Highlights

STGen: A Lightweight, Protocol-Agnostic IoT Testbed for Scenario-Based Protocol Evaluation with Network Condition Emulation

Hasan MA Islam, Mahamudur Rahman Maharaj, S. M. Nafis Shahriar, Subrata Nath, Pulok Akibuzzaman, Mahdin Islam Ohi, Michael Georgiades, Riadul Islam

- STGen is a lightweight, protocol-agnostic testbed for IoT traffic generation.
- Process-based architecture reduces memory usage by 87.5% compared to GothX.
- Supports real-time traffic analysis using the ELK stack (Elasticsearch, Kibana).
- Enables scalable emulation of 6,000+ sensor nodes on a single machine.

STGen: A Lightweight, Protocol-Agnostic IoT Testbed for Scenario-Based Protocol Evaluation with Network Condition Emulation⋆

Hasan MA Islam[a,*], Mahamudur Rahman Maharaj[a], S. M. Nafis Shahriar[a], Subrata Nath[a], Pulok Akibuzzaman[a], Mahdin Islam Ohi[a], Michael Georgiades[b] and Riadul Islam[c]

[a]Dept. of Computer Science and Engineering, East West University, Dhaka, Bangladesh
[b]Dept. of Computer Science, Neapolis University of Pafos, Pafos, Cyprus
[c]University of Maryland, Baltimore County, Baltimore, MD 21250, USA

ARTICLE INFO

Keywords:
Internet Protocol
IoT
Sensor Traffic
Verification
Validation
ELK

ABSTRACT

This paper introduces Sensor Traffic Generator, STGen, a lightweight testbed that works with any protocol. It helps developers design, test, and examine network performance, protocols, and IoT applications in a virtual setting. This tool allows for quick prototyping and debugging of applications before they are implemented on hardware. It saves time and money compared to building physical hardware by simulating node behavior, communication, and environmental factors in a controlled environment. The experiment demonstrates that STGen reduces setup time from 26 minutes to 1.6 seconds, corresponding to an approximately three-orders-of-magnitude speedup, and achieves an ≈8× reduction in memory consumption (2.56 GB vs. 20.4 GB) compared to the existing simulator, GothX, while scaling to 6,000 nodes on a single machine. In addition, STGen's modular architecture and flexible interfaces (CLI/GUI) provide a powerful and accessible platform for IoT protocol research.

1. Introduction

Wireless Sensor Networks (WSNs) refer to networks of distributed sensors that collect data from the surroundings and broadcast within range [1, 2]. These are similar to wireless ad-hoc networks [3] in the sense that they rely on wireless connectivity and the spontaneous formation of networks so that sensor data can be transported wirelessly. A typical sensor node consists of a communication module, a processing unit, different sensor interfaces, and a power source, usually a battery or an embedded source of energy [4]. The energy limitation along with computation constraint of sensor nodes demands efficient protocol design and evaluation to increase efficiency [5]. With the deployment of gigabit broadband and the rollout of 5G/6G network protocols, evaluation frameworks should emulate both constrained and high-performance network conditions [6].

Protocol researchers and network engineers often use traffic generation tools to benchmark network performance and troubleshoot issues [7]. Although many frameworks and tools [8–13] have been proposed by research communities and software industries, most of them lack support for lightweight and easily configurable traffic generation capabilities, as well as extensibility for Internet of Things (IoT) specific application layer protocols, particularly in controlled environments like WSNs.

Sim	Lang	Focus	Curve	Scale
NS-3 [15]	C++	Packet-level net sim, PHY/MAC models.	High	10^4+
OMNeT++ [14]	C++	Modular protocol sim; component modeling.	V. High	High
Cooja [16]	Java	OS-level emulation (Contiki), WSNs.	Mod.	< 100

Table 1
Common network simulators used for WSN evaluation

Network simulators give users control over the entire network stack, but they create challenges for testing application-layer protocols. OMNeT++ [14] requires knowledge of C++, which has a steep learning curve. NS-3 [15] needs a complex setup and configuration through Python or C++. Cooja (Contiki) [16] offers real OS simulation, but it has limited scalability and Java-based GUI complexity. Table 1 compares these simulators. While these tools are strong for network-layer research, protocol development can take weeks or months.

For researchers developing application-layer protocols, this challenge can hinder rapid experimentation, especially during prototyping phases that require fast iteration.

To enable extensive and scalable protocol evaluation, this article introduces STGen, a lightweight, plugin and

*Corresponding author
thumbnails/cas-email.jpeg (H.M. Islam);
rahmanmehraj627@gmail.com (M.R. Maharaj); nafisshahriar003@gmail.com (S.M.N. Shahriar); shuvra.dev9@gmail.com (S. Nath); pulok519@gmail.com (P. Akibuzzaman); mahdinislamohi@gmail.com (M.I. Ohi); riaduli@umbc.edu (R. Islam)
ORCID(s): 0009-0005-7028-7902 (H.M. Islam); 0009-0006-4513-6552 (M.R. Maharaj); 0009-0006-7414-6434 (S.M.N. Shahriar); 0009-0004-4119-2333 (P. Akibuzzaman); 0009-0005-4056-9757 (M.I. Ohi); 0000-0002-5930-8814 (M. Georgiades); 0000-0002-4649-3467 (R. Islam)

Figure 1: Distributed STGen testbed setup for experimenting with various IoT protocols and big data analytics on sensor data. Emulated testbeds feed sensor data to the STGen Core. Client applications access data using a custom UDP protocol.

process-based testbed designed to emulate large-scale IoT-based Wireless Sensor Network (WSN) environments. STGen supports the instantiation of thousands of virtual sensor nodes capable of generating high-volume sensor traffic, making it suitable not only for evaluating protocol performance in traditional IoT settings with limited bandwidth, but also for assessing scalability, CPU efficiency, and end-to-end latency under conditions representative of modern high-speed networks.

The STGen Core efficiently manages a large number of virtual sensor nodes while maintaining low memory overhead, enabling realistic large-scale experimentation on a single machine. In addition, STGen allows client applications to receive specific sensor data over the public Internet by subscribing to selected virtual sensors. This design facilitates seamless integration between emulated IoT environments and real-world client applications, thereby supporting scalable, reproducible, and practical protocol research.

To validate its capabilities, we have implemented a custom application-layer protocol built on top of UDP[17], following a publish/subscribe communication paradigm [18]. Protocol, scenario, or network configurations can be crafted to reflect realistic deployment environments, transmitting synthetic data to a designated sink node, referred to as the STGen Core in our architecture. The STGen Core handles resource-intensive tasks such as maintaining the client state table and delivering data to clients immediately upon subscription. Our evaluation focuses explicitly on the end-to-end communication path between external clients and the STGen Core using this custom protocol. This includes measuring performance metrics such as latency, packet loss, and utilization of memory and CPU under various levels. Throughout the communication process, network traffic is also logged, including sensor data, for subsequent analysis. These logs are utilized in conjunction with a robust data visualization stack (ELK) [19] to assess system performance and obtain deeper analytical insights.

STGen could be a great tool for researchers and developers to experiment with device integration and improve the performance of their IoT application services, such as real-time monitoring, traffic shaping, and edge-level data analytics. The key contributions of this work are the following.

1. Development of a lightweight, easily extensible, and customizable sensor traffic generator testbed for the experimentation of the IoT protocol in a hybrid network environment as shown in Figure 1.
2. Platform independent, easy to control, and easy to configure through the Command Line Interface (CLI).
3. Implementation of a lightweight Publish/Subscribe (Pub/Sub) based application on top of UDP for evaluating message delivery and communication efficiency in IoT systems.
4. Implemented a dual-redundancy logging system that stores application and sensor data both locally and in MongoDB, ensuring fault-tolerant logging and data reliability for large-scale IoT deployments.
5. Integrating the ELK stack (Elasticsearch, Logstash, and Kibana) for real-time traffic ingestion and visualization, enabling dynamic and interactive graphing capabilities [22].

Section 2 presents the motivation and scope of this work. Section 3 presents the state-of-the-art of existing IoT testbeds and highlights how STGen improves scalability and flexibility. Section 4 outlines the core components of STGen and their role in supporting scalable simulations. Section 5

System	Type	Architecture	Protocol Flex.	Boot Time	Mem	Net Emul.	Max Nodes
Physical Testbeds							
FIT IoT-LAB [20]	Phy	2700+ devices	Hardware (any)	Manual	N/A	Real RF	2700+
SmartSantander [21]	Phy	City sensors	Various	N/A (Always On)	N/A	Real	20k
VM-Based Testbeds							
Gotham [8]	VM	VMs	MQTT, CoAP	>20 min	>15GB	Limited	~ 140
GothX [9]	Hybrid	VM + Docker	MQTT, Kafka	~ 26 min	20GB	Limited	~ 450
Application Frameworks							
IoT-Flock [10]	Frmwk	App-level	MQTT, CoAP	Mod.	N/R	No	Low
PatrIoT [11]	Frmwk	JUnit-based	Code-defined	High	N/R	No	Custom
IoTGemini [12]	Frmwk	Stat. Model	Traffic patterns	Mod.	N/R	No	Traf. Gen
SENS [13]	Frmwk	Generic Sim	Generic	Mod.	N/R	No	~ 8192
STGen (This Work)	Proc	Lightweight	**Plugin-based**	**1.6s**	**2.5GB**	**Yes**	**6k**

Table 2
Comparison of IoT protocol evaluation testbeds and frameworks. (N/R = Not Reported, Mod. = Moderate)

presents the STGen system architecture. Section 6 describes the experimental hardware environment and measurement methodology. Section 7 details the testbed launcher and simulation parameters. Section **??** presents performance metrics demonstrating the testbed's efficiency. Section 8 illustrates potential future work. Finally, Section 9 concludes the work with the future direction of this work.

