SenseRT: A Streaming Architecture for Smart Building Sensors

Rohit Verma
rv355@cam.ac.uk
University of Cambridge
Cambridge, United Kingdom

Justas Brazauskas
jb2328@cam.ac.uk
University of Cambridge
Cambridge, United Kingdom

Vadim Safronov
vs451@cam.ac.uk
University of Cambridge
Cambridge, United Kingdom

Matthew Danish
mrd45@cam.ac.uk
University of Cambridge
Cambridge, United Kingdom

Jorge Merino
jm2210@cam.ac.uk
University of Cambridge
Cambridge, United Kingdom

Xiang Xie
xx809@cam.ac.uk
University of Cambridge
Cambridge, United Kingdom

Ian Lewis
ijl20@cam.ac.uk
University of Cambridge
Cambridge, United Kingdom

Richard Mortier
rmm1002@cam.ac.uk
University of Cambridge
Cambridge, United Kingdom

ABSTRACT

Building Management Systems (BMSs) have evolved in recent years, in ways that require changes to existing network architectures that follow the store-then-analyse approach. The primary cause is the increasing deployment of a diverse range of cost-effective sensors and actuators in smart buildings that generate real-time streaming data. Any in-building system with a large number of sensors needs a framework for real-time data collection and concurrent stream processing from sensors connected using a range of networks.

We present SenseRT, a system for managing and analysing in-building real-time streams of sensor data. SenseRT collects streams of real-time data from sensors connected using a range of network protocols. It supports concurrent modules simultaneously performing stream processing over real-time data, asynchronously and non-blocking, with results made available with minimal latency. We describe a prototype implementation deployed in two University department buildings, demonstrating its effectiveness.

CCS CONCEPTS

• Networks → Sensor networks; • Hardware → Sensors and actuators; • Computer systems organization → Real-time system architecture.

KEYWORDS

real-time data, stream processing, smart buildings, sensors, spatio-temporal data

1 INTRODUCTION

Increased penetration of the Internet of Things (IoT) in our lives [22], is making it increasingly easy to deploy energy-efficient sensors and actuators to perform tasks like occupancy detection [26] or indoor-environment control [17]. The number of deployed IoT devices by 2021 just for smart buildings would be around 10.8 billion (2.8 billion for residential buildings) [4]. To put that in perspective, an average home in 2020 would generate approximately 4.7 terabytes of data annually [4, 8]. Handling such amounts of data is a challenging problem, exacerbated by most such sensors generating real-time streams of data.

The data generated has both spatial and temporal aspects. For example, the humidity level falling below a threshold causing discomfort to occupants will be observed by a sensor deployed in a specific place in the building (spatial aspect) at a particular date and time (temporal aspect). The spatial aspect is relatively straightforward to handle, but there are several subtleties to the temporal aspect: what time should be assigned as the time of reading, and associated with a particular sensor reading? When the data was sensed, when the sensor transmitted the reading, or when it arrived at the receiving platform?

Figure 1: Timeliness vs latency.

This brings to the fore concepts of latency and timeliness, depicted in Figure 1, often incorrectly considered synonymous. We use latency to refer to the delay introduced between the time of recording the reading at the sensor and that reading arriving at the intended destination. In contrast, timeliness refers to the inherent characteristic of an event, the timescale on which it is appropriate to determine a reading changed. In the humidity example above, latency would be the delay between the time the sensor recorded the humidity and when the system determining the degree of occupant comfort received the reading; this will vary between sensors and over time. However, timeliness is the time period over which readings from the sensor lead to a situation where the humidity...
Building a system to handle devices in a building and the large volume of real-time data streams they generate while keeping these key spatio-temporal concepts in mind has multiple challenges. First, the heterogeneous nature of sensors and consequently the data they generate. The deployed sensors in a building could connect using channels including Bluetooth, Wi-Fi, ZigBee, LoRaWAN, or the in-building Modbus. Moreover, the definition of time of reading for sensors using these different networks could be different. The system should accept data from any such channels and normalise aspects such as time of reading. Second, analysis of the stream of data generated from all the in-building sensors in real-time should introduce minimal latency between data generation, data analysis, and publication of a result, especially for crucial events such as fire or power outage in critical areas. Moreover, processing of this stream of real-time data should be done for all sensors concurrently. Third, all data flow must be asynchronous and non-blocking while supporting concurrency.

