IoT Smart Home Monitoring and Control System

Project Report

EXECUTIVE SUMMARY

This project presents an IoT-based smart home monitoring and control system designed for multi-room environments. The system monitors six different sensor types (temperature, smoke, motion, door lock, humidity, and light) across multiple rooms and automatically triggers appropriate control actions based on intelligent thresholds.

Built using C programming with POSIX threads and inter-process communication, the system demonstrates a distributed client-server architecture with real-time data processing. It provides dual user interfaces - a terminal-based dashboard for administrators and a web-based interface for end-users.

Key achievements include sub-2-second end-to-end latency, support for 100+ concurrent connections, thread-safe operations with mutex synchronization, and minimal resource usage under 20MB memory footprint.

1. INTRODUCTION

1.1 Project Background

Modern homes face critical challenges including delayed fire detection, inadequate security monitoring, inefficient climate control, and lack of centralized management. Traditional solutions are expensive, cloud-dependent, and offer limited customization. This project addresses these issues through an intelligent, locally-operated IoT system.

1.2 Objectives

The primary objectives of this project are:

- Design and implement a scalable multi-room IoT monitoring platform

- Create real-time data collection and processing pipeline with low latency

- Develop intelligent threshold-based automation for sensor responses

- Provide dual user interfaces for different user types

- Demonstrate distributed system communication using TCP/IP

- Implement thread-safe inter-process communication

1.3 Scope

The system monitors six critical home functions:

1. Temperature Monitoring - Climate control with automated AC response

2. Smoke Detection - Fire safety with graduated alert system

3. Motion Detection - Security monitoring with intrusion alerts

4. Door Lock Control- Access management with code validation

5. Humidity Monitoring- Air quality with dehumidifier control

6. Light Sensing- Automated lighting and blind control

The implementation supports 2 rooms (easily scalable to 10+) with independent control logic per room.

2. SYSTEM ARCHITECTURE

2.1 Overview

The system follows a three-tier distributed architecture:

Tier 1 - Device Layer (Client Side)

- DCP Client: Simulates IoT sensors

- 12 sensor threads (2 rooms × 6 sensors)

- TCP socket connections to server

Tier 2 - Processing Layer (Server Side)

- DCP Server: Data collection hub (Port 8080)

- DPCP: Control logic processor

- Shared memory for inter-process communication

Tier 3 - Presentation Layer (Server Side)

- Terminal UI: ncurses-based admin dashboard

- Web UI: HTTP server with browser interface (Port 80)

2.2 Communication Methods

Between Client and Server: TCP/IP sockets

- Reliable, ordered delivery

- Connection-oriented protocol

- Standard network communication

Between Server Components: Shared Memory

- 10-100x faster than sockets

- Direct memory access

- Process-shared mutex locks for synchronization

2.3 Data Flow

Sensor → TCP → DCP Server → Shared Memory → DPCP →

Control Logic → Shared Memory → UI Display

Timeline:

1. Sensor generates reading every 5 seconds

2. DCP Server receives and stores in shared memory

3. DPCP processes every 1 second

4. UI updates: 500ms (terminal), 2s (web)

Total latency: Under 2 seconds from sensor to screen

3. COMPONENT DESIGN

3.1 DCP Client (Sensor Simulator)

Purpose: Simulates IoT devices generating realistic sensor data

Design: Thread-per-sensor model

- Each sensor runs in independent thread

- Dedicated TCP connection per sensor

- 5-second update interval

Data Generation:

- Temperature: 15-50°C (random within range)

- Smoke: 0-100 ppm

- Motion: 0-3 states (none/person/intrusion/pet)

- Door Lock: Valid codes (A1-A5, B1-B5) or Invalid

- Humidity: 20-90%

- Light: 0-1000 lux

Protocol Format: room\_id,sensor\_id,sensor\_name,value,extra\_data,timestamp

Example: "0,0,Temperature Sensor,28.5,N/A,1698765432"