2. Motivation and Scope

The key motivation of this work is to address the practical challenges faced by the research community in developing and evaluating emerging IoT protocols, particularly during the early stages of prototyping. Researchers often require testbeds to verify message delivery, communication efficiency, and system behavior under realistic traffic conditions, as well as to handle large volumes of sensor-generated data. However, building interoperable and universal IoT testbeds that integrate heterogeneous physical devices and sensors embedded in microcontrollers is costly, time-consuming, and difficult to scale.

As a result, early-stage researchers and students must invest significant effort in learning microcontroller platforms, integrating sensors, and managing physical infrastructure before meaningful protocol evaluation can take place. This overhead substantially slows experimentation and limits rapid iteration, highlighting the need for an affordable, efficient, and flexible environment that supports early-stage validation of IoT protocols without relying on extensive physical deployments.

To address this need, this work introduces STGen, a lightweight and scalable IoT testbed designed to enable rapid prototyping and large-scale experimentation. Instead of depending on physical devices, STGen emulates large numbers of virtual sensor nodes while preserving realistic communication behavior. Its lightweight architecture allows efficient management of thousands of virtual nodes on a single machine, enabling realistic large-scale experimentation without physical hardware.

3. State of the Art

In recent years, several experimental testbeds have been developed to support a large-scale IoT environment for the experimentation of IoT protocols and their applications. For example, FIT IoT-LAB [20] and SmartSantander [21] have advanced the field by providing researchers with extensive sensor networks. However, these testbeds often come with limitations in node accessibility and complex configurations. For instance, the main limitations of SmartSantander are its battery life and limited user reprogramming options, while FIT IoT-LAB focuses on providing access to bare metal but requires careful management of heterogeneous nodes and experiment setup.

Gotham [8] and GothX [9] are tools for creating IoT datasets with both legitimate and malicious traffic. Gotham works with protocols like MQTT and CoAP. GothX adds Kafka and SINETStream for better IoT traffic generation. These tools allow testing across different communication protocols and automate scenario execution. Gotham's emulated scenarios include various IoT environments, such as smart homes, industrial systems, and urban monitoring, rather than just homes and offices. Both tools reduce costs by using virtualization, like VMs and Docker, rather than physical setups. GothX also offers more flexibility with customizable topologies and automatic labeling of datasets.

IoT-Flock [10] is an open-source framework that creates IoT traffic to help develop security solutions for smart homes by mimicking real-world IoT devices. IoT-Flock works with two popular IoT application layer protocols, MQTT and CoAP. However, the paper does not cover memory usage or performance measurements for different numbers of nodes. It also does not discuss limitations or compare resource efficiency with other traffic generation frameworks.

The Patriot [11], has been developed to offer a flexible IoT system testbed. This testbed scales from a physical setup to a simulated environment and supports mixed configurations. The framework is built on JUnit 5, which has been extended for testing interoperability and integration. However,

complex synchronization and orchestration of events often need to be managed explicitly in the test code.

IoTGemini [12] is a framework for synthetic IoT traffic generation that models device functions and network behaviors to enable customized traffic generation. It incorporates a Device Modeling Module and a Traffic Generation Module based on Packet Sequence GAN (PS-GAN) to preserve packet-level sequentiality. IoTGemini achieves high fidelity and usability for downstream tasks such as intrusion detection, anomaly detection, and device fingerprinting.

SENS [13] is a modular simulator for wireless sensor networks that integrates sensor, environment, and network components. It enables realistic modeling of physical environments using tile-based propagation and supports application portability by allowing the same source code to run on simulated and real nodes. SENS provides diagnostic features such as power usage analysis and supports customizable network models, making it suitable for testing WSN applications under diverse environmental and network conditions.

4. STGen Positioning

STGen has the potential to secure a unique position in the IoT evaluation landscape by bridging the gap between heavyweight simulators and physical testbeds. Moreover, it also avoids the shortcomings of both VM-based testbeds and application-specific frameworks. This positioning stems from three key architectural decisions that distinguish STGen from existing tools. First, protocol-agnostic architecture through real library integration. Unlike network simulators as shown in (Table 1) that require complete protocol implementation in simulation-specific languages, STGen uses protocol libraries like Eclipse Paho for MQTT, aiocoap for CoAP, and custom solutions for proprietary protocols. This eliminates weeks of effort and ensures that protocol behavior in STGen experiments matches production deployments. The plugin-based architecture enables researchers to integrate new protocols in hours rather than weeks, with zero modification to STGen's core infrastructure. Second, lightweight process-based design for resource efficiency. Unlike VM-based testbeds (Gotham, GothX in Table 2) that incur substantial virtualization overhead, STGen's process-based architecture minimizes resource consumption while maintaining process isolation and reproducibility. As demonstrated in section 5.2, this architectural choice enables STGen to achieve 99.9% faster initialization (1.6 seconds versus 26 minutes for GothX with comparable node counts) and 87.5% lower memory footprint (2.56 GB versus 20.4 GB). The efficiency gains stem from eliminating VM hypervisor resource overhead, redundant network stack virtualization, and excessive context switching, which enables large-scale experimentation (6,000+ nodes) on commodity hardware without investing in cloud infrastructure. Third, integrated scenario and network awareness. Unlike application-specific frameworks (Table 2) that lack network emulation or provide hardcoded scenarios, STGen offers six predefined scenario templates (smart home, industrial IoT, healthcare wearables, smart agriculture, stress testing, event-driven burst) and five network condition profiles (WiFi, 4G/LTE, LoRaWAN, Congested, Perfect) with realistic link characteristics implemented via Linux NetEm. This scenario-based approach enables reproducible experiments across research groups and facilitates comparative studies where multiple protocols evaluate under identical conditions, a capability that absent in existing frameworks. Compared to physical testbeds (FIT IoT-LAB, SmartSantander in Table 2), STGen provides unlimited experimentation without hardware constraints, scheduling delays, or battery replacement requirements, while sacrificing only the highest-fidelity aspects of real RF propagation and hardware driver validation. For the 80% of IoT protocol research focused on application-layer behavior rather than physical-layer phenomena, this trade-off proves highly favorable. Researchers can iterate rapidly during protocol development (leveraging STGen's speed and flexibility).

This positioning enables STGen to serve as a practical bridge between early-stage protocol prototyping and deployment validation. By providing rapid, realistic application-layer protocol evaluation without infrastructure overhead or implementation burden, STGen fills a critical gap in the IoT research paradigm, where the existing solutions address the issues inadequately.

5. STGen System Architecture

5.1. Scope Definition: Application-Layer Focus

The scope of STGen is constrained to the application layer in order to keep the focus on the evaluation of protocols like MQTT and CoAP. So, it does not cover lower-level behaviors such as radio signal attenuation, MAC layer collisions, or multi-hop routing. The framework focuses on end-to-end metrics like latency, throughput, and system resource costs under controlled conditions via Linux NetEm. Hence, STGen remains lightweight and provides a rapid testing alternative that complements traditional by abstracting away these transport and physical layer complexities.