To tackle these challenges, we develop SenseRT, a system for real-time data flow in smart buildings. Key contributions include:

- SenseRT defines a set of design principles ensuring that the architecture is easy to mutate based on the needs of the building at any time of building design.
- SenseRT provides custom decoders to retrieve data from sensors on different channels, accumulated via bridging framework of Message Queuing Telemetry Transport (MQTT) [59], and normalise it for easy use throughout the system.
- SenseRT provides a suite of modules which follow the stream processing [60] framework to ensure that the streams of real-time data from a large number of sensors is managed and analysed with minimal latency and adhere to timeliness bounds.
- All data handling and stream processing modules are designed to follow the actor model [34], which guarantees asynchronous and concurrent processing. We modify the actor model such that the concurrent modules could work in a non-blocking fashion.
- SenseRT provides support for client-side applications to analyse and process the data as they need.

We continue by discussing related work in smart buildings (§2), before describing the design principles and high-level system architecture (§3). We implemented a prototype of SenseRT in two buildings in our university and ran experiments to observe the architecture’s effectiveness for seven months. We describe this implementation (§4) and a case study of an end-to-end application (§5), before evaluating the implemented prototype (§6). We conclude with a discussion of limitations and future work (§8).

### 2 RELATED WORK

The IoT market is expected to increase to 5.8 billion endpoints in 2020, a 21% increase from 2019, with utilities (electricity, water) the most significant users (1.17 billion) and building automation showing the highest growth of 42% [2]. A boost towards this direction could be because of the growth in the number of smart devices and smart objects that build the ecosystem for a smart building [24]. This ecosystem of smart devices and smart objects is enabled by the network technology use, with different protocols (e.g., LoRaWAN, ZigBee, Wi-Fi, Bluetooth) being selected based on the desired goals.

#### 2.1 Communication in Smart Buildings

Communication protocols enable the exchange of a massive stream of data between sensors and the network. Factors like range, data load, power demand, and security define which communication protocol would be suitable for a particular set of smart devices. A comparison of the major protocols is given in Table 1.

| Protocol | Frequency | Data Rate | Range | Power | Security |
|----------|-----------|-----------|-------|-------|----------|
| Bluetooth [52] | 2.4 GHz | 1 Mb/s | 50 m | Low | 64 and 128 bit encryption |
| Wi-Fi [13] | 2.5 GHz, 5 GHz | 150-200 Mb/s (typical) | 50–100 m | High | 256 bit key encryption |
| ZigBee [14] | 2.4 GHz | 250 kb/s | 10–100 m | Low | 128 AES layer security |
| LoRa [16] | Region specific | 0.3–50 kb/s | 2–5 km | Low | 128 bit AES encryption key |
| Modbus [6] | 2.4 GHz | 9600 b/s | Layout dependent | High | 128 bit AES encryption (if Modbus/TLS) |

Table 1: Popular sensor communication protocols used in smart buildings
a high level of indoor environment quality (IEC) [11] by using different types of sensor. We divide them into two broad categories:

**Dumb Sensors.** This type includes sensors such as pressure mats, IR sensors, CO₂ sensors, temperature sensors, particulate matter sensors. They simply read and transmit readings for values they sense, and can be used to understand occupant behaviour patterns [30, 68] or to optimise IEC parameters to improve thermal comfort, visual comfort, or indoor air quality [27, 47].

**Intelligent Sensors.** This type compromise one or more dumb sensors with attached computation that can process the raw data and relay analysed results. Example include smartphones, fingerprint sensors, wearable sensors, smart cameras, and body thermometers. In smart buildings these sensors can be used for traditional purposes such as occupancy detection and managing IEC [66], but can also provide more personalised functions [56].

### 2.3 Smart Building Management and Control

Several works provide a framework for building control to balance energy consumption and maintain occupant comfort. iDorm [32] set up a testbed where multiple embedded sensors were fitted in a dorm room to obtain responsive inputs from the user, which helped to learn user preferences using distributed AI and fuzzy-genetic logic. MASBO [41] is a multi-agent system where data arriving from a Building Management System (BMS) is observed and analysed by a set of agents to provide suitable energy-efficient control for the building without compromising on occupant comfort. Chen et al. [18] propose a hierarchical system architecture that emphasises improving savings over the building life-cycle while addressing stakeholder goals.

