SmartCloud - Normalizing IoT Data for Event-Driven Architectures

Brian Granatir
Smart Parking

John Heard
Smart Parking

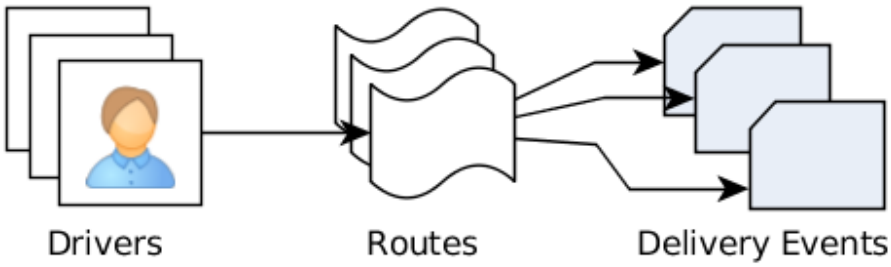
Pyit Phyo Aung
Smart Parking

Abstract

With the rise of IoT management sub-platforms (such as Google IoT Core and AWS IoT Platform), organizations are able to easily gather massive datasets from distributed devices. While previous research has focused on the methods and methodologies of collocating data streams from IoT devices, few studies have been published on correlating this data for interpretability and action. In this paper, we propose a set of guiding principles to help normalize incoming data from disparate devices spatiotemporally with the express purpose of deriving both analytical and actionable meaning. After the theory, we'll explore implementation using the context of the SmartCloud Platform, an event-driven architecture currently in use by Smart Parking Ltd to deliver Smart City capabilities to our global customers. The SmartCloud Platform uses 'cloud native' and 'API First' implementation principles. The data architecture is designed to avoid restrictively static data schema and utilise the continuum of schema options from highly structured through to the developing dynamic multi-path graph frameworks which can be quite naturally used within machine learning based services. As a platform, the SmartCloud Platform delivery strategy is based upon the premise that 'innovation can occur anywhere' and thus we are adopting an open technology access strategy with several mechanisms of access for solution creators to be able to leverage combinations of functional or information capabilities available from the SmartCloud Platform.

1 Introduction

With cloud-based IoT management sub-platforms now publically available (available from providers such as Google, Amazon, and Microsoft), onboarding of distributed data streams has become a cost-efficient opportunity for many organizations. For most, this means gathering sensor data from various internet-connected instruments deployed across large, and often dynamic, geographical zones. Problems arise when introducing a 'second device', a new data source that provides valuable data that has little in common with already established inputs. Typically, this means adding custom logic to handle this new stream, resulting in a tight coupling between these two data sets. For example, a shipping organization may have devices installed on transport vehicles to monitor routes. A second device could be introduced to monitor capacity utilization. The development team could easily form linkages between drivers, routes, and capacity to determine the efficiency of delivery plans (i.e., are we limiting transport time while maximizing vehicle usage?). The model for store this data could look like:



This model answers our initial questions, but what if we add additional data points? Consider incoming traffic data. Can we easily reroute our vehicles based on outstanding delivery events? What about weather data, gas prices, tire pressure, and cargo-hold temperature? Our development team could certainly expand an existing data model, but our system will soon begin to resist adoption of new devices.

In the following sections, we'll explore how to normalize data around spatiotemporal data points readily available from any IoT connected device. We expand on this by exploring how this data can easily integrate with an event-driven architecture. Next, we'll examine how these principles have been implemented using serverless microservices on a released Smart City platform. Finally, we'll give a brief overview of how analytics and business rules leverage this architecture.