5.2. Architectural Overview

The architecture of the STGen testbed is designed with an emphasis on scalability, modularity, and customization. Figure 1 illustrates the distributed deployment model, where multiple STGen Core instances can run concurrently. Each core coordinates with numerous emulated sensor and client nodes, which may operate on the same physical machine or be distributed across multiple systems in the network. The high-level software architecture of STGen is shown in Figure 3. This layered model allows the user to customize the configuration according to their need. They can experiment with their protocols under different network configurations, and simulate domain-specific scenarios such as smart homes, smart agriculture, industrial IoT, and more. From the configuration layer users can select protocols and configure the parameters according to their need. During the simulations, the observability layer collects logs, metrics and performance indicators. So, it allows users to evaluate

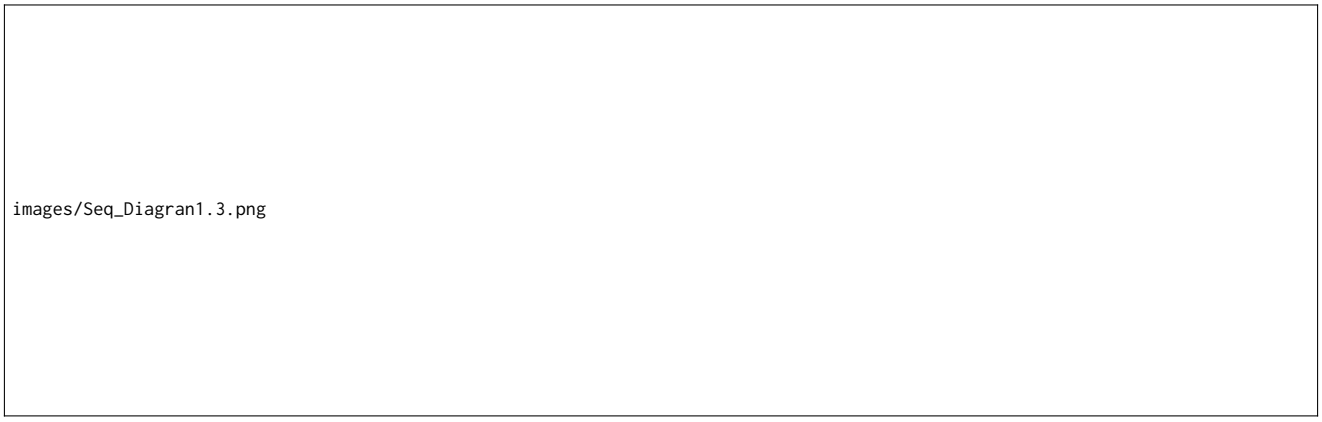

Figure 2: Sequence diagram of STGen testbed illustrating client applications requesting the STGen Core Node with sensor metadata and receiving sensor data over a custom application layer protocol. The STGen core node collects sensor data from the testbed at configurable intervals, stores it in a cloud database, and forwards it to the ELK Stack for real-time traffic analysis.

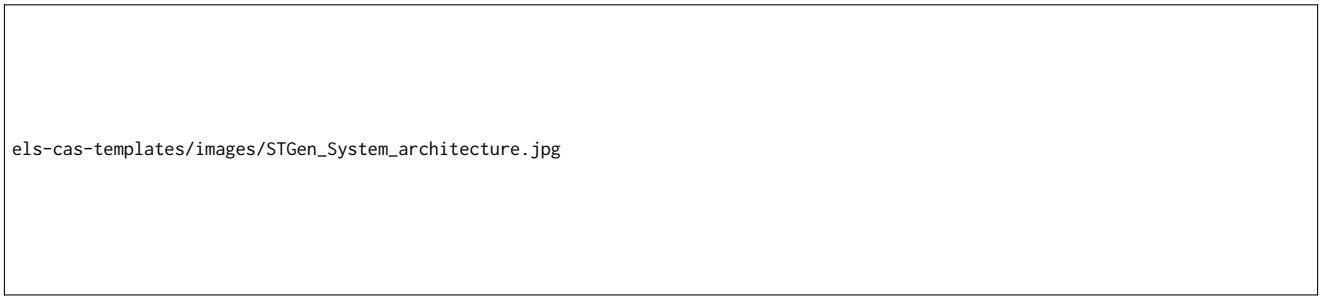

Figure 3: STGen high-level system architecture.

and compare protocol behaviors under diverse workloads and communication patterns. This multi-layer architecture ensures that STGen remains flexible and extensible.

5.3. Core Components

The core components of STGen considered at the early stage of prototyping are described below:

5.3.1. Sensor Profile

Existing IoT testbeds often rely on uniformly random or perfectly periodic values that do not capture realistic sensor behavior. When synthetic data do not align with the physical constraints of the underlying hardware, it ceases to represent a valid sensor measurement and instead degenerates into noise. To address this limitation, STGen generates sensor streams that follow physical laws and hardware-specific dynamics. Instead of producing arbitrary numeric values, STGen emulates the natural constraints and response characteristics of real devices.

1. **Thermal Drift (Ornstein–Uhlenbeck Process)** Real temperature measurements exhibit thermal inertia, meaning that they do not change abruptly. Naively generated random temperatures often produce unrealistic spikes that may trigger false alarms. To avoid this, `STGen` models temperature evolution using the Ornstein–Uhlenbeck (OU) process [23]:

$$dT_t = \theta(\mu - T_t)\,dt + \sigma\,dW_t,$$

where θ represents the thermal relaxation rate (i.e., resistance to rapid change), and σ denotes the noise floor of the hardware. This ensures smooth drift patterns, where injected anomalies remain statistically distinguishable from natural fluctuations.

2. **Hardware Lockouts in PIR Sensors** Standard simulations treat PIR sensors as ideal instantaneous switches; however, real PIR modules (e.g., the HC-SR501) contain capacitors and internal logic that impose a refractory period [24]. After activation, the sensor remains high for a dwell time (T_{dwell}) and then becomes unresponsive for a blocking time (T_{blocking}). The minimum permissible trigger interval is

$$\Delta t_{\text{trig}} \geq T_{\text{dwell}} + T_{\text{blocking}} \approx 5.5\,\text{s}.$$

This prevents unrealistic rapid bursts of motion events, ensuring that energy consumption and packet rates reflect the limitations of real PIR hardware [25].

3. **Noise Floor Injection for Accelerometers** In vibration sensing, perfectly smooth signals are unrealistic. Instead of adding arbitrary random noise, `STGen` derives the RMS noise from the MPU-6050 datasheet [26]:

$$\sigma_{\text{noise}} = \text{NSD} \times \sqrt{BW \times 1.6},$$

where NSD is the Noise Spectral Density and BW is the sensor bandwidth. The resulting noise is injected

Figure 4: Real-Time Delay Statistics of Network Traffic in STGen Traffic Analyzer – This graph shows the minimum, average, standard deviation, and 95th percentile of frame time delta (delay) over time, visualized using the ELK stack for real-time traffic analysis.

as a Gaussian process,

$$\mathcal{N}(0, \sigma^2_{\text{noise}})$$

ensuring that accelerometer readings exhibit realistic micro-vibrations.

4. **Kinematic Clamping for GPS Sensors** We enforce a velocity-based spatial clamp similar to mobility-constrained models [27], using standard geodesic approximations of 111 km per degree [28]. To maintain realism, `STGen` applies a velocity clamp:

$$|\Delta_{\text{lat, lon}}| \leq \frac{v_{\max} \Delta t}{111{,}000},$$

where $v_{\max}$ is the maximum allowable user or asset speed. This constraint prevents nonphysical transitions and maintains plausible mobility trajectories.

5. **Traffic Shaping Using the Weibull Distribution** IoT traffic seldom follows a Constant Bit Rate (CBR) pattern. Instead, devices typically wake up, sense, and transmit in irregular bursts. To model this behavior, `STGen` generates inter-arrival times (IATs) using a Weibull distribution with shape parameter $k \approx 0.8$ [29]:

$$f(t; \lambda, k) = \frac{k}{\lambda} \left(\frac{t}{\lambda} \right)^{k-1} e^{-(t/\lambda)^k}.$$

When $k < 1$, the distribution produces long quiet periods followed by short bursts of packets—a characteristic trait of real IoT workloads. This burstiness reveals congestion and queue overflows that are otherwise hidden in smooth CBR simulations.

To unify these physical constraints and statistical models into a coherent execution flow, the sensor node logic is implemented as shown in Algorithm 1.

5.3.2. STGen Core:

The STGen core is the central component of the STGen design, located between the client and the sensor nodes. It includes a node registry that records the states of both the client and the sensor nodes. STGen core acts as a sink node, accumulating data from the sensor nodes and facilitating delivery to the client application through the public Internet. For the purpose of experimental validation, the STGen core is equipped with a custom-designed, lightweight application-layer protocol based on the publish/subscribe (pub/sub) communication model. Upon receiving a sensor subscription request from a client, the protocol persistently records the client's state and subscription metadata. As multiple clients subscribe to one or more sensors, the core maintains a concurrent and scalable state registry for all active subscriptions, ensuring efficient and timely dissemination of sensor data to the relevant subscribers as soon as the data becomes available. In addition, the STGen core stores the collected sensor data and application logs on the local hard disk and periodically stores the sensor data in MongoDB at runtime for further use.