Another class of work concentrates more on how sensor data could be collected and managed in a smart building. Choubey et al. [19] set up a localised sensor network in an area and perform localised data processing for this set of sensors. LabVIEW [54] provides a data collection framework to collect humidity, temperature, and light data from sensors in a wireless sensor network in the building. Bashir et al. [15] provide an IoT Big Data Analytics (IBDA) based framework for storage and analysis of real-time data that the IoT sensors in a smart building generate.

Most architectures follow the trend of using a particular approach to collect sensor data locally and then store it on the cloud for further processing. Data collection efficiency is achieved by using different means like adding a programmed data acquisition chip [10], setting up a fog server [29], or using IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) [28]. However, all of these systems rely on stored data to perform any analysis rather than analysing it as and when it arrives.

### 2.4 Necessity of Stream Processing

The existing systems [10, 15, 25, 29, 54], utilize the prevalent store-then-analyze architecture, where data is first stored in a data repository and then analyzed as per the needs of the applications (Figure 2). However, several use-cases of sensing in buildings are linked to time-critical responses. It is very important to minimize delay in safety based use-cases like gas leak [31], water leak [44] or fire detection [42] in buildings. When considered as basic components of a smart-grid network, timely analysis of energy consumption is important to optimize smart-grid management [21, 46]. When the building is used for specialized purposes like hospitals [37] or elderly care homes [12, 61], adhering to timeliness of sensor data becomes crucial. When trying to act on such scenarios, which generate always moving time-critical data, the store-then-analyze approaches fall short. The reliance on a source of data repository would involve several to-and-fro network transactions resulting in crucial time loss. This fails a real-time system’s adherence to timeliness [60, 62].

Along the line of these time-critical solutions, ScaleOut [53] has pointed out the importance of real-time stream processing [60] for digital twins. ThoughtWire [63] who also work with stream processing based systems for digital twins share a similar notion. CityPulse [64] and Zhou et al. [69] advocate the necessity of stream processing based data analytics for smart city projects.

Figure 3 shows the stream processing approach which unlike Figure 2 performs storage and analysis simultaneously. The stream processing module has two (or more) concurrent processors which analyse and store data at the same time. The analysed results are then made available to any applications which have subscribed a priori.

### 3 DESIGN & ARCHITECTURE

We first describe the design principles SenseRT is built upon and then its high-level architecture.
3.1 Design Principles

The SenseRT architecture must be flexible to changes at any period of deployment and robust enough to handle the continuous stream of real-time data entering the system through the numerous sensors in the building via different channels. To achieve this, we apply the following design principles.

All data is spatio-temporal. Over a long enough period, the metadata about any part of the building or sensors is bound to change. For instance, a sensor could be moved to a new room changing its location, or a room be divided into two rooms changing its boundary. Tagging every datum with a timestamp and location information, enables change tracking of deployments.

Spatial hierarchy of container objects. Sensors could be deployed in different locations having different usage and access control policies depending on the granularity of location considered (e.g., building, floor, room, desk). This defines each component as an object which could hold multiple other objects, each of which could have its own set of sensors. This ensures physical changes in the building are handled with minimal changes, e.g., changing only the parent when a desk is moved to another room, or reusing the floor model on adding a new floor to the building.

Stream processing of data. SenseRT uses a publish/subscribe model for data exchange, ensuring no polling is required for the real-time stream of data arriving at any time instance to the system. This reduces delivery latency, essential for real-time data analysis.

Asynchronous message transfer in a non-blocking framework. Asynchronous message transfer ensures that any message from any sensor or system module can be analysed in real-time, and multiple modules can work on the same data concurrently. This in turn guarantees a non-blocking data processing framework.

3.2 System Architecture

The high-level decomposition of SenseRT results in five layers, shown in Figure 4: Physical, Network, Integration, Data, and Application.

**Physical Layer.** Holds all hardware devices like sensors, actuators, and building meters. Devices report data and exchange messages using one or more of their supported protocols and formats. Our deployment uses a wide range of sensors outlined in Table 2.

**Network Layer.** Houses the devices required to support message transfer from the sensors such as Wi-Fi access points, ZigBee or LoRaWAN gateways, or Modbus components. Our deployment uses Zigbee and Wi-Fi networks for short-range connectivity within particular areas, and a LoRaWAN network to provide backhaul interconnection within and between buildings.

