CityEar - Presentation Speaking Script

Slide 1: Title Slide

[Duration: 10 seconds]

Good morning everyone. My name is Tran Anh Dung, student ID 20226031. Today I'll be presenting CityEar, an IoT-based Urban Noise Monitoring System that I developed as my IoT System Engineer project.

Slide 2: The Problem

[Duration: 30 seconds]

Let me start with the problem. Urban noise pollution is a serious public health issue in Hanoi. Traditional monitoring approaches rely on expensive, sparse static stations that can't provide the comprehensive coverage we need.

To effectively monitor a city, we would need over 1000 sensors operating 24/7. Deploying this many physical sensors would be extremely costly and time-consuming.

So our approach was to use a Digital Twin simulation - a virtual replica of the entire sensor network - to validate our system architecture before any physical deployment. This allows us to stress-test the backend and prove the system can scale.

Slide 3: Network Architecture - Star Topology

[Duration: 45 seconds]

Our system follows a three-layer IoT architecture.

At the Perception Layer, we have 1000 simulated IoT nodes built with Node.js and NestJS. Each node mimics a physical noise sensor.

The Network Layer uses EMQX, which is an MQTT broker that handles all the message routing between devices and the backend.

Finally, the Application Layer consists of our stream processing services and the web dashboard for visualization.

We chose a Star Topology where each sensor communicates directly with the central broker. This design gives us deterministic latency, which is critical when we need to detect and respond to emergency events like gunshots within 150 milliseconds.

Slide 4: Software Architecture - Microservices

[Duration: 50 seconds]

The system is built using a microservices architecture with four core services.

First, the Simulator Service manages the virtual device lifecycle - creating, starting, stopping devices - and generates realistic noise data.

Second, the Consumer Service subscribes to MQTT topics, ingests the raw sensor data, and uses batch processing to efficiently write to the database.

Third, the API Service provides REST endpoints for historical data queries and operates a WebSocket gateway for real-time updates.

And fourth, we have the Frontend built with Next.js, which displays a real-time map and an analytics dashboard.

This diagram shows how data flows through the system - from the simulator, through MQTT, into our backend services, and finally to the frontend.

Slide 5: Why MQTT? (Protocol Analysis)

[Duration: 40 seconds]

A key design decision was choosing MQTT over HTTP for device communication.

MQTT is extremely lightweight - headers are just 2 bytes compared to HTTP's much larger overhead. This is crucial for power-constrained IoT devices.

It uses a Pub/Sub model, which means devices can publish data and go back to sleep immediately - they don't wait for responses like HTTP requires.

We also leverage MQTT's Quality of Service levels. For routine telemetry, we use QoS 0, which is "at most once" delivery. Some packet loss is acceptable for continuous noise readings. But for critical alerts like gunshots, we use QoS 1, which guarantees "at least once" delivery. Safety data must arrive.

Slide 6: The Simulator - Digital Twin Logic

[Duration: 50 seconds]

The simulator implements sophisticated time-based noise patterns to realistically model urban acoustics.

During rush hours - 7 to 9 AM and 5 to 7 PM - the base noise level is set to 80 decibels to simulate heavy traffic. During night hours from 11 PM to 5 AM, it drops to 40 decibels for quiet ambient noise. Normal hours average around 60 decibels.

To test our critical alert pipeline, the simulator probabilistically injects acoustic anomalies. There's a 0.5% chance of generating a gunshot event at 120 to 130 decibels, and another 0.5% chance for a scream at 95 to 105 decibels.

These screenshots show the device management interface where you can create virtual devices, start and stop them, and monitor their status in real-time.

Slide 7: Data Pipeline - Fast Path vs Batch Path

[Duration: 45 seconds]

We implemented two different data paths optimized for different types of events.

For normal events, we use the Batch Path. Data is buffered for 1 second, then written in bulk to TimescaleDB. This dramatically reduces database I/O pressure.

But for critical alerts - gunshots and screams - we use the Fast Path. These events immediately bypass the buffer, get republished to an internal MQTT topic, and trigger WebSocket broadcasts to the frontend.

The result is end-to-end latency of less than 150 milliseconds from the moment a gunshot is detected until it appears on the dashboard. This flow shows the complete path: Simulator, MQTT, Consumer, Database, API, WebSocket, and finally the Frontend.

Slide 8: Visualization & Analytics

[Duration: 35 seconds]

The frontend provides two main views.

The real-time map uses color-coded markers - green for normal noise, yellow for moderate, orange for high, and red for critical events. When a gunshot is detected, the marker pulses red to immediately draw attention.

The analytics dashboard provides deeper insights. It includes a pie chart showing event distribution, time-series charts for noise trends over time, and a bar chart highlighting the top 5 noise hotspots in the city.

These are actual screenshots from the running system showing real simulated data.

Slide 9: Technical Results

[Duration: 35 seconds]

Our performance testing validated that the architecture can handle real-world scale.

We're successfully simulating 1000 concurrent devices with a sustained throughput of over 200 messages per second. Critical alerts maintain sub-150 millisecond latency even under full load.

For scalability, we use TimescaleDB with hypertables for time-series data, which allows us to ingest millions of rows per day without query performance degradation. We also use PostGIS extensions for efficient geospatial queries on the map.

Slide 10: Live Demonstration

[Duration: 60-90 seconds]

Now I'd like to show you a quick video demonstration of the system in action.

[Play video or describe key moments]

In this demo, you'll see the real-time map updating as devices send data. Notice the color changes as noise levels fluctuate. Here I'm creating a new virtual device through the admin interface. And now in the analytics dashboard, you can see the interactive filtering and sorting capabilities.

This red pulsing marker shows a gunshot event that was just detected, and you can see the timestamp - it appeared on the dashboard in under 150 milliseconds.

[Or if doing live demo]

If time permits and the environment allows, I can run a live demonstration on my laptop.

Slide 11: Future Work

[Duration: 25 seconds]

Looking ahead, there are three main directions for future development.

First, transitioning from simulation to actual hardware deployment with physical ESP32 devices equipped with microphones.

Second, implementing TinyML for edge AI - this would allow on-device sound classification to distinguish between different types of sounds like jackhammers, cars, or actual gunshots before transmitting.

And third, investigating LoRaWAN protocols for low-power wide-area coverage in areas without reliable WiFi infrastructure.

Slide 12: Q&A

[Duration: Remainder]

Thank you for your attention. I'm happy to answer any questions you may have about the architecture, implementation, or results.

The complete source code and documentation are available on GitHub at the link shown here.

Backup Q&A Responses

Q: Why simulate instead of using real hardware? A: Simulation allowed us to validate the architecture at scale - 1000 devices - without the cost of physical deployment. Once the system is proven, the simulator logic can be directly ported to ESP32 microcontrollers.

Q: How does the system handle sensor failures? A: Each device publishes status heartbeats. If a device stops reporting, it's marked inactive on the map. The decoupled MQTT architecture means one failing device doesn't affect others.

Q: What's the data retention policy? A: TimescaleDB's hypertables support automatic data retention policies. We could configure it to keep raw data for 30 days and aggregated hourly data for years.

Q: Can the system scale beyond 1000 devices? A: Yes, the MQTT broker and microservices architecture is horizontally scalable. We could add more Consumer and API instances behind a load balancer to handle additional devices.

Q: How much does it cost to run? A: The entire stack runs in Docker containers. For 1000 devices, a mid-range cloud instance would cost approximately $50-100/month depending on the provider.