5.3.3. Flexible Configuration

The STGen testbed can operate in both command-line interface (CLI) mode and Web server mode, offering flexible configuration options. The Web-based graphical user interface (GUI) provides a convenient means of configuring and accessing the testbed without requiring prior familiarity with

Algorithm 1 Sensor Data Generation Loop

Input: Sensor Type S_{type}, Config $\mathcal{C}$ $(\theta, \mu, \sigma, T_{dwell}, T_{blocking}, \text{NSD}, \text{BW}, v_{\max}, \lambda, k)$
Output: Stream of sensor packets
$V_t \leftarrow \text{GetInitialValue}(S_{type})$
$t_{last} \leftarrow \text{CurrentTime}()$
while Simulation is Active **do**
  $t_{now} \leftarrow \text{CurrentTime}()$
  $\Delta t \leftarrow t_{now} - t_{last}$
  if $S_{type} ==$ Temperature **then** ▷ Thermal Drift (OU Process)
    $dW_t \leftarrow \text{GaussianRandom}(0, 1)$
    $V_t \leftarrow V_t + \theta(\mu - V_t)\Delta t + \sigma dW_t$
  else if $S_{type} ==$ PIR **then** ▷ Hardware Lockout Logic
    if $t_{now} < t_{last\_trigger} + T_{dwell} + T_{blocking}$ **then**
      continue ▷ Sensor is in refractory period
    end if
    $t_{last\_trigger} \leftarrow t_{now}$
    $V_t \leftarrow \text{DetectMotion}()$
  else if $S_{type} ==$ Accelerometer **then** ▷ Noise Floor Injection
    $\sigma_{noise} \leftarrow \text{NSD} \times \sqrt{\text{BW} \times 1.6}$
    $Noise \leftarrow \text{GaussianRandom}(0, \sigma^2_{noise})$
    $V_t \leftarrow \text{GetBaseReading}() + Noise$
  else if $S_{type} ==$ GPS **then** ▷ Kinematic Clamping
    $V_{next} \leftarrow \text{GenerateCandidateCoord}()$
    $\Delta_{deg} \leftarrow |V_{next} - V_t|$
    $Limit \leftarrow \frac{v_{\max} \times \Delta t}{111{,}000}$
    $Limit \leftarrow \frac{v_{\max} \times \Delta t}{111{,}000}$
    if $\Delta_{deg} > Limit$ **then**
      $V_t \leftarrow V_t + \text{Sign}(V_{next} - V_t) \times Limit$
    else
      $V_t \leftarrow V_{next}$
    end if
  end if
  $\text{Packet} \leftarrow \text{ConstructPayload}(V_t)$
  $\text{SendUDP}(\text{STGenCoreIP}, \text{Packet})$
    ▷ Traffic Shaping (Weibull Distribution)
  $IAT \leftarrow \text{Weibull}(\lambda, k)$
  $\text{Sleep}(\text{IAT})$
  $t_{last} \leftarrow t_{now}$
end while

the CLI. User selections made through the Web interface are translated into API calls, which are subsequently converted by the Web server into CLI commands to execute the required setup and control actions.

Overall, STGen is designed as a user-friendly tool with simple and intuitive configuration parameters. Its adaptability across a wide range of IoT research domains enables researchers and developers to model complex experimental settings efficiently and with minimal effort.

To further enhance usability and automation, users can run the backend service that exposes these REST endpoints, converting REST API query parameters into the appropriate CLI commands, an approach that is particularly beneficial for those who prefer to avoid direct terminal command execution. This backend service simplifies and automates processes such as sensor, client, and core creation, reducing manual intervention and enhancing reproducibility. Additionally, API calls stream real-time application logs directly to the browser when utilizing the front-end service, providing transparent insight into ongoing operations. For researchers relying solely on the REST API endpoints, a comprehensive OpenAPI specification has been defined, which is available through the Swagger UI at /swagger-ui/index.html. This includes detailed documentation, interactive testing capabilities, and version control features, further facilitating integration, reproducibility, and ease of use. Moreover, for those who prefer direct interaction with the CLI, all available CLI commands are provided in the appendix section for reference.

5.3.4. *Real-Time Data Transmission and Customization*

A major challenge in IoT networks stems from the heterogeneous nature of the sensor, actuators and devices encompassing "heterogeneity of devices", "heterogeneity in data formats", and "interoperability issues arising from heterogeneity" [30–32]. When forecasting heterogeneous IoT data values, temporal and spatial dependencies become particularly evident, as data patterns are influenced by numerous context-specific factors. Our system incorporates a default transmission rate that can be modified through input parameters, allowing researchers to customize the rate for particular sensor types or applications, hence improving the versatility of the testbed.

5.3.5. *Client Application*

Client applications of the STGen platform are connected to the STGen core to interact indirectly with the sensor nodes. Clients can retrieve STGen core traffic by submitting a request for sensor nodes through the custom application layer protocol. Clients initiate subscriptions by sending sensor node requests to the STGen core, which manages the state and data flow. This architectural separation of concerns ensures that the core is solely responsible for handling the complexities of sensor registration, state management, and data dissemination, while clients remain focused on consuming and processing sensor data. This architecture accommodates numerous clients who are interfacing with the identical STGen core.

5.4. STGen as Hybrid Network Model

In STGen, the emulated Wireless Sensor Network (WSN) nodes are not constrained to a single physical machine. These software-defined nodes can be distributed across multiple systems and communicate with the STGen Core either via wired channels (such as Ethernet or LAN switches) or wirelessly (e.g., Wi-Fi). This architectural flexibility enables the formation of spatially separated WSN clusters that still maintain synchronized communication with the

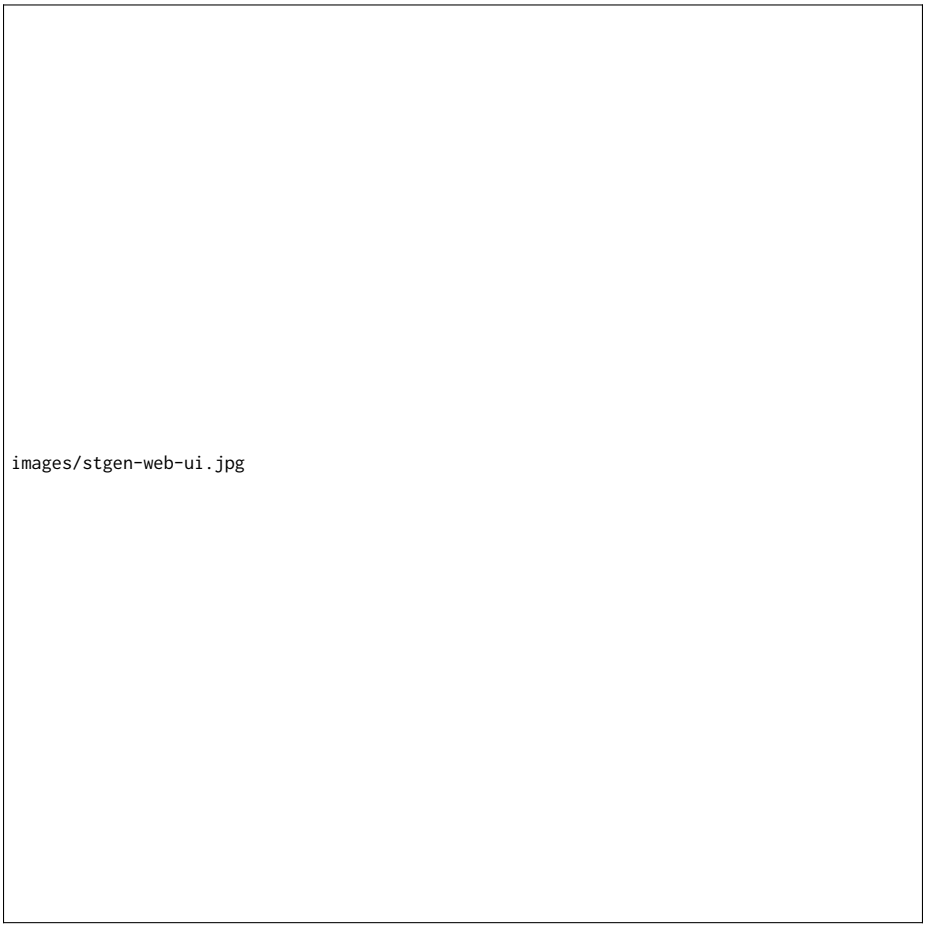

Figure 5: STGen Testbed Launcher Web UI.

core. The STGen Core itself is decoupled from the emulated node's host systems, thus enabling a hybrid network model that integrates both wired and wireless communication modalities. Building on this foundation, STGen serves as an IoT testbed, as illustrated in Figure 1, offering a powerful platform for traffic generation and protocol evaluation in hybrid IoT networks. The testbed is designed to simulate thousands of sensors and actuators, which, in the real world, are typically resource-constrained [33], that is, are operated by batteries and have a limited amount of storage and processing capabilities. Therefore, communications in IoT are sensitive to time. In such scenarios, the User Datagram Protocol (UDP) [34] is the appropriate transport protocol for the communication of sensor data in WSN. Within Wireless Sensor Networks (WSNs), sensor nodes are generally deployed within the boundaries of a private network. As a result, direct communication with external systems is generally mediated through a designated sink node. In the proposed hybrid network model, a physical machine is designated as the STGen core, which assumes the role of the sink node, effectively offloading the computational and communication burdens from the emulated sensor nodes.

5.5. Three Tier Configuration Model

As highlighted in Table 2, existing frameworks often suffer from high initialization times and complex deployment procedures. To address this, STGen employs a composable configuration model that partitions experiments into three independent tiers. This structural decoupling ensures that protocol logic remains distinct from application scenarios and environmental constraints, as illustrated in Figure 6.