**Integration Layer.** Provides services to collate and homogenise data received over different network types from different hardware devices, making it much more straightforward to build applications to analyse and react to data from one or more sources. Consider a room that has smart plugs, LoRa sensors, and an electric meter. The smart plugs could be uploading data over Wi-Fi, LoRa sensors through LoRaWAN via The Things Network (TTN) [65], and the electric meters through a Modbus. Services in this layer ensure that data received over all these protocols are available through a single channel. Furthermore, this layer also ensures that data exchange in the system is through a publish/subscribe method. Our deployment achieves this by making use of the Message Queuing Telemetry Transport (MQTT) [59] and MQTT bridging.

**Data Layer.** Provides services to manage data streams arriving from the integration layer. A set of decoders normalises the data from multiple devices, which is made available to a Real-Time Server (RTS). The RTS holds various crucial modules of the architecture. These modules handle real-time stream processing, routing messages to other similar systems as required, storing data for future usage, and making data available to the external application. In our deployment the RTS is implemented to be asynchronous and non-blocking using Vert.x. The data layer also houses the database, which stores the spatio-temporal metadata of all the sensors and the object-level components in a hierarchical structure.

**Application Layer.** The client-facing layer, providing APIs and user interfaces for those wishing to access sensor data. Our deployment comprises a server that acts as the point-of-contact with SenseRT for all client-side applications trying to access sensor data. The details of application architecture are out of scope for this paper as we are primarily concerned with the overall system.

4 IMPLEMENTATION

We deployed a prototype of SenseRT over two department buildings on our University campus. The two buildings each have multiple floors containing offices, labs, communal areas, and corridors. Figure 5 depicts the overall architecture of the prototype, and we detail each layer next.

4.1 Physical Layer

Table 2 lists the sensors used in our deployment, dumb and intelligent.
Dumb Sensors. The three types of dumb sensor we deployed are depicted in Figure 6, and are:

**Smart plugs** (Figure 6a). These are COTS smart plugs from several vendors built around the ESP8266 part [1]. We replaced their default firmware with the Tasmota firmware [3] so we could control where data were sent. The smart plugs were controlled over Wi-Fi using Message Queuing Telemetry Transport (MQTT).

**LoRaWAN Sensors** (Figure 6b). We use different types of COTS LoRaWAN sensors, measuring e.g., CO\(_2\), temperature, and occupancy from vendors including Elsys and Radio Bridge. These were managed over LoRaWAN via The Things Network (TTN).

**ZigBee Sensors** (Figure 6c). We use two types of ZigBee sensor: infra-red motion sensors, and door/window open/closed sensors. These sensors were accessed via ZigBee gateways.

**Intelligent Sensors.** The DeepDish [23] (Figure 6d) intelligent sensor counts the number of people in an area. DeepDish uses TensorFlow to identify and track selected objects (e.g., cars, bicycles, people) in the video feed, and supports occupancy counting by counting how many people cross a line in the scene in each direction. Only the cumulative occupancy count is reported, over Wi-Fi; no video data is collected or transmitted.

![Figure 5: Our prototype implementation of the SenseRT architecture. Solid black lines show real-time data flow while dashed red lines show request-response data flow.](image)

![Table 2: Sensors used in our deployment.](image)

![Figure 6: Four types of sensors used in our deployment.](image)
Sensors

We used the Things Network (TTN) services, deCONZ [7], and Wi-Fi to connect to TTN.

which do not require Internet access and have a range of up to 30 m

with a low packet error rate [38].

(Figure 8) for which any ZigBee device connected to mains power

(network devices).

The second, from Ubiquiti, follows the IEEE 802.11ac standard,

The hardware not only provided long-range connectivity but also

of 1500 devices transmitting through the gateway on the same channel

low spreading factor guarantees low latency and supports around

suggested for real-time monitoring systems,

factor

spreading

These gateways forward the LoRaWAN

RaWAN messages over the LoRa radio protocol, which is received

by the deployed gateways. These gateways forward the LoRaWAN

messages over the Internet, which is made available through

TTN. These messages are sent to the TTN MQTT broker (say

ttnMQTT) as well as the ZigBee MQTT broker (say zigbeeMQTT) by configuring localMQTT appropriately. This configuration is flexible because we could decide if the bridging is required on all the topics or a subset of topics. Once set up, localMQTT acts as the sole broker for data exchange between the Physical and other layers. This ensures that the Data layer need not tackle any changes in inclusion or removal of brokers in the architecture.

