT = 400
- LEGITIMATE_LOAD_SIZE_LOWER = 1
- LEGITIMATE_LOAD_SIZE_UPPER = 3
- MALICIOUS_TRAFFIC_RATE = 15
- MALICIOUS_CLIENT_COUNT = 80
- MALICIOUS_LOAD_SIZE_LOWER = 10
- MALICIOUS_LOAD_SIZE_UPPER = 15

Across these executions, variance targeting the following five configuration parameters was introduced. These parameters were specifically chosen in order to assess the impact network infrastructure, processing capability, and network traffic have on simulation outcomes as a whole.

- NUM_SERVERS
- PROCESSING_POWER
- MAX_REQUEST_QUEUE_LENGTH
- MALICIOUS_TRAFFIC_RATE
- LEGITIMATE_CLIENT_COUNT

The baseline configuration was designed to model an enterprise system under moderate to increased load. In this execution, a majority of requests were serviced with minor lapses in network availability, as illustrated by an overall success rate of 87.76%. This configuration exhibits this kind of behavior in order to better identify which changes in parameters alleviate network pressure or otherwise send the network into an outage. A high and low execution was performed for each targeted parameter whilst the remaining baseline parameters remained the same. By recording the simulation configurations under each scenario, these records allow us to calculate the % change in input when assessing the sensitivity for each varied parameter. This was done in order to comprehensively assess the simulation's sensitivity across a wide range of potential system inputs, and to provide practical insights surrounding the data. The sensitivity values can be seen in Table 1.

Of the sensitivity calculations outlined in Table 1, the parameters that exhibit the highest sensitivity with respect to simulation outcomes are NUM_SERVERS and PROCESSING_POWER. It is clear from the calculations that the system is highly vulnerable to drops in overall request success rate when either of these two parameters are decreased. However, it is important to note that this sensitivity relationship is not symmetric and does not

Table 1. Sensitivity Analysis Results Demonstrating High Network Infrastructure Sensitivity.

Parameter	**Case**	**Success %**	**% Change Input**	**% Change Output**	**Sensitivity**
NUM_SERVERS	Low (6)	45.58%	-25.00%	-48.05%	1.9218
	High (10)	99.35%	25.00%	13.25%	0.5298
PROCESSING_POWER	Low (200)	56.44%	-20.00%	-35.67%	1.7833
	High (300)	99.70%	20.00%	13.64%	0.6822
MAX_REQUEST_QUEUE_LENGTH	Low (300)	69.41%	-25.00%	-20.88%	0.8353
	High (500)	90.64%	25.00%	3.32%	0.1327
MALICIOUS_TRAFFIC_RATE	Low (10)	99.51%	-33.33%	13.43%	-0.4028
	High (25)	38.45%	66.67%	-56.17%	-0.8426
LEGITIMATE_CLIENT_COUNT	Low (200)	93.37%	-50.00%	6.43%	-0.1286
	High (700)	49.11%	75.00%	-44.02%	-0.5870

behave in the same fashion when increasing either of these parameters. As exhibited in the calculations above, the sensitivity for the high value for NUM_SERVERS is 0.5298 and the high value for PROCESSING_POWER is 0.6822. These values demonstrate that changes in the output are less than changes in the input, illustrating how increasing these parameters only marginally increase the overall request success rate. Ultimately, as previously expressed throughout Milestones 3 and 4, it is clear that network processing capabilities and network infrastructure have the most significant impact on simulation outcomes.

8.2 Scenario Comparisons

Over the course of the past sixteen weeks, a total of twenty-four unique simulation executions were conducted in order to capture nuanced aspects of system performance capability, parameter analysis, and other valuable system information. Of these twenty-four scenarios outlined across Milestones 3 and 4, the most insightful scenarios were the three execution scenarios captured in Section 3.1 of Milestone 4. These scenarios, as outlined previously, were finely tuned to model real-world attack scenarios in an attempt to tailor results to authentic network situations and illustrate a clear story surrounding simulation executions. It is important to clarify that among these scenarios user configurable parameters such as CPU utilization health weight (0.4), queue utilization health weight (0.6), increased utilization threshold (0.70), high utilization threshold (0.85), and critical utilization threshold (0.95) remained the same in order to preserve the validity of simulation outcomes and ensure that variance in execution outcomes were a direct result of core pipeline changes as opposed to calculation variance. Furthermore, values such as simulation duration (250), interval data polling (5), and interval output polling (10) were kept unchanged between simulation scenarios in order to maintain a consistent level of data granularity between execution outcomes. The rest of the scenario parameters are as follows:

Scenario 1: Daily Traffic with Mixed Malicious Traffic

- NUM_SERVERS: 15
- REQUEST_TIMEOUT: 3
- SERVER_TIMEOUT: 2
- PROCESSING_POWER: 500
- MAX_REQUESTS_CONCURRENT: 40
- MAX_REQUEST_QUEUE_LENGTH: 800
- LEGITIMATE_TRAFFIC_RATE: 5
- LEGITIMATE_CLIENT_COUNT: 700
- LEGITIMATE_LOAD_SIZE_LOWER: 1
- LEGITIMATE_LOAD_SIZE_UPPER: 3

- MALICIOUS_TRAFFIC_RATE: 20
- MALICIOUS_CLIENT_COUNT: 200
- MALICIOUS_LOAD_SIZE_LOWER: 12
- MALICIOUS_LOAD_SIZE_UPPER: 20

Scenario 2: Holiday Peak with Moderate Malicious Traffic

- NUM_SERVERS: 15
- REQUEST_TIMEOUT: 3
- SERVER_TIMEOUT: 2
- PROCESSING_POWER: 500
- MAX_REQUESTS_CONCURRENT: 40
- MAX_REQUEST_QUEUE_LENGTH: 800
- LEGITIMATE_TRAFFIC_RATE: 5
- LEGITIMATE_CLIENT_COUNT: 900
- LEGITIMATE_LOAD_SIZE_LOWER: 1
- LEGITIMATE_LOAD_SIZE_UPPER: 3
- MALICIOUS_TRAFFIC_RATE: 28
- MALICIOUS_CLIENT_COUNT: 250
- MALICIOUS_LOAD_SIZE_LOWER: 12
- MALICIOUS_LOAD_SIZE_UPPER: 20

Scenario 3: Coordinated HTTP Flood

- NUM_SERVERS: 15
- REQUEST_TIMEOUT: 3
- SERVER_TIMEOUT: 2
- PROCESSING_POWER: 500
- MAX_REQUESTS_CONCURRENT: 40
- MAX_REQUEST_QUEUE_LENGTH: 800
- LEGITIMATE_TRAFFIC_RATE: 5
- LEGITIMATE_CLIENT_COUNT: 700
- LEGITIMATE_LOAD_SIZE_LOWER: 1
- LEGITIMATE_LOAD_SIZE_UPPER: 3
- MALICIOUS_TRAFFIC_RATE: 30
- MALICIOUS_CLIENT_COUNT: 350
- MALICIOUS_LOAD_SIZE_LOWER: 12
- MALICIOUS_LOAD_SIZE_UPPER: 20

8.3 Results Under Each Scenario

Table 2. Scenario 1 Results: Daily Traffic with Mixed Malicious Traffic

Metric	Value
Traffic Summary	
Total Generated Requests	1,874,800
Total Served Requests	1,866,063
Total Dropped Requests	8,737
Drop Breakdown	
Dropped - Queue Full	17
Dropped - Timeout	140
Dropped - High Load	740
Dropped - No Server	7,840
Performance Metrics	
Overall Success Rate	99.53%
Overall Drop Rate	0.47%
Server Health Metrics	
Average CPU Utilization	63.8%
Average Queue Utilization	1.4%
Average Health Score	0.71
Average Offline Percentage	1.3%

Table 3. Scenario 2 Results: Holiday Peak with Moderate Malicious Traffic

Metric	Value
Traffic Summary	
Total Generated Requests	2,875,000
Total Served Requests	2,180,057
Total Dropped Requests	694,943
Drop Breakdown	
Dropped - Queue Full	402
Dropped - Timeout	10,893
Dropped - High Load	150,831
Dropped - No Server	532,817
Performance Metrics	
Overall Success Rate	75.83%
Overall Drop Rate	24.17%
Server Health Metrics	
Average CPU Utilization	82.0%
Average Queue Utilization	65.3%
Average Health Score	0.260
Average Offline Percentage	1.0%

Table 4. Scenario 3 Results: Coordinated HTTP Flood

Metric	Value
Traffic Summary	
Total Generated Requests	3,500,000
Total Served Requests	1,148,861
Total Dropped Requests	2,351,139
Drop Breakdown	
Dropped - Queue Full	1,332
Dropped - Timeout	1,317
Dropped - High Load	29,771
Dropped - No Server	2,318,719
Performance Metrics	
Overall Success Rate	32.82%
Overall Drop Rate	67.18%
Server Health Metrics	
Average CPU Utilization	56.1%
Average Queue Utilization	28.6%
Average Health Score	0.584
Average Offline Percentage	1.1%

8.4 Scenario Comparison and Insights

The execution scenarios outlined above were inspired by the different network conditions that enterprise service clusters face year-round. The first scenario was designed to simulate daily traffic of an ecommerce service such as Amazon.com, with a low volume of malicious traffic mixed in with high legitimate traffic in order to simulate daily operations for a service of this scale. This scenario established a comparative baseline for the other scenarios, as it captures the expected operating conditions of this service for any given day. The next simulation scenario was designed to capture this same service's behavior when faced with holiday peaks, characterized by an increased presence of both legitimate and malicious traffic. Finally, the last simulation scenario was designed to capture a large, coordinated attack against the service, characterized by an increase in malicious traffic during what would otherwise be normal operating conditions. These scenarios are the most insightful out of the twenty-four simulation executions conducted over the course of this project – they tell a clear, actionable story of how network traffic to enterprise services evolves overtime, as opposed to nuanced executions testing aspects such as network resilience.