- **Tier 1: Protocol Configuration.** This layer governs transport specific parameters. It defines how data is delivered, such as toggling MQTT QoS levels or enabling SRTP encryption, without altering the data generation logic. Figure 6a illustrates this separation, defining a CoAP configuration that specifies a Weibull distribution for inter arrival times while remaining agnostic to the specific sensors being simulated.
- **Tier 2: Scenario Configuration.** This defines the application context. It dictates the active sensor types, node counts, and specific traffic patterns. As shown in Figure 6b, a "Smart Agriculture" scenario specifies low bandwidth temperature and soil moisture sensors. Crucially, this file acts as the bridge, referencing the target network profile to link the tiers at runtime.

- **Tier 3: Network Profile.** This layer controls the environment. It applies specific link characteristics, such as latency, jitter, and packet loss, to simulate real world network instability. Figure 6c demonstrates a "Congested Network" profile, imposing high latency (300ms) and significant packet loss (15%) to stress test the chosen protocol.

These tiers merge at runtime to form a complete executable experiment. This approach significantly reduces the configuration workload by converting the effort from a multiplicative factor to an additive one. For instance, evaluating 4 protocols across 6 scenarios under 5 network conditions requires defining only 15 source files ($4 + 6 + 5$) rather than managing 120 separate monolithic configuration files ($4 \times 6 \times 5$).

5.5.1. Dual-Archiving for Redundancy and Scalability:

As the STGen core is ephemeral, it is essential to aggregate the data transmitted by the different sensor nodes in a storage system. For this, the STGen architecture performs two distinct tasks: first, distributing the data to MongoDB for efficient retrieval and analysis, and second, ensuring that the sensor data is aggregated locally. This dual-archiving strategy ensures redundancy and facilitates access in real-time or temporal analysis later on. The network traffic log is indexed and stored in Elasticsearch for real-time visualization and meaningful insights.

5.5.2. Realtime Network Log Analysis:

To determine the efficacy of the underlying protocols, network performance analysis is crucial during client-sensor interactions through the STGen middleware. Capturing real-time network traffic at the designated STGen middleware ports is achieved using TShark, the command-line version of Wireshark. However, raw traffic data lacks structure and requires transformation for meaningful analysis, necessitating the identification of key performance metrics such as latency, jitter, packet loss, retransmissions, and protocol overhead. To address this, Logstash preprocesses and transforms the raw traffic data into a structured format in real time, extracting relevant insights before transmitting it to Elasticsearch, where the data is indexed and stored for efficient querying and analysis [35]. The STGen core node captures this network traffic by extracting real-time data from specific UDP ports and outputting it in JSON format, making it easier to feed into Elasticsearch for further processing. Researchers evaluating communication protocols can leverage this indexed data to analyze protocol performance under various conditions. By integrating Elasticsearch with Kibana, researchers gain access to dynamic, real-time visualizations that enable effective benchmarking of network behaviors. Kibana's dashboards, time-series charts, and heatmaps provide insights into latency trends, traffic congestion points, and packet size distributions across different sensor nodes. By tailoring these visualizations to emphasize specific network performance metrics, researchers can extract actionable insights from STGen's network traffic, optimizing both protocol design and middleware efficiency. Figure 4 illustrates real-time delay statistics visualized in Kibana, showcasing the effectiveness of this TShark and ELK workflow in STGen's testbed analysis.

5.5.3. Interaction Layers and User Interface

STGen offers functionality via three major interfaces.

1. **Command Line Interface (CLI):** STGen primarily offers a CLI based interface allowing users to configure the system parameters. Users can launch the entire testbed and customize its behavior through command-line arguments.
2. **Web-Based Graphical User Interface:** The Web GUI can be characterized as a graphical interface to the underlying system, which makes configuration management easier. It connects directly to the backend services and it transfers user inputs into standard API calls.
3. **REST API on OpenAPI Standards:** The main functionality of it is provided as a RESTful API. It is documented in Swagger UI. The standardized interface allows interactive testing.

6. Experimental Configuration

To ensure a comprehensive evaluation of STGen across different resource constraints and deployment topologies, three distinct hardware configurations were utilized.

6.1. Hardware Environment

The hardware specifications for the workstations used in this study are detailed in Table 3.

- **Workstation A (High-Performance Host):** Utilized for generating the baseline resource consumption graphs (Figure 8) and validating stability under maximum load (up to 36 GB RAM).
- **Workstation B (Standard Testbench):** Served as the primary environment for the recent single-machine scalability analysis (Section 7.1) and protocol benchmarking (Section 7.2). This machine represents a typical commodity server environment (16 GB RAM).
- **Workstation C (Distributed Client Node):** Employed exclusively in the distributed setup to emulate external client traffic, physically separated from the STGen core.

6.2. Measurement Methodology

- **Startup Time:** Measured using `time.perf_counter()` with 1 ms precision; defined as the elapsed time between the orchestration launch command and the initialization of all client threads.

(a) Tier 1: Protocol (CoAP) (b) Tier 2: Scenario Logic (c) Tier 3: Network Profile

Figure 6: JSON configuration examples demonstrating the modular Three Tier architecture. Figure 6a defines protocol mechanics; Figure 6b defines the application logic and links the tiers; Figure 6c sets environmental constraints.

Table 3
Hardware Specifications of Experimental Devices

Spec	Workstation A	Workstation B	Workstation C
Role	Baseline Stress Testing	**Scalability & Protocols**	Distributed Client
CPU	AMD Ryzen 3600X	AMD Ryzen 5600G	Intel Core i5-8400
RAM	36 GB DDR4	16 GB DDR4	8 GB DDR4
OS	Ubuntu 24.10	Ubuntu 24.10	Windows 11

- **Memory Usage:** Resident Set Size (RSS) measured via `psutil` every 0.1 seconds, aggregated across the parent process and all child processes.
- **CPU Load:** Instantaneous CPU utilization sampled at 10 Hz (0.1 s interval); the reported value represents the peak utilization observed during the test window.
- **Throughput & Latency:** Throughput is derived from valid server-side receipts over the test duration. Bandwidth (Mbps) is calculated assuming a fixed packet size of 112 bytes (12-byte header + 100-byte payload). Latency is computed as Δt between the client-side send timestamp and the server-side write timestamp.

6.3. Network Emulation Profiles

All protocol comparisons were conducted using the Linux NetEm tool on Workstation B to inject artificial network impairments. Table 4 lists the NetEm network condition presets.

6.4. Simulation Parameter Description

This sub-section details the configuration parameters required for orchestrating the STGen platform. To facilitate a modular and reproducible experimental design, STGen utilizes a tiered configuration schema that decouples communication logic from environmental constraints. These parameters are essential for accurately modeling sensor behavior, configuring core server orchestration, and managing client-side subscriptions.

As summarized in Table 6, the configuration is partitioned into three functional tiers: the Protocol Tier, which governs transport-layer selection and addressing; the Scenario Tier, which defines the scale, duration, and specific sensor types of the emulation; and the Network Tier, which manages the emulation of physical link characteristics.

Furthermore, to maintain consistency across comparative studies, STGen provides pre-configured Network Condition Profiles. As detailed in Table 4, these profiles (e.g., WiFi, 4G/LTE, and LoRaWAN) define specific parameters for delay, jitter, packet loss, and bandwidth. This structured approach allows the platform to be deployed seamlessly in both local and distributed environments, enabling researchers to evaluate protocol performance under realistic and varying levels of network stress.

7. Experiment

The STGen Testbed is platform independent, easy to control, and easy to configure through the Graphical User Interface (GUI) and Command Line Interface (CLI).

7.1. Single-Machine Scalability

This experiment evaluates the scalability of the STGen platform when executed entirely on a single workstation. Tests were conducted on Workstation B (16 GB RAM) to demonstrate performance on standard hardware. The objective is twofold: (i) to validate the lightweight design of STGen under increasing numbers of sensor nodes, and (ii) to benchmark its performance limits. All tests were conducted using a custom lightweight UDP protocol to isolate the orchestrator's overhead. The system was configured with a perfect network profile and a runtime of 10 seconds per iteration.

Scalability Analysis:
Linear Startup Scaling: The startup time grows linearly with respect to the number of sensor nodes, demonstrating the efficiency of the parallelized initialization routine. A linear fit over the data points yields:

$$T(n) \approx 0.0032n + 0.3 \quad (R^2 \approx 0.99)$$

This indicates a marginal initialization cost of approximately 3.2 ms per node, allowing the testbed to instantiate 6,000 nodes in under 20 seconds.

Memory Efficiency: The memory footprint exhibits strict linearity, validating the absence of memory leaks in the client generation logic:

$$M(n) \approx 0.0015n + 0.04 \text{ GB}$$

This corresponds to an average overhead of just 1.5 MB per sensor node. This high efficiency allows STGen to simulate thousands of independent nodes on commodity hardware without exhausting physical RAM resources.