4.3.1 Smart plug Message Exchange. We used the MQTT protocol for message exchange with the smart plugs. MQTT is a popular lightweight protocol for IoT projects, providing publish/subscribe support for real-time data exchange. MQTT has three primary components; (i) broker: which is the server handling data exchange, (ii) publisher: a client which sends a message, and (iii) subscriber: a client which retrieves messages. Publishing and subscribing is with reference to a unique topic, and a client can act both as a publisher or subscriber. We use the mosquitto broker [5]. Each smart plug publishes and subscribes to a topic unique to the device id. The broker receives periodic messages from all the sensors.

4.3.2 ZigBee Sensor Message Exchange. We set up our framework to obtain ZigBee sensor data through MQTT, as shown in Figure 8. We used deCONZ [7], an application which communicates with the gateway to expose the devices connected to the gateway. It provides websocket support for real-time data exchange and a set of APIs to manage sensors. We introduced the Read/Write module, which continuously listens for data from the deCONZ websocket and publishes the same on the Zigbee MQTT broker.

4.3.3 LoRa Sensor Message Exchange. On deployment, The LoRa sensors are registered with the TTN using Over The Air Activation (OTAA). Once deployed, these sensors start broadcasting the LoRaWAN messages over the LoRa radio protocol, which is received by the deployed gateways. These gateways forward the LoRaWAN messages to TTN over the Internet, which is made available through the MQTT API. The messages obtained through the MQTT API are linked to a topic to which the sensors and the server subscribe. TTN also ensures that message transfer is secured.

4.3.4 Bridging. In our implementation, we had three channels, each having its own MQTT brokers. In order to integrate messages from all channels at a single broker, we utilise the MQTT bridging technique. Effectively, we subscribe the Wi-Fi MQTT broker (say localMQTT) to the TTN MQTT broker (say ttnMQTT) as well as the ZigBee MQTT broker (say zigbeeMQTT) by configuring localMQTT appropriately. This configuration is flexible because we could decide if the bridging is required on all the topics or a subset of topics. Once set up, localMQTT acts as the sole broker for data exchange between the Physical and other layers. This ensures that the Data layer need not tackle any changes in inclusion or removal of brokers in the architecture.
4.4 Data Layer

This handles three major tasks, (i) decoding and homogenising the data obtained from different channels via the localMQTT broker, (ii) storing data for future use, with associated metadata, and (iii) making the data available in real-time for processing by client-side applications.

4.4.1 Decoders. The messages transmitted by the sensors vary based on factors like the type of sensor, vendor, design, transmission channel. For example, the smart plugs we used include their unique device number in the MQTT topic and not in the message, while most of the LoRa sensors include this information in the message itself. Moreover, although SenseRT strives that the time of a reading is generated as far upstream the architecture as possible, preferably at the sensor, it is not always achievable. Several real-world sensors do not include time in the message as they do not have a real-time clock. Simply receiving and storing the message without assigning any time to it would make any further processing a complex task, especially for a real-time system. The decoders take care of these problems and generate a normalised message for each sensor. We implemented a set of decoders for the different classes of messages received from the localMQTT broker. The decoder program consists of two main components, (i) the decoder set, which includes all the decoders for the available sensors, and (ii) the decoder manager, which based on the message decides which decoder to use as well as automatically registering new decoders added to the decoder set. For our prototype, we simply append the timestamp and unique device id to all messages. However, more complex processing could be done to generate a more advanced payload.

4.4.2 Storage. SenseRT has two types of storage: a data store containing all the data received from the physical devices in separate JSON files, and a metadata store containing metadata for all devices except network devices.

We used PostgreSQL to implement the metadata store. As different devices have different metadata available, we use a simple two column table, one with the sensor’s unique id, and the other a jsonb column storing all other information in JSON format. The JSON entry for each device contains at least a timestamp property, guaranteeing that we have all the historical changes of a sensor in the database instead of only having the latest information; and a location property, indicating where in the building the sensor is deployed. The location information also included the coordinates in the XYZ plane, with one corner of the building as the origin to calculate the XY coordinates, and the Z coordinate combines the floor the sensor is on and the height relative to that floor (Figure 9).

