Modeling and Building IoT Data Platforms with Actor-Oriented Databases

Yiwen Wang
University of Copenhagen
Copenhagen, Denmark
y.wang@di.ku.dk

Julio Cesar Dos Reis
University of Campinas
Campinas, SP, Brazil
jreis@ic.unicamp.br

Kasper Myrtue Borggren
SenMoS
Copenhagen, Denmark
kmyrtue@gmail.com

Marcos Antonio Vaz Salles
University of Copenhagen
Copenhagen, Denmark
vmarcos@di.ku.dk

Claudia Bauzer Medeiros
University of Campinas
Campinas, SP, Brazil
cmbm@ic.unicamp.br

Yongluan Zhou
University of Copenhagen
Copenhagen, Denmark
zhou@di.ku.dk

ABSTRACT

Vast amounts of data are being generated daily with the adoption of Internet-of-Things (IoT) solutions in an ever-increasing number of application domains. There are problems associated with all stages of the lifecycle of these data (e.g., capture, curation and preservation). Moreover, the volume, variety, dynamicity and ubiquity of IoT data present additional challenges to their usability, prompting the need for constructing scalable data-intensive IoT data management and processing platforms. This paper presents a novel approach to model and build IoT data platforms based on the characteristics of an Actor-Oriented Database (AODB). We take advantage of two complementary case studies – in structural health monitoring and beef cattle tracking and tracing – to describe novel software requirements introduced by IoT data processing. Our investigation illustrates the challenges and benefits provided by AODB to meet these requirements in terms of modeling and IoT-based systems implementation. Obtained results reveal the advantages of using AODB in IoT scenarios and lead to principles on how to effectively use an actor model to design and implement IoT data platforms.

1 INTRODUCTION

Internet-of-Things (IoT) systems enable data interactions through machine-to-machine communication stemming from supporting devices connected to the Internet [13]. IoT systems generate a potentially huge amount of data from devices that dynamically enter and leave the IoT environment, with very high-speed data flow and processing. Data, in turn, are generated by a wide variety of devices, thus giving rise to highly heterogeneous data streams. In this paper, we distinguish between *IoT systems* (i.e., the entire software ecosystem involved in an IoT scenario) and *IoT data platforms* (i.e., the data and data management software modules that are part of an IoT system). Our work focuses on the latter.

Enormous challenges need to be addressed in order to realize the full potential of IoT. First, there is a tension between effective data management and fulfillment of performance requirements in IoT data platforms. Indeed, many IoT systems are processor-intensive and require processing a massive amount of highly concurrently generated data. The management of these interactions among data with low latency remains an open research problem. Second, being able to deal with dynamic scaling while guaranteeing protection of data from different entities is another significant challenge. Therefore, we focus our investigation on how to manage the data from large volumes of devices and, at the same time, ensure the dynamic and flexible development of applications. This dual aim must be achieved while respecting application constraints for low latency in interactive functionality as well as data protection and access control.

Given these characteristics, we propose that actor-oriented databases (AODBs) are ideally suited to manage the data of real-world IoT systems. Actors comprise a model of computation specifically aimed at high concurrency and distribution [1]. To that effect, actors keep their private states and can modify states by communicating with each other via immutable asynchronous messages [19]. As such, actors are natively applicable to support the management of an arbitrary number of independent and heterogeneous streaming data sources. AODBs, in turn, enrich actors with classic RDBMS functionality by integrating data management features, such as indexing, transactions, and query interfaces, into actor runtimes [17]. These features make AODBs attractive for building an IoT data platform. In more detail, AODBs stand out for several reasons. First, IoT systems comprise many different devices with distinct functionality. This requirement is directly met by the actor model, through the principle of assigning different logic and tasks to actors. Second, in IoT, data changes frequently; actors provide a natural alternative to conventional concurrency models that rely on synchronization of shared mutable state using locks. Third, the characteristics of non-blocking interactions via immutable messages between actors match well with the demands of IoT systems. Fourth, the number of actors can scale out quickly without consuming excessive resources. Dynamic scaling is a common situation in IoT in which all kinds of sensing devices (including humans!) can quickly enter – but also leave – a system.