Under the normal operating conditions captured in the first scenario, there were a total of 1,874,800 requests generated across 250 simulated seconds. Furthermore, there were 700 legitimate and 200 malicious clients in this instance, with a legitimate traffic rate of 5 requests per second and a malicious traffic rate of 20 requests per second. Under these conditions, the scenario performed well within what was expected for simulated daily traffic. The simulation results provided valuable insight concerning the scenario, as illustrated by a 99.53% request success rate and a 0.47% request drop rate combined with minimal server downtime and zero timeout drops. These scenario results established a realistic base of comparison for an average day of service, which was later used to establish insights surrounding the data as a whole.

Under the holiday peak scenario, there were a total of 2,875,000 requests generated across the same simulated time frame, an increase of 53.3%. Furthermore, the number of legitimate clients increased 29% to 900 while the number of malicious clients rose to 250. There was an associated rise in malicious traffic to 28 requests per second while legitimate traffic continued at the same rate as the baseline scenario. Under these conditions, the holiday peak scenario performed within what was expected for a system under duress. The overall request success rate dropped from 99.53% to 75.83% within the time period, while the drop rate rose from 0.47% to 24.17%. Despite these metrics indicating a drop in service availability, server downtime hovered around the 1% mark in both instances. This illustrates that request drops during the holiday scenario were not induced by the network being overwhelmed and ultimately shutting down due to the influx of traffic but were instead the result of request process timeouts due to the load size exceeding the processing capabilities of the servers in the cluster. This effectively captured the fact that there were enough servers to extinguish the threat of an outage, but the servers were too slow to be effective in servicing network traffic.

The coordinated HTTP flood scenario posed the greatest challenge for the target service. Under this scenario, there were a total of 3,500,000 requests generated, an increase of 86.7% across the simulated time frame. Furthermore, the number of legitimate clients remained the same as the baseline at 700, while the number of malicious clients rose to 350, an increase of 75%. This scenario had the steepest growth in malicious traffic rate, rising 50% from the baseline to 30 requests per second while the legitimate traffic rate remained the same as the baseline at 5 requests per second. These scenario parameters were designed to capture the expected traffic for the target service, in combination with a coordinated HTTP flood DDoS attack. Since the scenario was designed to capture expected traffic in combination with underlying malicious traffic, the parameters concerning legitimate traffic were unchanged from the baseline. The HTTP flood scenario performed within what was expected for this style of attack, as service availability significantly diminished. The overall request success rate dropped from 99.53% to 32.82%, while the overall drop rate increased from 0.47% to 67.18%. It is imperative to note that of the 3,500,000 requests generated during this scenario, 2,318,719 were dropped as a direct result of no server being available.

This indicates that unlike the holiday peak scenario where servers simply didn't have the ability to process all of the incoming requests efficiently, in this instance, the 75% increase in malicious clients and 50% increase in malicious request rate warranted 66.2% of requests being dropped due to a network service outage.

When observing these scenario outcomes, it is clear that the simulation is disproportionately affected by malicious traffic as opposed to legitimate traffic. This is illustrated when looking at the scenario outcomes across the holiday peak and HTTP flood scenarios. Despite the holiday peak scenario having 100 more total clients than the HTTP flood scenario, the additional 100 malicious clients as well as the 7.1% difference in attack rate in the flood scenario triggered a 43.01% reduction in overall success rate, even when considering the 200 client reduction in legitimate clients between the holiday peak scenario and coordinated attack scenario. Furthermore, data suggests that the modes of network failure have differing levels of severity as evidenced by the overall success rate between the holiday scenario and HTTP flood scenario. While the system is under duress in the holiday scenario, the data demonstrates that the network can still service a majority of the incoming requests and the overall network impact is not as severe. However, as evidenced by the flood scenario, there is a tipping point by which malicious request generation becomes severe to overall network service availability causing a critical decline in overall request processing.

8.5 Statistical Analysis

Table 5. Statistical Summary of Simulation Results Demonstrating High Data Variability.