4.4.3 Real-Time Server. The Real-Time Server (RTS) supports minimal latency processing of real-time data and asynchronous & non-blocking data management. This is guaranteed by following the Actor Model [34] using Vert.x [20] to receive and support analysis of data in real-time by multiple Vert.x modules called vertices. The actor model supports concurrency by enforcing that each actor (here implemented as a verticle) only interacts with other actors (verticles) through messages posted to its message-box. This model provides the following advantages:

Asynchronous message passing. The actor model guarantees that the RTS adheres to an asynchronous message-passing paradigm, providing real-time data handling from whichever source it is received. The FeedHandler receives a message arriving at the RTS and publishes on the EventBus to be used by other verticles.

Non-blocking modules. Using the Vert.x library with the actor model provides support to build a non-blocking framework for the RTS. Unlike the standard actor model, where each actor has its own message-box, in our implementation, the EventBus acts as the common message-box for all verticles. The message exchange is performed using a publish/subscribe approach. Any verticle accesses data by subscribing to the EventBus and sends messages by publishing it to the EventBus. The model also ensures that any communication between two verticles also happens only through the EventBus.

Modular server. Each verticle is an independent actor ensuring that the RTS is modular. This guarantees that any number of modules could be added or removed as needed without affecting any existing verticles. As a result, a standard production implementation could have thousands of verticles accessing data concurrently.

As well as following the actor model for concurrency, all verticles in SenseRT act as stream processors. Unlike most systems, which store the data in a storage unit and then query or perform computation over it, stream processing differs in two key ways:

1. Events substitute messages. Verticles in SenseRT react to the incoming stream of events instead of a message or a batch of messages. Many sensors will send periodic updates reporting the status quo – these are typically not of interest to verticles, which are concerned rather with events indicating some change of state. For instance, a verticle controlling lights in a room might only be interested in the events indicating a change from unoccupied to occupied, or vice versa, and not in processing periodic messages of current occupancy which the sensor sends. Working with events also ensures that SenseRT handles timeliness of the event being processed by a verticle.

2. Reversing the norm. Unlike the store-then-analyse approach, stream processing focuses first on enabling real-time reactive processing of data (events). Upon receiving a relevant event, a stream processing application (a verticle in SenseRT) reacts to the event by updating some information, creating another event, or simply storing it. The result is that data can still be archived for historical
processing, but this does not negatively affect the performance of real-time processing.

There are four classes of verticles that SenseRT requires: (i) **Data ingestion verticles**, which receive data from the MQTT broker and publish the same on the EventBus (e.g., FeedHandler), (ii) **Data storage verticles**, which subscribe to the EventBus for any new data and store it for future usage (e.g., MessageFiler), (iii) **Real-time analysis verticles**, which analyse the stream of data in real-time and publish updates on the EventBus, and (iv) **Outbound verticles** which make the data available to the outside world. As shown in Figure 5, additional verticles include the MessageRouter verticle, used to share data to other similar systems; the Data Monitor, used to interact with client-side applications; and the Real-Time Analysis verticle, comprising one or more verticles and performing tasks like identifying events such as the measured CO\(_2\) level crossing a threshold or a power outages, and broadcast the results of such analysis as derived events on the EventBus.

We set up three backup servers, which could act as the primary server during any outage. These backup servers subscribed to the EventBus to receive any new message from a sensor. This ensured that the backup server had all the data that the primary server had.

5 CASE STUDY

We next examine an end-to-end example of a system using SenseRT to fuse sensor data to provide a useful application: a modernisation of the *Trojan Room Coffee Pot* [58]. The original deployed one of the first webcams to monitor how full was a research group’s coffee pot.

In our modernised version, where an opaque coffee pot renders the webcam approach ineffective, we measure and transmit in real-time the coffee-making and consuming events of the coffee pot in one of our buildings. The system analyses data from a set of sensors deployed at the coffee pot to recognise one of five events; (i) *pot-removed*, indicated the pot is not present, (ii) *new-pot*, indicating the presence of freshly-made coffee in the pot, (iii) *pot-poured*, indicating that coffee has been poured, (iv) *pot-empty*, indicating that no coffee remains, and (v) *coffee-grinding*, indicating that the coffee bean grinding machine appears to be active.