2 Normalizing IoT Data

If we ignore security, the onboarding of IoT data follows a standard flow:

```
|     message packing    |     |   transfer    |     |     message unpacking    |
-------------------------     ----------------     ----------------------------
[1. Sensor]->[2. Gateway] -> [3. IoT Manager] -> [4. Routing]->[5. Decoding]
```

1. Readings from the device sensor(s) are encoded and broadcast to the gateway
2. The gateway wraps this message and sends it over an internet connection (note: some devices will have their gateway built in, while others will use a gateway hub)
3. The IoT management platform accepts the message and passes it to the backend
4. Message is routed, based on the message [device] type
5. Message is decoded and normalized to a common API

The first challenge of onboarding IoT data is the varied nature of the messages. For example, the data coming from a smart fridge would be vastly different from the streamlined, encoded readings sent over radio frequencies from parking sensors buried in city streets. To overcome this challenge, we define a common wrapper to help route incoming messages to their proper decoder. A common implementation for this wrapper in JSON would be:

```
{
  message: "HSIFIG3KT2NSF43IT23",
  messageInfo: {
      type: "BadgerSensor",
      version: "1.0.5"
  }
}
```

While the message field contains the raw sensor payload (typically JSON or an encoded byte string), we can perform routing without having to unbox it by having a standard messageInfo block. This is where we see the importance of having the sensor(s) and gateways be logically separate. The sensor should only focus on reporting readings. It's up to the gateway to handle packaging and shipment of this payload.

Also, currently deployed devices can redirect to different central platforms by simply changing or supplementing the gateways that service them. It is reasonable to expect custom gateways for specific backends. In fact, the Smart City offering explored in the implementation section below offers a SmartSpot gateway that will package any incoming broadcast, regardless of the origin device type.

2.1 Duplicate, Missing, and Misordered Messages

Now that we're onboarding and decoding data in messages, we can focus on the second challenge of IoT data streams: windowing. As with any distributed systems, our architecture must account for duplicate, missing, or misordered messages. In a sequence of network hops, there is always the potential for delivery failure. Since our goal is to make actionable meaning, incorrectly ordered events could be detrimental. For example, imagine if two temperature readings came in out of order (e.g, the initial lower reading comes in after the subsequent higher reading). Our system may incorrectly interpret a spike in temperature as a recovery.

The quickest solution is to add a timestamp to our message wrapper to allow us to perform proper ordering and backfilling if a message should arrive late:

```
{
  "gateway": {
      "id": "1232",
      "type": "SmartSpot",
      "timestamp": 1496025214182,
  },
  "message": "HSIFIG3KT2NSF43IT23",
  "messageInfo": {
      "type": "BadgerSensor",
      "version": "1.0.5"
  }
}
```

The timestamp is reported by the gateway when the message was received. While the timestamp could (and should) be contained in the message, we cannot assume all sensors will have synced clocks.

With a consistent timestamp, we are able to handle basic ordering and introduce advanced windowing techniques. Many fabulous papers have been written about windowing, including [1, 2]. The goal of this paper is to simply emphasize the importance of having a timestamp that originates with the wrapping event (as opposed to further downstream) to assure eventual ordering.

Business logic must be aware of the state of windows when making any action. More on this will be discussed in our implementation section.

2.2 Spatiotemporal Messages

Now that we have a strategy to deal with identifying and windowing, we're ready to address the final requirement of normalizing IoT data: relating. Our services are unable to apply generic logic or analytics to data points without having an implicit or explicit relationship. While our Smart City implementation adds an additional level of explicit relation based on a defined site structure (more can be found in our implementation section), this is not required to achieve a baseline of meaning. Instead, we can focus on the most commonly available datapoint: location. Let's look at a couple of examples.

First, consider 800 devices that report the temperature of manufacturing equipment throughout a factory. Assume an event where twelve devices report a dangerous increase in operational temperature. With a basic system, we could react by turning off this machine and calling for maintenance. However, if we know that these twelve devices are nearby we can react to the potential of a more significant event, such as a fire.

Second, consider 3500 devices that detect parking events by using IR/magnetic sensors distributed throughout a city. Assume an event where 600 of these devices report a departure. With a basic system, we can update nearby signs to reflect the current count of vacancies.