	Mean	**Std. Deviation**	**Min**	**Max**
Overall Success Rate (%)	75.39%	22.95%	38.45%	99.70%
Overall Drop Rate (%)	24.60%	22.95%	0.30%	61.55%
Drops - No Server	138,278.73	166,806.37	0	467,140
Drops - Timeout	11,827.27	14,030.68	343	40,570

Given key statistical outcomes across the eleven distinct parameter analysis executions outlined in Section 8.1, metrics concerning success rate, drop rate, server availability timeouts, and request processing timeouts demonstrate significant variability between runs. Success rate ranged anywhere from 38.45% to 99.70%, with an average success rate of 75.39% and a standard deviation of 22.95%. This indicates that there was severe variability in performance concerning the success rate across the eleven executions. Furthermore, this sentiment is echoed when observing the overall drop rate across these executions. The drop rate maintained the exact same standard deviation of 22.95%, with an average drop rate of 24.60% and values ranging between 0.30% and 61.55%. For discrete request service counts, server availability drops ranged anywhere from 0 to 467,140 dropped requests. As such, the standard deviation was 166,806.37 for server availability drops, and the average across runs was 138,278.73. Finally, the average drops induced by a process timeout was 11,827 across eleven executions, with a minimum of 343 and a maximum of 40,570. As such, the standard deviation for this metric was 14,030.68 across the eleven executions.

8.6 Data Visualizations

While the simulation executions outlined in Milestone 3 were not tailored to tell a story regarding traffic evolution, these usage scenarios were important as they were specifically designed to capture nuances regarding system performance. These scenarios were utilized in order to capture and visualize trends with regards to request servicing, throughput and drop rate, and traffic network impact. Key findings with regards to performance patterns can be seen as follows:

Request Servicing

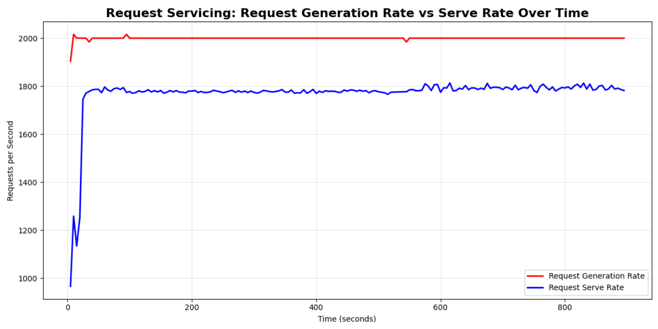


Fig. 9. Request generation rate versus service rate over time for the baseline configuration, illustrating a minor gap in request generation and request servicing.

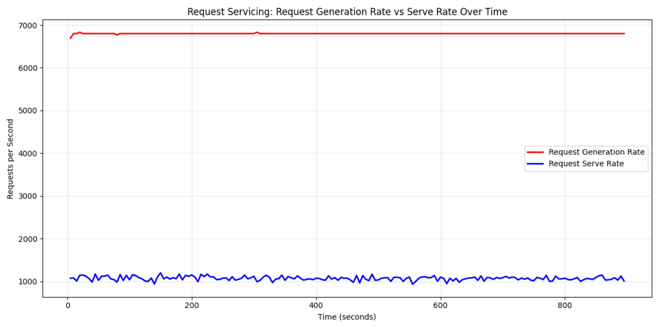


Fig. 10. Request generation rate versus service rate over time for intense HTTP flood configuration, demonstrating a large gap between request generation and servicing. This relationship indicates an inability to effectively serve traffic.

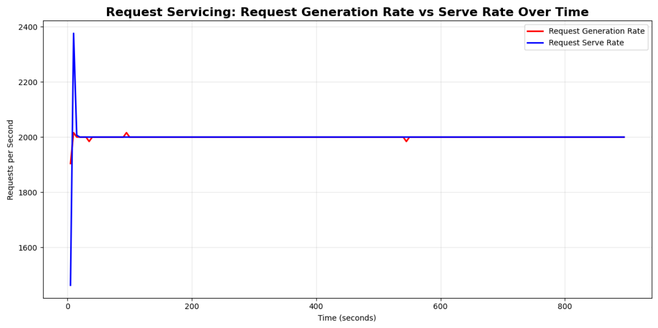


Fig. 11. Request generation rate versus service rate over time for load balancing pressure configuration, demonstrating a clear ability to service the entirety of incoming network traffic.

Throughput and Drop Rate

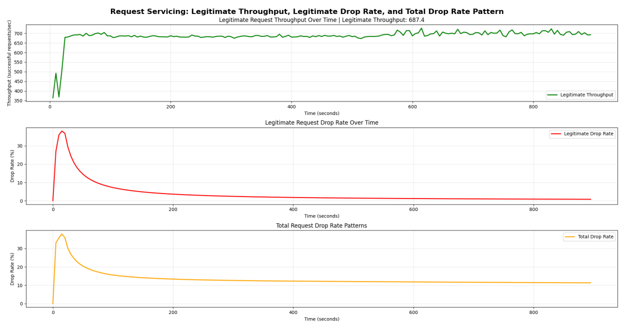


Fig. 12. Legitimate request throughput, legitimate request drop rate, and total request drop rate over time for baseline configuration. Stability in throughput in combination with declining drop rate illustrate an overall ability to service traffic.