The coffee pot setup (Figure 10a) is designed as a sensor node, a coordinated collection of multiple sensors: weight sensors connected to the Raspberry Pi periodically monitor the weight of the coffee pot, while two smart plugs monitor the power usage of the grinder and the coffee brewing machine respectively. The Raspberry Pi also provides a Wi-Fi gateway to connect the two smart plugs. The sensor node accumulates data from each sensor and transmits a message to the local MQTT broker over Wi-Fi. The message consists of the weight and power readings and the time when the reading was recorded.

After being homogenised by the corresponding decoder, the message is published on the EventBus by the FeedHandler verticle. The MessageFiler, subscribed to the EventBus for any new message receives the new message and stores the attached data. A real-time analysis verticle, RTCoffee, looks for two types of events in the published data: did the power consumed by either the grinder or the coffee machine cross a threshold (40 W in our case), indicating the grinder or the coffee machine was in use; and has the measured

Figure 10: The Coffee Pot sensor node. The pot weighs \(~0.5\) kg, and when full holds \(2\) kg of coffee. Each cup taken is \(~0.25\) kg.
We then compare against some of the existing alternative systems. We show that SenseRT provides an architecture for minimal latency real-time data processing. Many sensors were added during the experiment period, and we examine how this impacted performance. We then compare against some of the existing alternative systems to highlight the advantage of the design choices made in SenseRT.

### 6.1 Real-time Performance

As a real-time architecture, it is important for SenseRT to show minimal latency between the generation of data at the sensor and the consumption of data by an application. In order to validate this, we measured the latency at four key points in the SenseRT architecture: (i) the gateway receiving the first hop message from a sensor (ii) the bridge MQTT broker in the Integration layer, (iii) the EventBus in the Data layer, and (iv) a client-side application similar to that described in §5.

![Figure 11: Events observed by RTCoffee.](image)

(a) The level of coffee in the pot at a given time. (b) Level of coffee in the pot after a coffee-grinding event with approximate time remaining. (c) Following a pot-removed event.

#### Table 3: Latency at key points of data flow in the SenseRT architecture. All values are in ms and are calculated from the time of message generation at the sensor.

|                | Gateway | MQTT Broker | EventBus | Client Application |
|----------------|---------|-------------|----------|--------------------|
| Mean (ms)      | 57.15   | 147.86      | 157.86   | 159.55             |
| Std Dev (ms)   | 10.21   | 63.56       | 2.35     | 0.56               |

As is evident from Table 3, at all four points latency is minimal. With as low as 57ms average latency at the gateway, an increase is observed at the integration layer (approx 90ms in average) owing to the processing involved with the bridging of information from different protocols. The messages are published at the EventBus with a latency of just 10ms, while the client-side application receives the desired data with almost no latency (2ms in average). For all practical purposes, an average latency of 159.55ms is a negligible value which validates SenseRT as a real-time system.

Furthermore, SenseRT is robust enough that addition of new sensors or category of sensor in the system doesn’t affect the overall working of the architecture. It is clear from Figure 13a that even on increasing the number of sensors in the system, the overall latency remains close to a similar mean value of 200ms. This is true also for when we calculated the mean latency for the different category of sensors used in the implementation of SenseRT (Figure 13b). Here, the maximum average latency is observed for DeepDish (400ms) which is because of 200ms processing time involved in computing the number of people from a video frame.

### 6.2 Comparison Against Alternatives

We compare different aspects of SenseRT with three similar systems. The first is the work by Al-Ali et al. [10] which sets up a Wireless Sensor Network with each sensor interfaced with a data acquisition system on a chip. The sensors exchange data through an MQTT broker which is then sent to a central server for analysis. Any analysis in this system is request/response based. The second system by Fayyaz et al. [29] does something similar, however uses a fog server as the first hop point for the sensor messages. The third work we compare with is developed by Evangelatos et al [28] which uses IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) to collect sensor information and store it for further processing. However, data is only requested for events such as a person entering a room, so no real-time data flow is required.

We compare systems on two fronts, (i) latency at the first hop point for the sensor messages (at the gateway), and (ii) latency for...
a complete data flow transaction from sensor to application. Both results were not reported for all the three competing systems, hence we only include the ones which were reported for each criteria. Also, the results for Fayyaz et. al. [29] are based on simulation.