Table 4
Network Condition Profiles

Profile	Delay (ms)	Jitter (ms)	Loss (%)	Bandwidth (Kbps)
WiFi	20	5	0.5	54000
4G/LTE	70	10	1.2	15000
LoRaWAN	600	40	2.5	50
Congested	150	30	5.0	1000
Perfect	1	0	0	Unbounded

Table 5
Simulation Parameters for STGen Sensor Testbed

Parameter	Description
Protocol	IoT protocol to evaluate (`mqtt`, `coap`, `srtp`, `custom UDP`)
Mode	Operational mode: active `(sensor generation)` or passive `(traffic replay)`
Server IP	IP address of the broker/server to which sensors connect
Server Port	Port number on the broker/server for sensor connections
Number of Clients	Number of simulated sensor clients/nodes
Duration	Total experiment duration in seconds
Sensors	List of sensor types to simulate (`temp`, `humidity`, `gps`, `motion`, `light`, `etc.`)
Traffic Pattern	Per-sensor transmission rate in Hz and burst behavior
Network Conditions	Emulated network characteristics (`latency`, `jitter`, `bandwidth`, `packet loss`)
Network Profile	Pre-configured network environment (`wifi`, `4g`, `lorawan`, `congested`, `perfect`)
Failure Injection	Packet loss rates and client crash schedules for fault testing
Deployment Mode	Single-machine or distributed `(core node + remote sensor nodes)`

Table 6
Configuration Parameters for STGen Simulation Setups

Parameter	Description
Protocol Tier	
Protocol	IoT protocol to evaluate (`mqtt`, `coap`, `srtp`, `custom udp`)
Mode	active `(traffic generation)` or passive `(traffic replay)`
Server IP/Port	Networking coordinates for the STGen core or broker
Scenario Tier	
Num. Clients	Total number of simulated concurrent sensor nodes
Duration	Total experiment execution time in seconds
Sensors	List of sensor types (`temp`, `humidity`, `gps`, `camera`)
Role	core `(orchestrator)` or sensor `(generator)`
Network Tier	
Profile	Presets: `wifi`, `4g`, `lorawan`, `congested`, `perfect`
Network Stats	Manual overrides for Latency (ms), Loss (%), and Jitter

CPU Utilization and Throughput: Contrary to typical simulation tools that become CPU-bound due to context switching, STGen effectively utilizes available CPU cycles to drive network traffic. CPU usage peaked at 87% during the 6,000-node test, correlating with a massive throughput of 154,823 messages per second. This confirms that the framework is capable of generating high-velocity data streams that saturate the processing bandwidth (138.7 Mbps) before hitting orchestration bottlenecks.

System Breaking Point: As shown in Table 7, STGen maintains linear resource scaling on Workstation B. At 6,000 nodes, memory usage reached 9.03 GB, leaving ample headroom on the 16 GB machine. This confirms that STGen can comfortably simulate massive topologies on mid-range hardware without the "System Breaking Point" issues often observed in VM-based approaches.

7.2. Protocol Performance Benchmarking

To demonstrate the capability of STGen to evaluate protocol suitability under constrained conditions, we conducted a comparative analysis of MQTT (TCP-based) and CoAP (UDP-based).

Experimental Design

We utilized the Smart Agriculture scenario to simulate low-power sensor nodes (e.g., Soil Moisture, Temperature) transmitting periodic telemetry.

- **Network Profile:** "Congested" (150 ms latency ± 30 ms jitter, 5% packet loss).
- **Objective:** Measure the impact of transport-layer overhead on battery-critical transmission time.

Table 7
Scalability Results of STGen on Workstation B (16 GB RAM)

Nodes	Startup (s)	Memory (GB)	CPU Peak (%)	Throughput (msg/s)	Throughput (Mbps)	Lat P95 (ms)
100	0.61	0.19	36	14,120	12.7	0.2
500	1.66	0.79	52	19,619	17.6	0.2
1000	2.92	1.54	60	27,379	24.5	0.1
2000	5.94	3.04	73	45,972	41.2	0.1
3000	8.70	4.54	78	66,809	59.9	0.3
6000	19.46	9.03	87	154,823	138.7	0.5

*Note: Packet loss remained at 0.0% across all scalability test cases.

Table 8
Protocol Latency Analysis (Smart Agriculture Scenario)

Protocol	Transport	Avg Latency	P95 Latency	Stability
CoAP	UDP	**291.31 ms**	**315.09 ms**	Stable
MQTT	TCP	624.87 ms	1,198.01 ms	Unstable

*Network Condition: 150ms Delay, 5% Loss.

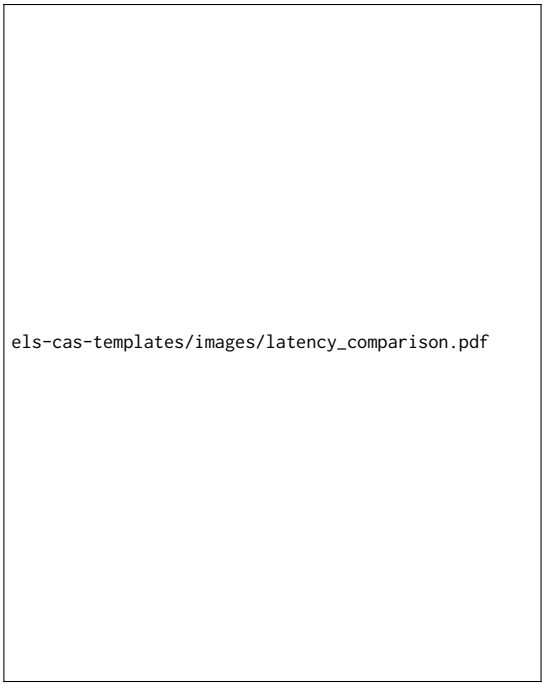

Figure 7: Protocol Latency Comparison: Average, Median, and P95 Latency under Congested Network conditions. MQTT exhibits severe tail latency (P95) due to TCP retransmissions.

Performance Analysis

Latency Amplification in TCP: As summarized in Table 8, CoAP demonstrated superior performance with an average latency of 291.31 ms. This aligns closely with the theoretical Round-Trip Time (RTT) of the emulated link (2× 150ms), confirming that CoAP introduces minimal overhead beyond physical propagation. In contrast, MQTT suffered from significant latency amplification, averaging 624.87 ms (+114% increase). This is attributed to the TCP connection overhead and the need for multiple handshake packets before payload delivery.

Impact of Packet Loss (The "P95" Spike): Figure 7 highlights the critical difference in stability. While CoAP's 95th percentile (P95) latency remained low (315 ms), MQTT exhibited a massive spike to 1,198 ms. This 4x increase in tail latency is caused by TCP's exponential backoff algorithms when handling the 5% packet loss. For a battery-powered sensor, this means the radio must remain active 4x longer for a single transmission, significantly reducing device battery life.

7.3. Distributed Deployment Performance

To evaluate the system's behavior in a physical network environment, we deployed the STGen core on Workstation B (Orchestrator) and a sensor load generator on Workstation C (Laptop). The components communicated via a local Gigabit Ethernet connection.

Network Validation

The distributed deployment successfully validated STGen's ability to coordinate workloads across physical machines. As detailed in Table 9, the remote sensor node generated 2,757 messages over a 30-second interval. The system achieved a 96.1% delivery rate with an average end-to-end latency of 3.38 ms (median 3.30 ms, P95 4.28 ms). This represents a 6 to 7 times increase in latency compared to the localhost baseline, which is less than 0.5 ms, confirming that data transmission occurred over the physical network infrastructure rather than inter-process communication. Furthermore, the standard deviation of the latency time was quite low ($\sigma \approx 0.5$ ms), indicating that the connection was stable. Combined with the observed 3.9% packet loss, these results give a good representation of how data behaves in a real-life wireless LAN environment.

Resource Overhead Analysis

Transitioning to a distributed architecture introduces a measurable "Network Tax" on the Core node due to serialization and TCP/IP stack processing. As shown in Table 10, the Core's average CPU usage increased from 2.3% to 5.1%

Table 9
Distributed Deployment Validation Results

Metric	Single Machine	Distributed	Observation
Latency (Avg)	< 0.5 ms	3.38 ms	Real Wire Delay
Packet Loss	0%	3.9%	Realistic Noise
Throughput	Loopback	12 Mbps	Physical I/O
Core Role	Full Stack	Broker Only	Workload Offloaded

Table 10
Resource Utilization: Single-Machine vs. Distributed Core

Metric	Single Machine	Distributed (Core)	Change
CPU Usage (avg)	2.3%	5.1%	+118%
CPU Usage (peak)	3.9%	6.5%	+67%
Memory (RAM)	104 MB	408 MB	+291%
Network I/O	Loopback (lo)	Ethernet/WiFi	Real Traffic

(+118%), while memory footprint rose by 291% (to 408 MB) due to network buffer allocation.

Scalability Trade-off: While the distributed setup incurs higher per-node resource costs on the Core, it provides the critical advantage of horizontal scalability. By offloading the memory-intensive sensor simulation logic to external machines (Workstation C), the total network size is no longer bounded by the RAM of a single host, allowing STGen to scale beyond the 6,000-node limit observed in the single-machine experiments.

7.4. Maximum Load Stress Testing

To evaluate the absolute capacity limits of the STGen architecture, stress tests were conducted on Workstation A (36 GB RAM). Figure 8 illustrates the startup time and memory footprint as a function of node count across different memory configurations.