However, if we know that these 600 devices are all located downtown (perhaps a concert just ended), we can react by changing traffic lights to assist with outbound traffic.

Fortunately, geolocation data is readily available and can be included with our primary message packet:

```
{
  "gateway": {
      "gatewayId": "1232",
      "gatewayType": "SmartSpot",
      "timestamp": 1496025214182,
      "lat": 80.614842,
      "long": 95.273438,
      "alt": 160
  },
  "message": "HSIFIG3KT2NSF43IT23",
  "messageInfo": {
      "type": "BadgerSensor",
      "version": "1.0.5"
  }
}
```

By adding latitude, longitude, and altitude data during message wrapping, we're able to create an implied relationship between all our inbound messages. For gateway hubs servicing multiple devices, an additional lookup may be required at the time of unwrapping (e.g., we conduct an additional lookup to determine that device FOO was installed in unit BAR at location-36.831307, 174.893675). In such systems, the device would need to either self-identify within the payload or be identified by the packaging gateway. As such, our final inbound message would look like:

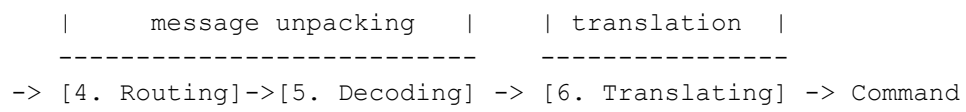
```
{
  "deviceId": "12AB75FG",
  "deviceType": "badger",
  "gateway": {
      "gatewayId": "1232",
      "gatewayType": "SmartSpot",
      "timestamp": 1496025214182,
      "lat": 80.614842,
      "long": 95.273438,
      "alt": 160
  },
  "message": "HSIFIG3KT2NSF43IT23",
  "messageInfo": {
      "type": "BadgerSensor",
      "version": "1.0.5"
  }
}
```

Since devices may move (i.e., our proximity relationship to other readings may shift over time), we must also include temporal data. Fortunately, we already included timestamp data for ordering purposes. By combining location and timestamp data we've normalized all our input spatiotemporally. Our next section will discuss how to integrate this normalized data into an event-driven architecture.

3 IoT into Event-Driven Architecture

Event-driven architecture requires one simple principle: the output of every functional unit must be either a command or an event. Commands trigger events and events can trigger new commands. However, this causality must never be broken. A command cannot beget another command, just as an event cannot spawn another event. While this architecture may seem limiting at first, it marries well with streaming data and allows us to leverage principles of temporal reasoning for our business rules.

In the previous section, we learned how to normalize IoT data by using simple message wrapping. Hooking this data into a standard event-driven architecture is as simple as adding an additional step: translation.



For event-driven architectures, this means creation of a command. In most instances, our device readings will result in a UpdateReadings or UpdateHealth command. Of course, if a device or gateway natively speaks the command API, the decoding and translating steps can be skipped.

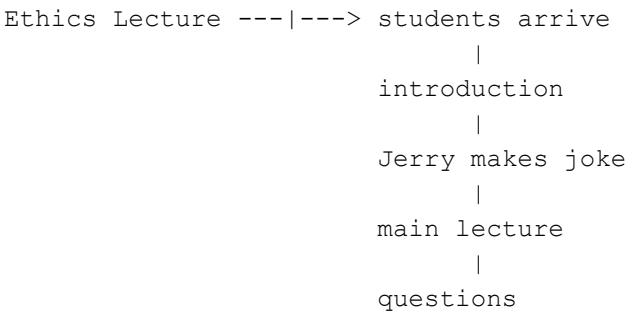
We could end our journey here and incorporate a Complex Event Processor BRMS (like Drools) to handle our logic, but there are several key challenges that most Complex Event Processor (CEP) services cannot handle with the massive streaming input that we'd expect to see from a robust, city-wide IoT network. First, current CEP offerings are monolithic. They've yet to embrace a distributed deployment and synchronization design that is best suited for auto-scaling architectures (especially serverless). Second, to support the expected throughput, most CEPs would need massive memory footprints that are not practical and introduce a significant state-maintenance risk. Third, CEP rules require a predefined understanding of the input structure and won't easily allow meaning to be derived from previously unknown devices. Thusly, without a CEP, we need one further step to help with analytics in event-driven architecture: aggregation.