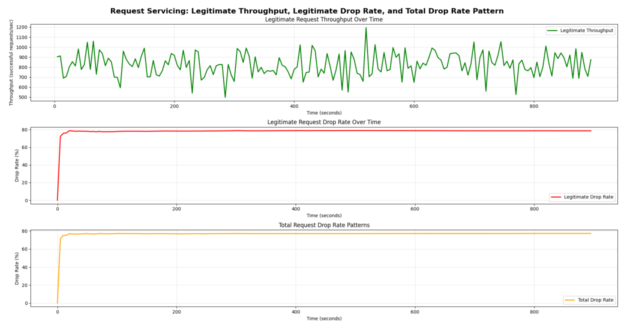


Fig. 13. Legitimate request throughput, legitimate request drop rate, and total request drop rate over time for intense HTTP flood configuration. Instability in throughput in combination with climbing drop rate illustrate an overall inability to service traffic.

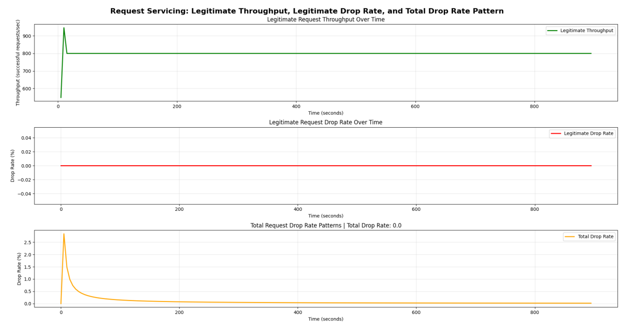


Fig. 14. Legitimate request throughput, legitimate request drop rate, and total request drop rate over time for intense load balancing pressure configuration. Constant throughput in combination with 0 drop rate illustrate an overwhelming ability to service incoming traffic.

Traffic Comparison and Impact

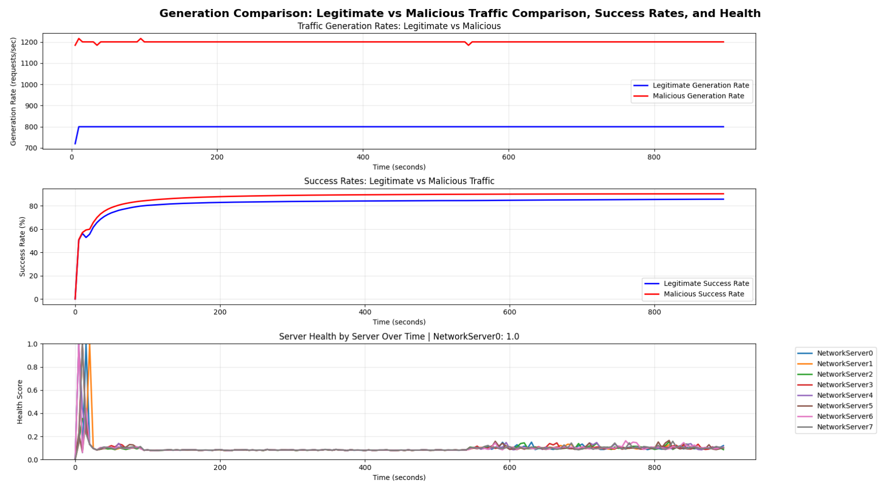


Fig. 15. Legitimate and malicious traffic rate, legitimate and malicious request success rate, and server health over time for baseline configuration. Demonstrating that server health is impacted due to incoming traffic, however, the network still sees significant success.

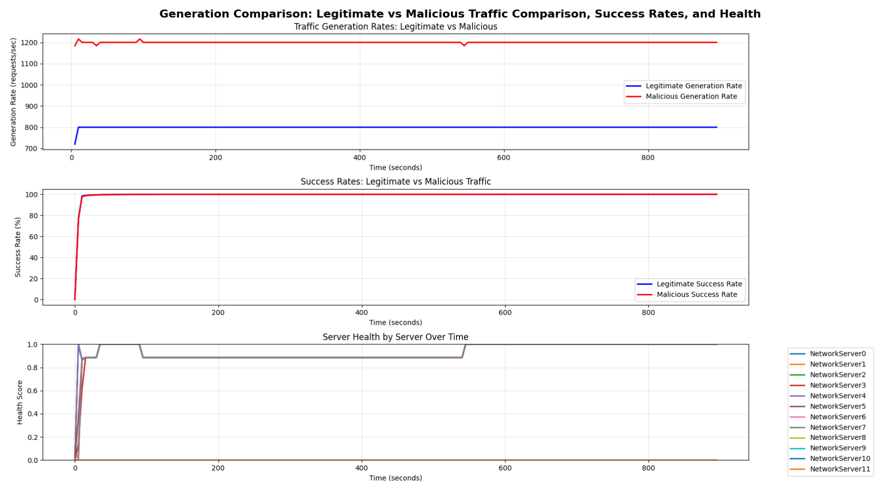


Fig. 16. Legitimate and malicious traffic rate, legitimate and malicious request success rate, and server health over time for load balancing pressure configuration. Illustrating that server health and success rate remain high despite incoming traffic.