1. Time taken to boot STGen platform consisting of six thousand sensors: 21.981 seconds
2. Average latency to retrieve information from a particular sensor: 0.01965 seconds
3. Memory consumption by the STGen platform with the 6K nodes (including swap memory): 38.3 GB

Since the sensor nodes of the STGen are predominantly memory-constrained, the simulations were primarily conducted on a machine with 36 GB of RAM, with additional tests performed on machines with 16 GB and 28 GB of RAM. Each experiment was performed for 30 seconds while varying the number of nodes and maintaining an equal distribution of each sensor type. This procedure was carried out ten times to guarantee predictable outcomes. We observe from Figure 8a and 8b that decreasing memory not only limits the number of deployable nodes it also affects boot times with available memory. The root cause behind this observation may be the operating system, which spends extra time managing memory allocations.

The experiment of benchmarking the startup of 7k sensor nodes has failed due to our experiment's limited system resources. We observe that initializing more than 6k sensor nodes causes issues with our experimental setup's available RAM, swap, and virtual memory. However, our workstation can handle up to 6k nodes with a memory footprint of about 38.3 GB, while the physical memory is only 36 GB. The benchmark is run with CLI commands, as the backend (Java) and frontend (Next.js) of the web application cause extra memory overhead. The `system.process.cpu.total.norm.pct` metric, as shown in Figure 9, represents the total percentage of CPU time consumed by 3000 sensor node processes. It was observed to peak at 0.6% during sensor startup and remain at 0% during subsequent periods when simulating thousands of sensor nodes concurrently. This behavior indicates minimal CPU usage per process, particularly during periods of low system load. Since the modules are decoupled, and in real-world scenarios, clients can request sensor data at any time, each module operates with an independent simulation period, ensuring they do not interfere with one another. The execution time T for generating sensor data is determined by the simulation time period S:

$$T = S \tag{1}$$

Every sensor type on the STGen testbed is programmed with the default transmission rate. The waiting time between consecutive data transmissions, which defines the data rate, can be adjusted without altering the execution time T. Given an initial transmission interval I (the default timeout for a sensor type), the adjusted interval I' is determined by a percentage factor P (ranging from 1 to 100), which will be taken as input through the CLI arguments:

$$I' = I \times \frac{100}{P} \tag{2}$$

An increase in P results in a decrease in I', leading to a higher data transmission rate. However, a decrease in P increases I', reducing the transmission frequency of the data. Since the execution time T remains unchanged, this approach allows dynamic control over data rates without interfering with the real-time execution of the sensor modules.

(a) The start-up time of the STGen platform for varying the number of sensor nodes from 1K to 6K under different RAM configurations.

(b) Memory footprint of the STGen platform for varying the number of sensor nodes from 1K to 6K under different RAM configurations.

Figure 8: Performance Analysis of the STGen Platform Under Varying Sensor Node Configurations

Figure 9: CPU (6 core) Utilization percentage of 3000 concurrent sensor node processes over a 5-minute monitoring interval.

All simulations were conducted on five different components, such as temperature, humidity, GPS, switch, and camera, to ensure systematic analysis. Figure 10a represents the number of packets transmitted over one-hour time intervals in a six-hour-long simulation. The packets are categorized according to the contributing components: camera, humidity, temperature, device, and GPS. The total number of packets remains relatively stable across the time intervals, with some variations in each sensor type's contribution, which is shown in Figure 10b. The camera sensors are the most data-intensive due to high-resolution data transmission, whereas devices and GPS sensors contribute significantly less.

7.5. Comparison With GothX and Gotham

In this article, STGen has been compared with the two most popular testbeds, Gotham and GothX, respectively, with the latter being developed as an improved version of the former to meet further demands and challenges in IoT

(a) Contribution of each sensors over 1 hour time intervals (1 hour)

(b) Data Transmission Frequency of each sensors

Figure 10: Packet Distribution Analysis of different sensors deployed on the STGen Testbed.

Figure 11: Memory Footprint Comparison: STGen vs Gotham for Varying Sensor Node Counts.

simulations. The purpose of this comparison is to demonstrate the effectiveness of STGen in terms of resource utilization and startup time in performing large-scale simulations. As shown in Figure 11, STGen consistently outperforms Gotham in terms of memory usage, with reductions ranging between 84% and 89% as the number of nodes increases. The percentage reduction decreases slightly as the number of nodes increases. This result indicates a diminishing return, but still a substantial reduction overall. The lightweight design of STGen results in approximately 87.5% lower memory usage, requiring just 2.56 GB of RAM to initialize 450 nodes, compared to the 20.4 GB exploited by GothX. GothX takes approximately 26 minutes to set up a large topology with four VM nodes and 498 Docker nodes, whereas STGen takes just 1.645 seconds to initialize 500 sensor nodes along with one client and one core node. This represents a significant improvement, reducing the setup time by 99.9%. Despite the fact that virtual machines (VMs) and Docker nodes are effective in complex virtualized environments, it is clear that Gotham and GothX incur significant overhead due to their reliance on them. The lightweight architecture of STGen thus provides a more efficient and rapid initialization process, making it highly suitable for scenarios requiring quick setup and scalable IoT simulations.

Gotham and GothX enable the deployment of complex network topologies by leveraging virtual machine (VM)-based nodes, each requiring substantial computational resources. However, this approach incurs significant processing overhead, which makes large-scale IoT simulations on a single machine computationally impractical. The allocation of resources to multiple VM nodes leads to excessive CPU and memory consumption, resulting in system bottlenecks and resource contention. Consequently, the high computational demand restricts scalability and degrades the performance of other concurrent processes running on the system. To maintain a lightweight architecture, STGen does

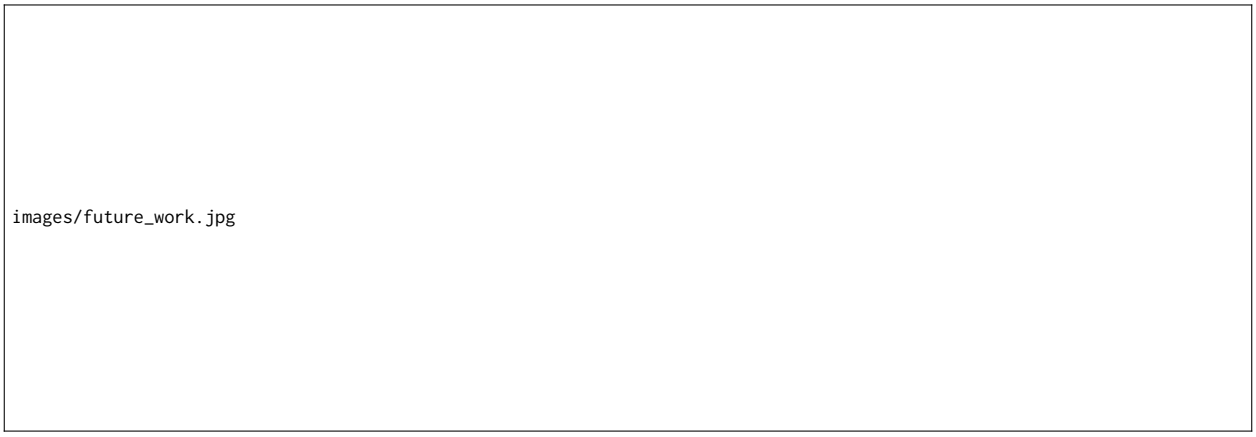

Figure 12: Universal IoT Middleware Architecture: Foundation and Future Work Directions

not natively support the deployment of complex network topologies; rather, it prioritizes a streamlined setup process to facilitate efficient protocol evaluation and experimentation.

The modular architecture and lightweight design features of STGen make it a scalable and extensible experimentation platform, enabling systematic evaluation of IoT protocols and adaptation to evolving research requirements. STGen provides a robust foundation for conducting experiments at varying scales by bridging controlled testing environments with realistic IoT scenarios. Its design streamlines the setup and maintenance of test environments, reducing configuration overhead and facilitating seamless experimentation with heterogeneous sensor deployments.

Furthermore, the Web-based user interface enhances usability by offering intuitive interaction and visualization of IoT experiments through a user-friendly web application. The modular architecture also supports distributed deployment of the STGen ecosystem across multiple premises, allowing experiments to scale from thousands to potentially millions of virtual sensor nodes with minimal configuration effort.

8. Future Work

Figure 12 presents future work focused on building a protocol-agnostic middleware for IoT devices. The middleware will bridge heterogeneous IoT devices and real-world applications by abstracting underlying protocol complexities. It will support application-layer protocols, such as MQTT [36], and CoAP [37] , through protocol adapters.

Future research will specifically focus on three key areas of improvement:

- The STGen high-fidelity network emulation is currently based on the Linux tc (Traffic Control) subsystem. As a result of this, capabilities such as latency injection and bandwidth throttling can only be achieved in a Linux environment. To provide a consistent experience across both the Windows and macOS platforms, we are actively investigating the use of automating WSL2 integration.
- A centralized Web UI will be re-integrated to provide a no-code environment for experiment design. This layer will provide deep-link integration with the ELK stack, allowing researchers to visually monitor P95 latency trends and system health across thousands of emulated nodes in real-time.
- Through incorporating Artificial Intelligence (AI) and Reinforcement Learning (RL) into the aforementioned middleware, future iterations of STGen will be capable of dynamic protocol switching. This will enable the optimization for energy efficiency or latency as required through autonomous adaptation.