Aggregation is a view of objects in an event-driven architecture at a moment in time defined by the sequence of events before it. In other words, we're creating an aggregate view of an object based on seen events. If humans lived in an event-driven architecture, we could, in theory, 'rebuild' the person by replaying all events on his or her Facebook timeline. While this concept is very powerful, only recent advances in cheap data storage and queueing services have made this methodology practical. More in our implementation section.

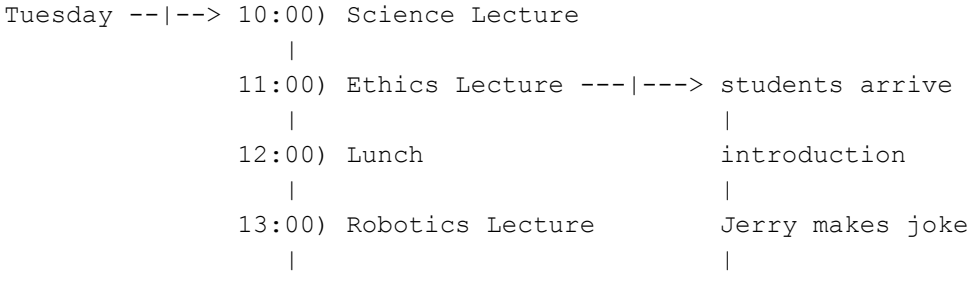
Typically, aggregation is used to maintain traditional objects within an event-driven architecture. For example, we could keep track of health message events to build a view of a specific deployed device on our IoT network. However, we've found that an aggregation of correlated events provides a powerful and generic way to view data entering a platform. In other words, we're making a larger event by aggregating together smaller events. These larger events allow us to find meaning and drive action.

3.1 Event of Events - Temporal Reasoning

Applying temporal reasoning within a computer system is best done through the context of intervals [3]. In summary, we gain insights and meaning by grouping smaller events into a larger event interval. For example, if particular larger event is a lecture, the lecture itself can contain many smaller events:



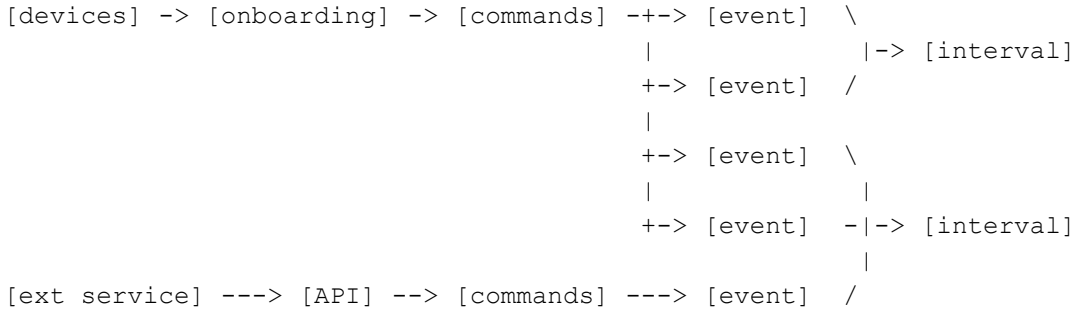
We can also take the lecture in the context of an even larger interval, such as a work day:



By creating intervals, we can store knowledge about generic events that can help drive analytics or system behaviour [4]. Moreover, as we'll see in our implementation section, we can build a simple tagging system to allow us to manage business rules on a per entity basis.