9 Validation and Verification

It is imperative that proper validation and verification were conducted in order to ensure that the final simulation implementation produces reasonable results that align with real world enterprise network behavior. In order to validate the final implementation, both output comparison and face validation were conducted in order to ensure that the final system behaves appropriately.

It is important to note that, as outlined in Section 2.4 of Milestone 3, the main issue when initially collecting simulation results was memory exhaustion and unexpected simulation outcomes. When first implementing research-driven configuration parameters, it became apparent that utilizing enterprise-scale infrastructure parameters would not be feasible, as these configuration values generated results that were far too large to store and accurately manage. In many cases, final values would appear as zero or negative numbers, indicating overflow and memory allocation issues. As a result, extensive research was conducted in order to establish infrastructure ratios that accurately captured the real-world relationships of core simulation components. In the final implementation, many of these values have been scaled down roughly 100:1 while maintaining their relative sizes to one another. This was done in accordance with an industry report published by OneChassis which states that enterprise data centers typically house between 500 and 5,000 servers, while in the final implementation the number of servers is between five and fifty.

When comparing outputs for refined simulation scenarios against network traffic data provided by Wikimedia's Grafana installation, in combination with OneChassis' infrastructure report, it became clear that the simulation is accurate in modeling real-world network behavior and that simulation parameter values are within reasonable ranges. Wikimedia is the nonprofit organization that hosts Wikipedia, and their service provided a comprehensive and comparable baseline in order to help validate the network simulation's outcomes. According to Wikimedia's Grafana installation [12], as of November 12th the service receives anywhere between 100,000 and 200,000 requests per second across a twenty-four hour period. Furthermore, according to Wikipedia's own technical FAQ, the service is maintained by approximately 350 servers. With this information, we determined that each enterprise server receives anywhere from 286 to 572 requests per second. Scaling these ranges to the simulated fifteen server network, the total request generation rate for the daily traffic scenario outlined in Section 8.2 should fall between 4,290 and 8,580 requests per second. Although it is not explicitly listed as a simulation outcome, the total requests per second – 1,874,800 generated requests over a 250 unit time frame – for the scenario is 7,500. This illustrates that the request generation is accurate for a simulation of this size, and values are properly scaled for simulating daily traffic for an enterprise service such as Wikipedia. Furthermore, Wikimedia's Grafana dashboard reported an average of 4.87 errors per second across the same twenty-four hour period, an error rate of approximately 0.003%. While the baseline simulation's overall drop rate of 0.47% is higher than what is reported by Wikimedia, this difference can be attributed to advanced network infrastructure and architectural decisions in combination with the simulation limitations outlined both in Milestone 1 and later in this section. Despite this difference, the exceptionally low values of drop rates are still a testament to the validity of the implementation approach and serve to reflect lapses in service when under load. In all, these values illustrate that the model is behaving reasonably.

In addition to output comparison, face validation also supports that the final implementation is correct. As outlined in Milestone 1 as well as Section 4 of this report, the simulation design and core architectural components have been carefully crafted according to academic literature such as:

- *DDoS Attack and Detection Methods in Internet-Enabled Networks* by Kazeem A. Adnan A. and Anish M. [1]
- *Botnets Multiply and LevelUp* by Chris C., published by NETSCOUT SYSTEMS, INC [3]
- *BOTNET: Lifecycle, Architecture and Detection Model* by Kazeem A. Adnan A. and Anish M. [6]
- *Queuing Analysis* by William S. [9]
- *Routing Algorithms: A Review* by Ujjwal S., Vikas K. and Shubham K., under the guidance of Dr. Jayasheela [8]

The literature above has served as an extensive means of validation and has been instrumental in ensuring that the final simulation behaves as accurately as possible within the context of network behavior. To illustrate, the core architectural design for the Botnet component within the simulation is directly modeled after the centralized C&C topology outlined in Divya N., Pooja W., Sanjay S., and Deepak K's academic paper, furthermore, the server processing design is heavily inspired by William Stalling's findings in his paper Queuing Analysis. These literature driven decisions encompass the entirety of the simulation's architectural design, ensuring that the final implementation is as robust and comprehensive as possible given simulation's limitations.