First hop latency. We calculated the average first hop latency observed over all category of sensors which we report in Table 4. It is evident that the latency is at par with the state-of-the-art values (and in fact it beats them by a few ms).

End-to-End latency. When compared to the fog-based system [29], SenseRT performs 37 times better. This is primarily because the application server subscribing to the Data Monitor and hence receiving the messages almost at the same time as the EventBus (2 ms latency, as shown in Table 3). However, that system stores all the data on the cloud and then queries that from every application, increasing the latency. The 6LoWPAN system [28] does achieve almost similar latency as SenseRT, but only for the data stored at the server. Additional analysis would have further increased latency.

7 DISCUSSION
SenseRT provides a robust architecture for in-building real-time data flow but there remain some key elements requiring further work.

7.1 Scalability
This is more a feature of our prototype implementation than the SenseRT architecture itself. The prototype implemented in this paper covers only two buildings so far, with two more buildings in the deployment pipeline. It will be essential to observe the impact on performance as this number is increased.

As well as extending to cover more buildings, we are extending the types of sensors used in the implementation. We would like to integrate other types of data sources like Modbus or Monnit sensors [45], and understand how to fit them within the architecture.

As the set of sensors being deployed increases, optimising the number of sensors in the building becomes important. For instance, if a ZigBee and a LoRaWAN sensor provide the same readings, only one might be used dependent on the client’s needs. As sensors are increasingly integrated, a single sensor might provide multiple readings and so could replace multiple sensors. With more sensors in place, deciding the optimal placement strategy for effective building coverage is essential. Others have investigated this [39, 50], providing directions for finding a strategy for SenseRT.

7.2 Privacy
Privacy is a key concern with IoT-based systems, and many have looked into how to achieve this in smart buildings and smart cities [35, 36, 43, 51]. Currently, SenseRT provides basic privacy support through encryption provided by network protocols and limited data access. However, SenseRT is amenable for more fine-grained privacy approaches. Our intended approach has three aspects.

First, we will ensure all messages from any sensor are encrypted. As some sensors do not have the capacity to encrypt data at source, decisions will need to be made about where is the most appropriate point in the architecture to provide encryption, and what scheme should be used.

Second, we need to define mechanisms to support different data access strategies governing who can access what data. For example, a person might have access to all the data generated by sensors in her room, but only specific sensors on the floor. This could involve assigning the different stakeholders, building managers, third-party clients, or building occupants, into groups by which access is controlled. Another option could be to provide unique tokens to each stakeholder, and access is provided based on tokens. This could also include building visitors to whom limited data could be made available, e.g., in a time of COVID-19 with social distancing recommendations, current (but not historical) room occupancy could help keep visitors and occupants safe.

Third, under whatever access control regime is provided, we must still determine how occupants’ privacy should be protected. For instance, a building manager receiving power readings from smart plugs in a room every hour could infer when the occupant was
in their office, whereas receiving overall power usage for an office over a week might be sufficient for managing energy efficiency in a building. We thus need to examine how client applications can receive sensor data so as to ensure privacy while still meeting the differing goals of users of the system.

8 CONCLUSION

As the world moves towards more and more smart buildings, we anticipate considerable (perhaps exponential) increase in the number of sensors and consequently volumes of real-time data generated. Building Management Systems (BMSs) need to be re-architected to better support both data management and reliable real-time information dissemination. SenseRT provides a robust architecture satisfying these goals by considering several key aspects: (i) spatio-temporal characteristics of real-time sensor data, (ii) improving timeliness and maintaining low latency throughout, (iii) ensuring a homogenous data flow through the architecture notwithstanding the wide range of sensor types and capabilities, and (iv) performing efficient real-time analysis of sensor data with minimal latency.

Our implementation of a prototype and the experiments carried on for seven months show that SenseRT does provide an efficient network architecture for in-building real-time data flow and real-time data analysis. There do exist key aspects of scaling and privacy, which would improve SenseRT further. However, as it stands, SenseRT is a first step towards providing a robust network architecture that could tackle the incoming challenges BMS faces regarding the increasing volume of sensors and the real-time data being generated by these sensors every day.

REFERENCES