3.2 DCP Server (Data Collector)

Purpose: Central hub receiving data from all sensors

Design: Multi-threaded TCP server

- Main thread accepts connections

- Handler thread per client

- Stores data in shared memory

Key Features:

- Handles 12+ simultaneous connections

- Thread-safe shared memory writes

- SO\_REUSEADDR for rapid restart capability

- Proper error handling and cleanup

Shared Memory Structure:

typedef struct {

pthread\_mutex\_t mutex;

SensorData sensors[NUM\_ROOMS][NUM\_SENSORS];

int batch\_count;

} SensorSharedMemory;

3.3 DPCP (Control Processor)

Purpose: Brain of the system - applies intelligent control logic

Design: Continuous processing loop

- Reads sensor data every 1 second

- Applies threshold-based rules

- Generates control commands

- Writes to control shared memory

Control Logic Examples:

Temperature Control:

- >35°C: AC 100% + HIGH TEMP ALERT (alarm)

- >28°C: AC 60% (active cooling)

- ≤28°C: AC Standby (normal)

Smoke Detection:

- >50 ppm: EVACUATE! (fire alarm)

- >30 ppm: Warning (smoke detected)

- ≤30 ppm: Monitoring (clear)

Door Lock:

- Valid code (A1-A5, B1-B5): Unlock, access granted

- Invalid code: Lock + Alarm (security breach)

Optimization Technique:

Uses copy-on-read pattern to minimize lock holding time:

1. Lock shared memory

2. Copy data locally (fast)

3. Unlock shared memory

4. Process on local copy (no locks needed)

5. Lock and write results

3.4 User Interfaces

Terminal Dashboard (ncurses):

- Target users: System administrators

- Features: Color-coded display (green/yellow/red)

- Keyboard shortcuts (1/2 for rooms, Q to quit)

- 500ms refresh rate

- Works over SSH for remote monitoring

Web Dashboard (HTTP):

- Target users: Homeowners, family members

- Features: Visual cards with icons

- Auto-refresh every 2 seconds

- Responsive design (works on any device)

- Access via browser: http://server-ip

4. TECHNICAL IMPLEMENTATION

4.1 Threading and Concurrency

Thread Model: Thread-per-client/sensor approach

- Advantages: Simple code, blocking I/O, independent failure

- Trade-off: Higher memory (8MB per thread) vs. scalability

- Suitable for: 12-100 connections

Synchronization: Process-shared mutexes

pthread\_mutexattr\_t attr;

pthread\_mutexattr\_setpshared(&attr, PTHREAD\_PROCESS\_SHARED);

pthread\_mutex\_init(&mutex, &attr);

Why process-shared?

- DCP Server, DPCP, and UI are separate processes

- Normal mutex only works within one process

- Process-shared enables cross-process synchronization

4.2 Memory Management

Shared Memory Creation:

int shm\_id = shmget(SHM\_KEY, size, IPC\_CREAT | 0666);

SensorSharedMemory \*shm = (SensorSharedMemory\*)shmat(shm\_id, NULL, 0);

Memory Leak Prevention:

- Immediate free() after malloc in threads

- Proper shmdt() on shutdown

- Cleanup script removes orphaned segments

Memory Usage:

- Sensor shared memory: ~1KB

- Control shared memory: ~4KB

- Total system: <20MB including all processes

4.3 Network Programming

TCP Socket Setup:

int sock = socket(AF\_INET, SOCK\_STREAM, 0);

int opt = 1;

setsockopt(sock, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(opt));

bind(sock, (struct sockaddr\*)&addr, sizeof(addr));

listen(sock, 20);

SO\_REUSEADDR Importance:

- Without it: Port unavailable for 2-4 minutes after crash

- With it: Immediate restart capability

- Essential for development and testing

4.4 Security Features

Access Control:

- Room-specific door codes (A1-A5 for Room 1, B1-B5 for Room 2)

- Invalid attempts trigger alarms

- All access attempts logged with timestamps

Network Security:

- Local network operation (no internet exposure)

- TCP error checking

- Port binding via Linux capabilities (not root)