While the current simulation implementation provides a comprehensive and valid means of assessing malicious network traffic, there are multiple key limitations within the model that must be addressed. As outlined in Milestone 1, request response time is not impacted by the network topology and the communication time between network components is zero. While this is sufficient for conducting executions in the simulated network environment, network topology in real-world systems can have a pronounced effect on network response times and processing capabilities. To further this point, request routing algorithms and traffic safeguards in real-world network systems often utilize metrics concerning network topology in order to drive routing and request handling decisions, in this simulation implementation, the only metric driving routing decisions is the health of the servers within the cluster. Beyond limitations induced by network topology decisions, the servers within this simulation are all identical. While this is sufficient for the simulated network environment, enterprise services may not maintain homogeneous network servers due to their individual business needs.

10 Discussion

The findings outlined in this report alongside explicit patterns in execution outcomes answer key questions concerning how Distributed Denial of Service attacks (DDoS) impact network performance metrics such as server response time, request throughput, and service availability. In addition, by meeting the implementation objectives set forth in Milestone 1, the results answer how safeguards such as load-balancing help reduce latency, improve throughput, and prevent server deterioration when under load. Finally, execution outcomes indicate key findings surrounding relationships between attack surface area and network topology that contribute significantly to the overall network domain.

As outlined in Section 8.1 of this literature, extensive sensitivity analysis was conducted in order to explore how changes to the configuration parameters impact the behavior and outcomes of the malicious network traffic simulation. It was found that the parameters that exhibit the highest sensitivity with respect to simulation outcomes are NUM_SERVERS and PROCESSING_POWER (Sensitivity values of 1.9218 and 1.7833 respectively). It became clear from the calculations that the system is highly vulnerable to drops in overall request success rate when either of these two parameters are decreased. However, interestingly enough, this sensitivity relationship is not symmetric and does not behave in the same fashion when increasing either of these parameters, as increasing these parameters only marginally increases the overall request success rate. While the fact that the sensitivity relationship is not symmetric was surprising, this hyper-sensitivity to network topology and performance capability solidified a previous hypothesis that the largest contributor to system performance was network infrastructure. This ultimately reinforced my thinking surrounding the relationship between network infrastructure and performance, whilst identifying a nuance concerning diminishing returns when expanding network infrastructure.

Furthermore, it is clear that the simulation is disproportionately affected by malicious traffic as opposed to legitimate traffic. This is illustrated when looking at the scenario outcomes across the holiday peak and HTTP flood scenarios. Despite the holiday peak scenario having 100 more total clients than the HTTP flood scenario, the additional 100 malicious clients as well as the 7.1% difference in attack rate in the flood scenario triggered a 43.01% reduction in overall success rate, even when considering the 200 client reduction in legitimate clients between the holiday peak scenario and coordinated attack scenario. Furthermore, the data suggests that the modes of network failure have differing levels of severity as evidenced by the overall success rate between the holiday scenario and HTTP flood scenario. While the system is under duress in the holiday scenario, the data demonstrates that the network can still service a majority of the incoming requests and the overall network impact is not as severe. However, as evidenced by the flood scenario, there is a tipping point by which malicious request generation becomes severe to overall network service availability causing a critical decline in overall request processing. These are two very important practical implications to be aware of in the context of modern network design, as the disproportionate impact of malicious traffic in combination with the varying degrees of service degradation can serve as powerful status indicators that can allow network professionals to identify attacks early.

Finally, simulation outcomes throughout previous milestones provide clear evidence to the effect that load-balancing plays a critical role in reducing latency, improving throughput, and preventing server deterioration when under load. As outlined in Section 7.2 of this reading, Milestone 3 contained ten distinct configuration scenarios in order to capture nuanced aspects of system performance capability – despite not having a formalized section for execution outcomes in this report, please see Appendix A, Table 6 for comparative results. The simulation data demonstrates that there is reason to believe that the low success rate is directly a result of unsuitable network infrastructure. Observing execution 5, the number of servers and their respective processing abilities have been increased in this execution's parameter configuration. Since execution 5 has a much stronger network infrastructure, and thus requests can be distributed among network actors fairly, the success rate remains close to 100% despite having a comparable number of requests generated to the other execution scenarios.When taking a look at executions 1, 6, and 7, we observe that the total requests generated is exactly 1,799,520 across all three simulation executions. However, the key difference between the three of these executions lies in the processing ability of the network. The baseline configuration - execution 1 - has eight network servers with a processing power of 250, a concurrent request maximum of twenty-five, and a request queue length of 400. On the other hand, execution 7 has six servers, with a processing power of 200, concurrent maximum of 22, and max queue length of 350. Finally, execution 6 had five servers, processing power of 300, concurrent maximum of 20, and max queue length of 300. Despite what would seem like small changes in the processing ability, there is a 54.23% success drop between executions 1 and 7, and additional 11.09% drop between executions 7 and 6. These outcomes are a stark indicator of the importance of the network processing ability and how distributing the traffic among servers allows for exponential gains in network throughput.In all, there are numerous practical implications that can be extracted from the results that are relevant to the overall network domain.