In short, these updates will take STGen beyond simple performance testing. By concentrating on security and scalability, we hope to make it a general-purpose platform that can be used to connect all kinds of sensor networks to existing industrial systems in the real world with security.

9. Conclusion

This paper introduced STGen, a scalable and cost-effective sensor traffic generator designed to facilitate the experimentation of IoT protocols by emulating wireless sensor networks (WSNs) in a hybrid environment that addresses key challenges associated with the deployment of real-world testbeds, including financial constraints, resource limitations, and scalability problems, by providing a lightweight yet powerful alternative. Its modular design enables researchers to conduct empirical studies on IoT protocols, which also allow new integrations or enhancements to be developed and deployed in isolation without affecting other modules, such as the integration of the ELK stack, enabling real-time traffic analysis, offering immediate insights into sensor activity and protocol behavior. Researchers can interact with STGen through various interfaces, a command-line tool (CLI), a web dashboard, and REST API endpoints following OpenAPI standards. Researchers can utilize

these interfaces for a wide range of experiments, such as simulating and mitigating DoS attacks, automating test environments, and assessing protocol performance under high-traffic loads. Although most popular testbeds struggle to run hundreds of sensor nodes on a single machine, the lightweight design of STGen enables it to handle thousands with ease. STGen offers a fast start time of 21.981 s for 6,000 nodes and uses only 38.3 GB of memory, outperforming recent testbeds like Gotham and GothX. Looking ahead, a key direction for STGen is to predict resource requirement and allow dynamic resource provisioning in cloud environments through tools that support Infrastructure-as-Code (IaC) [38] i.e., Terraform, which can enable automating cloud services.

References

[1] J. Yick, B. Mukherjee, D. Ghosal, Wireless sensor network survey, Computer Networks 52 (2008) 2292–2330.

[2] M. A. Matin, M. Islam, Overview of wireless sensor network, in: Wireless sensor networks-technology and protocols, IntechOpen, 2012.

[3] R. Ramanathan, J. Redi, A brief overview of ad hoc networks: challenges and directions, IEEE Communications Magazine 40 (2002) 20–22.

[4] S. Sudevalayam, P. Kulkarni, Energy harvesting sensor nodes: Survey and implications, IEEE Communications Surveys & Tutorials 13 (2011) 443–461.

[5] S. Saginbekov, C. Shakenov, Testing wireless sensor networks with hybrid simulators, arXiv preprint arXiv:1602.01567 (2016).

[6] C. Serôdio, J. Cunha, G. Candela, S. Rodriguez, X. R. Sousa, F. Branco, The 6G ecosystem as support for ioe and private networks: Vision, requirements, and challenges, Future Internet 15 (2023).

[7] J. Swann, L. Smith, A. Jones, Tools for network traffic generation–a quantitative comparison, Journal of Network and Systems Management 29 (2021) 1–19.

[8] X. Sáez-de Cámara, J. L. Flores, C. Arellano, A. Urbieta, U. Zurutuza, Gotham testbed: a reproducible IoT testbed for security experiments and dataset generation, IEEE Transactions on Dependable and Secure Computing 21 (2023) 186–203.

[9] M. Poisson, R. Carnier, K. Fukuda, Gothx: a generator of customizable, legitimate and malicious IoT network traffic, in: Proceedings of the 17th Cyber Security Experimentation and Test Workshop, pp. 65–73.

[10] S. Ghazanfar, F. Hussain, A. U. Rehman, U. U. Fayyaz, F. Shahzad, G. A. Shah, IoT-flock: An open-source framework for IoT traffic generation, in: 2020 International Conference on Emerging Trends in Smart Technologies (ICETST), IEEE, pp. 1–6.

[11] M. Bures, B. S. Ahmed, V. Rechtberger, M. Klima, M. Trnka, M. Jaros, X. Bellekens, D. Almog, P. Herout, Patriot: IoT automated interoperability and integration testing framework, in: 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST), IEEE, pp. 454–459.

[12] R. Li, Q. Li, Q. Zou, D. Zhao, X. Zeng, Y. Huang, Y. Jiang, F. Lyu, G. Ormazabal, A. Singh, H. Schulzrinne, IoTgemini: Modeling IoT network behaviors for synthetic traffic generation, IEEE Transactions on Mobile Computing 23 (2024) 13240–13257.

[13] S. Sundresh, W. Kim, G. Agha, Sens: A sensor, environment and network simulator, in: Proceedings of the Annual Simulation Symposium, IEEE, 2004, pp. 221–228.

[14] A. Varga, R. Hornig, An overview of the OMNeT++ simulation environment, in: Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops, pp. 1–10.

[15] G. F. Riley, T. R. Henderson, The ns-3 network simulator, in: Modeling and tools for network simulation, Springer, 2010, pp. 15–34.

[16] A. Velinov, A. Mileva, Running and testing applications for contiki os using Cooja simulator (2016).

[17] J. Postel, RFC768: User datagram protocol, 1980.

[18] N. Fotiou, D. Trossen, G. C. Polyzos, Illustrating a publish-subscribe internet architecture, Telecommunication Systems 51 (2012) 233–245.

[19] G. S. Sachdeva, Practical ELK stack, Practical ELK Stack. Apress (2017).

[20] C. Adjih, E. Baccelli, E. Fleury, G. Harter, N. Mitton, T. Noel, R. Pissard-Gibollet, F. Saint-Marcel, G. Schreiner, J. Vandaele, et al., Fit IoT-lab: A large scale open experimental IoT testbed, in: 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT), IEEE, pp. 459–464.

[21] L. Sanchez, L. Muñoz, J. A. Galache, P. Sotres, J. R. Santana, V. Gutierrez, R. Ramdhany, A. Gluhak, S. Krco, E. Theodoridis, et al., Smartsantander: IoT experimentation over a smart city testbed, Computer Networks 61 (2014) 217–238.

[22] T. T. T. Ngo, D. Sarramia, M.-A. Kang, F. Pinet, A new approach based on ELK stack for the analysis and visualisation of Georeferenced sensor data, SN Computer Science 4 (2023) 241.

[23] G. E. Uhlenbeck, L. S. Ornstein, On the theory of the brownian motion, Physical Review 36 (1930) 823–841.

[24] ElecFreaks, Hc-sr501 pir motion sensor datasheet, `https://www.elecfreaks.com`, 2011.

[25] Shaikh, F. K. and others, Energy harvesting in wireless sensor networks: A comprehensive review, Renewable and Sustainable Energy Reviews 55 (2016) 1041–1054.

[26] InvenSense, Mpu-6000 and mpu-6050 product specification, 2013. Revision 3.4.

[27] C. Bettstetter, Smooth mobility models for wireless networks, in: Proceedings of IEEE VTC, pp. 1728–1732.

[28] R. E. Deakin, M. A. Hunter, Geometric geodesy part a, School of Mathematical and Geospatial Sciences, RMIT University (2011).

[29] de Sousa, V. A. and others, Traffic modeling for M2M communications, IEEE Communications Surveys and Tutorials 19 (2017) 876–904.

[30] M. Noaman, M. S. Khan, M. F. Abrar, S. Ali, A. Alvi, M. A. Saleem, Challenges in integration of heterogeneous internet of things, Scientific Programming 2022 (2022) 1–14.

[31] S. K. Sowe, T. Kimata, M. Dong, K. Zettsu, Managing heterogeneous sensor data on a big data platform: IoT services for data-intensive science, in: 2014 IEEE 38th international computer software and applications conference workshops, IEEE, pp. 295–300.

[32] J. Moon, S. Kum, S. Lee, A heterogeneous IoT data analysis framework with collaboration of edge-cloud computing: Focusing on indoor pm10 and pm2. 5 status prediction, Sensors 19 (2019) 3038.

[33] Z. Shelby, M. Koster, C. Bormann, P. van der Stok, RFC9176: Constrained restful environments (core) resource directory, 2022. IETF.

[34] L. Eggert, G. Fairhurst, G. Shepherd, RFC8085: UDP usage guidelines, 2017. IETF.

[35] Elastic, Create visualizations in kibana, n.d. Accessed: 2025-03-20.

[36] S. Quincozes, T. Emilio, J. Kazienko, Mqtt protocol: fundamentals, tools and future directions, IEEE Latin America Transactions 17 (2019) 1439–1448.

[37] C. Bormann, A. P. Castellani, Z. Shelby, Coap: An application protocol for billions of tiny internet nodes, IEEE Internet Computing 16 (2012) 62–67.

[38] A. Rahman, R. Mahdavi-Hezaveh, L. Williams, A systematic mapping study of infrastructure as code research, Information and Software Technology 108 (2019) 65–77.