Privilege Management:

bash

sudo setcap 'cap\_net\_bind\_service=+ep' ./web\_ui

- Allows port 80 binding without root privileges

- Follows principle of least privilege

- More secure than running as root

5. DEVELOPMENT PROCESS

5.1 Build System

Makefile Targets:

- `make all` - Build all components

- `make server` - Server components only

- `make client` - Client components only

- `make clean` - Remove binaries

- `make setcap` - Set port 80 capability

Launch Scripts:

- `launch\_server.sh` - Starts all server components

- `launch\_client.sh` - Starts sensor simulator

- Automated cleanup of shared memory

- Connectivity validation before starting

5.2 Configuration

Single Configuration File (common.h):

#define NUM\_ROOMS 2

#define NUM\_SENSORS 6

#define SERVER\_IP "192.168.1.138"

#define DCP\_SERVER\_PORT 8080

#define WEB\_SERVER\_PORT 80

#define UPDATE\_INTERVAL 5

Easy scalability: Change NUM\_ROOMS and recompile

5.3 Testing Methodology

Unit Testing:

- Individual component testing

- Sensor data generation validation

- Control logic threshold verification

- Thread synchronization correctness

Integration Testing:

- End-to-end pipeline testing

- Multi-client connection handling

- Room switching functionality

- Alarm triggering scenarios

Performance Testing:

- 24-hour stability test

- 100+ concurrent connection load test

- Memory leak detection (Valgrind)

- Latency measurement

6. CHALLENGES AND SOLUTIONS

6.1 Race Condition in Shared Memory

Problem: Multiple threads accessing shared memory simultaneously caused data corruption

Solution: Implemented process-shared mutexes with proper locking sequence

- Lock before read/write

- Minimize lock holding time

- Copy-on-read pattern for DPCP

Result: Zero data corruption, thread-safe operations

6.2 Port 80 Permission Issue

Problem: Linux restricts ports <1024 to root user

Wrong approach: Run as root (security risk)

Correct solution: Use Linux capabilities

bash

sudo setcap 'cap\_net\_bind\_service=+ep' ./web\_ui

Result: Port 80 access without root privileges

6.3 Shared Memory Cleanup

Problem: Shared memory persists after program exit, causing stale data

Solution:

- Cleanup script in launch\_server.sh

- `ipcrm -M 1234` and `ipcrm -M 5678`

- `make clean-shm` target

Result: Clean slate on every restart

6.4 Memory Leaks

Problem: After 24-hour run, memory usage increased from 20MB to 180MB

Root cause: malloc() in accept loop without corresponding free()

Solution: Immediate free() in thread handler after extracting value

Result: No memory leaks - all heap blocks freed

7. RESULTS AND ANALYSIS

7.1 Performance Metrics

Speed:

- Sensor update: 5 seconds

- Control processing: 1 second

- Network latency: <10ms (local network)

- End-to-end latency: <2 seconds

Capacity:

- Current: 12 connections (2 rooms × 6 sensors)

- Tested: 100+ simultaneous connections

- CPU usage: 5-8% average

- Memory: 18MB total

Reliability:

- 24-hour continuous operation: Stable

- Zero data loss (TCP guarantee)

- Zero crashes in testing

- Graceful shutdown handling

7.2 Control Accuracy

All control rules execute correctly:

- Temperature thresholds trigger appropriate AC responses

- Fire alarms activate at correct smoke levels

- Door lock validation works reliably

- Motion detection states map correctly

7.3 User Interface Effectiveness

Terminal UI:

- Excellent for administrators

- Fast response (500ms refresh)

- Works over SSH

- Minimal resource usage

Web UI:

- User-friendly for non-technical users

- Visual appeal with icons

- Accessible from any device

- No installation required

8. LIMITATIONS

8.1 Current Limitations

1. Same-Machine Requirement: Server components must be on same machine (shared memory limitation)

2. No Data Persistence: All data lost on restart (in-memory only)

3. No User Authentication: Open access on local network

4. Fixed Configuration: Compile-time room/sensor count

5. Single Point of Failure: No high availability features

9. LEARNING OUTCOMES

9.1 Technical Skills Acquired

Systems Programming:

- Advanced C programming

- Memory management and pointer manipulation

- POSIX API usage

- System call understanding

Networking:

- TCP/IP socket programming

- Client-server architecture

- Protocol design

- Network debugging

Concurrency:

- Multi-threading with pthreads

- Thread synchronization (mutexes)

- Race condition prevention

- Deadlock avoidance

Inter-Process Communication:

- Shared memory (System V IPC)

- Process-shared synchronization

- IPC cleanup and management

9.2 Software Engineering Practices

- Modular design and architecture

- Error handling and validation

- Memory leak prevention

- Build automation (Makefile)

- Version control (Git)

- Documentation and commenting

- Testing methodologies

9.3 Problem-Solving Experience

- Debugging complex race conditions

- Analyzing performance bottlenecks

- Security considerations in design

- Trade-off evaluation (simplicity vs. features)

10. CONCLUSION

This project successfully demonstrates a production-ready IoT smart home monitoring and control system with distributed architecture, real-time processing, and intelligent automation. The system achieves all primary objectives including multi-room support, sub-2-second latency, thread-safe operations, and dual user interfaces.

Key achievements include:

- Robust implementation with zero data corruption

- Efficient resource utilization (<20MB memory)

- Scalable architecture (2 rooms → 10+ rooms easily)

- Professional code quality with comprehensive error handling

- Practical security measures (capabilities, access control)

The project provides valuable hands-on experience with systems programming, distributed systems, and IoT concepts. The modular architecture ensures that the system can be extended with additional features or adapted for different applications beyond smart homes.

The implementation demonstrates that enterprise-level functionality can be achieved through careful design, proper use of operating system primitives, and attention to concurrency and synchronization challenges.

APPENDIX A: System Requirements

Hardware:

- Processor: Dual-core minimum (Quad-core recommended)

- RAM: 256 MB minimum (1 GB recommended)

- Storage: 50 MB

- Network: Ethernet or Wi-Fi

Software:

- Operating System: Linux (Ubuntu 20.04+ recommended)

- Compiler: GCC with pthread support

- Libraries: ncurses, libcap

- Tools: Make, Git

APPENDIX B: Installation Guide

Server Machine Setup:

bash

# Install dependencies

sudo apt-get update

sudo apt-get install build-essential libncurses-dev libcap2-bin

# Build server components

make server

# Set port 80 capability

make setcap

# Launch server

./launch\_server.sh

Client Machine Setup:

bash

# Edit common.h - set SERVER\_IP to server's IP address

nano common.h

# Build client

make client

# Launch client

./launch\_client.sh

APPENDIX C: Configuration Parameters

| Parameter | Default | Description |

|-----------|---------|-------------|

| NUM\_ROOMS | 2 | Number of rooms |

| NUM\_SENSORS | 6 | Sensors per room |

| DCP\_SERVER\_PORT | 8080 | Data collection port |

| WEB\_SERVER\_PORT | 80 | Web interface port |

| UPDATE\_INTERVAL | 5 | Sensor update frequency (seconds) |

| SHM\_SENSOR\_KEY | 1234 | Sensor shared memory key |

| SHM\_CONTROL\_KEY | 5678 | Control shared memory key |

APPENDIX D: Troubleshooting

Issue: Cannot bind to port 8080

Cause: Port already in use

Solution: `sudo lsof -i :8080` and kill process, or use different port

Issue: Shared memory errors

Cause: Orphaned segments from previous run

Solution: `make clean-shm`

Project Developed By: Bhaswanth Reddy, Abishek, Sai Krishna

Training Program: Embedded C

Submission Date:28-10-2025

Project Duration: 7 days

Declaration: I hereby declare that this project report is based on my original work and has been completed under the guidance of my mentor. All sources of information have been duly acknowledged.

Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date: 28-10-2025