However, before we can start acting on intervals, we must find a consistent way to create and expand them. Fortunately, event-driven architectures accommodate this aggregation. A system must simply select a granularity that matches their domain. In our implementation section, we'll see a Smart City domain that focuses on a standard site stratification (regions -> sites -> sectors -> units). Our goal is then to start aggregating events based on these levels of granularity.

For example, let's say we were grouping events based at a site level. We would then aggregate incoming device events that share a common site. Our domain should define the the interval structure. For a site, this may be shifts or work days. For a unit, this may be individual occupancies or worksets. We can now see the flow of IoT events as aggregated intervals:



Moreover, since our incoming messages are all commands, we can ingest data from any external services, not just devices. A common example would be reservation information provided by an external payment app.

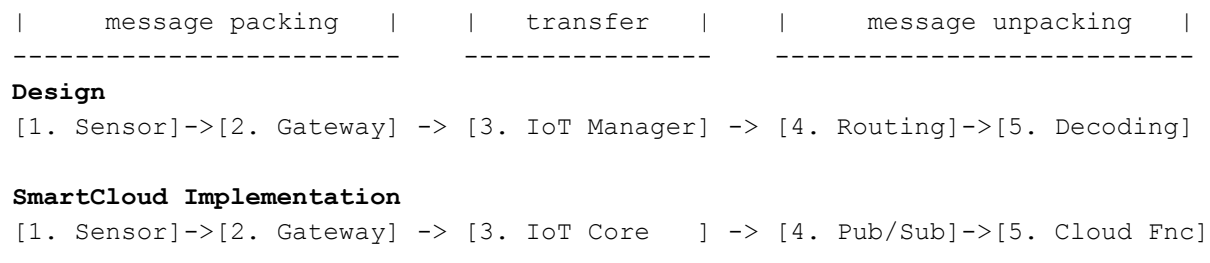
Therefore, to gain knowledge about normalized IoT data, an event-driven architecture must simply pick domain granularities and generate intervals (grouped events) based on that design. Note: a single event can appear in many different intervals based on granularity, as we'll see in our implementation section.

4 Example Implementation

The SmartCloud Platform is a production service for management of Smart City services negotiated through APIs. An overview of Smart Cities and their relationship to IoT can be found in [5] and a review of microservices to support such a model can be found in [6]. SmartCloud is built on the Google Cloud Platform (GCP) and implements the principles outlined in this paper. The goal of this section is to provide a brief overview of how covered concepts can be developed and deployed for use by any generalized IoT backend.

4.1 Onboarding

IoT devices are registered with Google IoT Core at the time of installation or site definition. Messages are placed into Pub/Sub queues that feed into Cloud Functions for decoding and translation into commands. In the context of our original onboarding flow, we establish the following workflow:



With the rapid scaling and low costs of Pub/Sub and Cloud Functions, the platform is able to handle significant data streams at reduced costs. Using microservice principles, expansion of the service is also rapid and low-risk. Messages are normalized based on timestamp and geolocation coordinates either provided by the device, gateway, or site structure.

4.2 Site Structure and Granularity

In SmartCloud, aggregation of events is based on a simple hierarchy:

```
Regions -> Sites -> Sectors -> Units -> Devices
```

One or more devices can be installed per unit. For example, if we're deploying parking sensors, then we can see each unit as being a bay. If we're deploying street lights, we can see each unit as a pole or assembly. A site has multiple sectors and sectors have multiple units of various types. Each level of the hierarchy contains GPS coordinates that helps with spatial stamping.

APIs target entities at any of these levels. For example, an external service can make a payment against a unit or a reservation can be made against a specific site.

4.2 Creation of Intervals

Temporal intervals are made for events arriving at various levels of of the SmartCloud site structure. Start and end points for these intervals are typically defined by utilization data arriving from devices. These intervals are stored within Datastore (Google's NoSQL service built on BigTable) as a time series. The rowkeys for this time series, and therefore indexing, follow the schema outlined in [6].