11 Conclusion and Future Work

In summary, the findings outlined in this report and previous milestones alongside the final simulation implementation contribute significantly to the overall network domain, as the findings conceptualize the depth by which Distributed Denial of Service attacks (DDoS) impact network performance metrics such as server response time, request throughput, and service availability. Furthermore, the report underscores the importance safeguards such as load-balancing contribute to reducing latency, improving throughput, and preventing server deterioration when under load. In all, I have learned many valuable lessons concerning performing academic research, development best practices, and conducting comprehensive data analysis, and I am extremely grateful for the opportunity to have participated in a project such as this. While there still remains areas by which the final solution can be improved, such as the implementation limitations outlined in Section 9, I am extremely proud of where the final simulation ended up. In the future, I wish to continue directing research toward the network domain as I feel it remains an area of relevance to those in the technology sector and other technologically adjacent fields.

References

[1] Kazeem B. Adedeji, Adnan M. Abu-Mahfouz, and Anish M. Kurien. 2023. DDoS Attack and Detection Methods in Internet-Enabled Networks: Concept, Research Perspectives, and Challenges. *Journal of Sensor and Actuator Networks* 12, 4, Article 51 (2023). doi:10.3390/jsan12040051

[2] Cloudflare. 2025. DDoS Threat Report for 2025 Q2. https://radar.cloudflare.com/reports/ddos-2025-q2. Accessed: 2025.

[3] Chris Conrad. 2022. *Botnets Multiply and Level Up*. DDoS Threat Intelligence Report. NETSCOUT SYSTEMS, INC. Report Number: SEC04.

[4] Offshore Dedi. 2024. *Server Components 101: A Comprehensive Look at Server Hardware*. OffshoreDedi. https://offshorededi.com/server-components-101-a-comprehensive-look-at-server-hardware/ Blog post.

[5] SimPy Developers. 2025. *SimPy: Discrete event simulation for Python*. https://simpy.readthedocs.io/ Python library for discrete-event simulation.

[6] Divya Nagpal, Pooja Wadhwa, Sanjay Singh, and Deepak Kumar. 2023. BOTNET: Lifecycle, Architecture and Detection Model. In *Proceedings of the International Conference on Cyber Security and Cryptography*. 112–125.

[7] pyest Development Team. 2025. *pytest: A mature full-featured Python testing framework*. https://docs.pytest.org/ Python testing framework.

[8] Ujjwal Sharma, Vikas Kumar, Shubham Kumar, and Dr. Jayasheela. 2024. Routing Algorithms: A Review. *arXiv preprint* (2024). https://arxiv.org/abs/2403.11228v1

[9] William Stallings. 2022. *Queuing Analysis in Computer Networks*. Pearson Education.

[10] Microsoft Azure Network Security Team. 2024. *Understanding the Evolving Threat of DDoS Attacks in 2024*. Microsoft. https://techcommunity.microsoft.com/blog/azurenetworksecurityblog/understanding-the-evolving-threat-of-ddos-attacks-in-2024/4362031 Microsoft Tech Community Blog.

[11] Matplotlib Development Team. 2025. *Matplotlib: A 2D graphics environment*. https://matplotlib.org/ Python plotting library.

[12] Wikimedia Foundation. 2025. Home: W Wiki Status - Wikimedia Grafana Dashboard. https://grafana.wikimedia.org/d/O_OXJyTVk/home-w-wiki-status. Accessed: 2025-11-12.

A Extended Data Tables

Table 6. Statistics from Simulation Runs Across Milestone 3 Configuration Scenarios

Run	**Scenario**	**Success %**	**Drop %**	**Attack Success**	**Total Generated**	**Total Served**	**Lead Drop Cause**
1	Baseline	88.95%	11.05%	0.43%	1,799,520	1,600,682	High Load (193,723)
2	Intense Flood	15.76%	84.24%	88.27%	6,119,600	964,570	No Server (5,123,638)
3	High Legitimate	22.92%	77.08%	74.87%	4,678,920	1,072,344	No Server (3,576,292)
4	Mixed Workload	21.89%	78.11%	80.83%	4,589,300	1,004,774	No Server (3,554,066)
5	Load Balancing	99.99%	0.01%	0.00%	1,799,520	1,799,321	High Load (134)
6	High Outage	23.63%	76.37%	77.35%	1,799,520	425,314	No Server (1,339,182)
7	Moderate Outage	34.72%	65.28%	64.58%	1,799,520	624,777	No Server (1,140,011)
8	Heavy Payload	32.18%	67.82%	65.44%	1,799,520	579,163	No Server (920,977)
9	Large Botnet	18.59%	81.41%	85.45%	5,219,600	970,443	No Server (4,216,698)
10	Endurance	38.02%	61.98%	61.14%	2,663,600	1,012,601	No Server (1,617,785)