Therefore, an interval is defined by a collection of rows in Datastore. These events have vastly different data. While we could store all this data within the NoSQL database, we'd soon hit the column limit. To avoid this, SmartCloud utilizes a linkage system commonly found in graphing databases. Whenever a row is added to an interval, an edge is created that connects this row with the corresponding aggregate. A possible interval example for a parking event will look like:

```
                  ===== Event =====       == Linked Aggregate ==
Interval --|--> unit occupied  <------- Unit
                       |
                  payment made <--------- Payment
                       |
                  overstay check <------- Tag
                       |
                  overstay detected <---- Tag
                       |
                  online extension <----- Payment
                       |
```

```
overstay retracted <--- Tag
     |
unit vacated <--------- Unit
```

As such, edges are automatically created between the interval and the aggregates associated with all appended events. This concept of linkages also allows us to translate a sequence of events into a graph of related objects. This knowledge can be used to answer simple queries, such as "Does this parking event have a payment?" or "Is this interval linked to any incomplete stream windows (i.e., are any crucial events potentially missing)?"

4.3 Business Logic

With spatiotemporal normalized data collected into event intervals, SmartCloud offers data sets that provide meaning for both analytics and real-time business logic. Business rules are applied at various levels of the site hierarchy through inherited tags. These functional tags are designed to sort based on temporal relationship and then trigger a command based on established edges. In other words, we sort our rules based on time relevance and then process these rules in order until one or more 'breaks', triggering a new command within SmartCloud.

A tag that would trigger an overstay command 10 minutes after arrival would look like:

```
{
  sorterType: "javascript",
  sorter: "( startTime ) => { return startTime + (60000 * 10) } ",
  breakerType: "javascript",
  breaker: "( interval ) => { return { command: 'overstay', break: true } }"
}
```

More information on these tags, temporal reasoning, and the associated breaker algorithm can be found in a subsequent paper.

Analytic data is made available in real-time and report views via a customizable dashboard and through Google's BigQuery service by streaming event and aggregate data via Cloud Functions.

5 Conclusions

IoT management platforms, serverless computing, and cheap cloud-based storage have made ingestion and reaction to large data streams a possibility for any architecture. Normalizing spatiotemporally allows us to find meaningful relationships between seemingly disparate data. For event-driven architecture, we can use these relationships to easily create intervals of collected events. These intervals allow us access to business logic and analytics based on powerful temporal reasoning. This reasoning is made far simpler by creating edges to aggregates associated with events contained in these intervals, essentially creating a graph between grouped events and related entities.

6 References

[1] T. Akidau, “The world beyond batch: Streaming 102,” O’Reilly Data Science Blog, January 2016.

[2] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernandez-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, S. Whittle, “The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing,” Proceedings of the VLDB Endowment, vol. 8 (2015), pp. 1792-1803.

[3] James F. Allen, “An Interval-Based Representation of Temporal Knowledge,” Proceedings of the 7th International Joint Conference on Artificial Intelligence, vol. 1 (1981), pp. 221-226.

[4] James F. Allen, “Maintaining Knowledge about Temporal Events,” Communications of the ACM, vol. 26 (1983), issue 11, pp. 832-843.

[5] A. Zanella, N. Bui, A. P. Castellani, L. Vangelista, and M. Zorzi, “Internet of Things for Smart Cities,” IEEE Internet of Things Journal, 2014.

[6] A. Krylovskiy, M. Jahn, E. Patti, “Designing a Smart City Internet of Things Platform with Microservice Architecture,” The 3rd International Conference on Future Internet of Things and Cloud (FiCloud), Rome, Italy, August 2015.

[7] “Cloud Bigtable Schema Design for Time Series Data,” GCP Documentation, (https://cloud.google.com/bigtable/docs/schema-design-time-series), Sept 2017.