There are several examples of the use of actors in IoT scenarios [4, 38, 41, 43]. However, these previous studies concentrate on implementation aspects, neither providing guidance on how to model IoT data platforms with actors nor analyzing the fit of AODB to the requirements and challenges brought about by IoT. By contrast, to the best of our knowledge, this paper is the first that builds an end-to-end case for the suitability of AODBs to manage IoT data, going from requirements and modeling to implementation and performance evaluation. Our work covers a wide gamut of issues to justify and showcase the adoption of AODBs as an appropriate solution to meet the main challenges of data management in IoT systems. Our main contributions are therefore:

(1) We discuss core requirements of IoT data platforms, and challenges to be met in their implementation. We illustrate

this discussion through the analysis of two real world IoT case studies.
(2) We present a methodology and guidelines to model an AODB for such platforms.
(3) We develop a prototype of one of the case studies and present its evaluation to show the effectiveness of adopting AODBs for IoT data platforms.

The remaining of this paper is organized as follows. Section 2 presents two case studies of IoT systems, which we use to extract functional and non-functional requirements, and to present some of the major challenges to be faced. Section 3 justifies our choice of AODBs as an appropriate technology to meet such requirements and challenges. In Section 4, we provide a detailed discussion of the challenges of modeling such platforms, and show how these challenges can be overcome for the running cases. Sections 5 and 6 respectively present our prototype for one of these cases and its evaluation. Section 7 revisits the paper by contrasting it with related work. Finally, Section 8 presents conclusions and ongoing work.

2 IOT DATA PLATFORM CASE STUDIES

In this section, we discuss the requirements for an IoT data platform and analyze two specific case studies. There are several scenarios in IoT data platforms, such as healthcare, personal security, traffic control, environmental monitoring, and disaster response. The two IoT data platform cases that we focus on are drawn from our experience with a structural health monitoring system (cf. Subsection 2.1) and a beef cattle tracking and tracing system (cf. Subsection 2.2). We have worked directly with these case studies, helping us validate common non-functional requirements for IoT data platforms as well as collect illustrative functional requirements for these applications.

For the first study, we have cooperated with SenMoS [45] in the area of structural health monitoring for large constructions, e.g., bridges. SenMoS is a Danish company that provides users with entire monitoring solutions, including requirement elicitation and cloud data management. The developers at SenMoS have participated in the design and implementation of the IoT data platform for the Great Belt Bridge [50]. The second case focuses on the management of cattle produce (and in particular the beef supply chain) from the perspective of traceability. This study is based on previous work with domain experts from the Brazilian agricultural research corporation Embrapa [28] that studied traceability in food for supply chains [35], and on interactions within the Danish Future Cropping partnership [30], particularly with experts from the agriculture solution provider SEGES [44]. Both of these organizations have substantial experience in the agricultural sector and are key players in agricultural extension systems of the respective countries. Both case studies concern the development of a scalable data platform that collects and stores data from IoT devices, processes operations, and provides information services to different users. Although these two IoT platforms target different scenarios, they present several common aspects and non-functional requirements. In particular, the systems should operate as Software-as-a-Service (SaaS) solutions and thus manage the data from several different tenants. Moreover, it is desired that scalability to large data volumes or users be achieved without a high burden on the data platform developers.
Non-Functional Requirements for IoT Data Platforms. We elicited the following common non-functional requirements shared by different IoT data platforms:

(1) **Data ingestion from endpoints**. The IoT data platform must have the capability to receive and store data from IoT devices, e.g., GPS collars on cows.
(2) **Multi-tenancy**. The IoT data platform must provide varied information services to different users.
(3) **Support for heterogeneous data**. The IoT data platform must be modular in its support for data ingested from IoT devices and allow for communication employing different data formats.
(4) **Cloud-based deployment**. The IoT data platform can be distributed in the cloud for ease of operation, management, and maintenance.
(5) **Scalable data platform**. The IoT data platform must not degrade in functionality or performance while expanding. This must occur without modifying existing software components.
(6) **High efficiency**. The IoT data platform must process massive amounts of concurrently generated data effectively.
(7) **Access control and data protection**. The IoT data platform should support data protection, enforcing authentication and access control over different users and profiles.

In addition to the requirements above, it is often the case that IoT data platforms must serve queries over historical data accumulated from devices over long time periods. In this paper, we focus, however, on online data ingestion and querying in SaaS scenarios. We note that at present, there is only limited support for declarative multi-actor querying in AODBs [17], and thus complex historical analyses could still be served by a data warehouse.

2.1 Case 1: Structural Health Monitoring

Structural Health Monitoring (SHM) systems aim to identify damaged sections on parts of large constructions that can cause safety concerns. SHM systems can help organizations save time on inspections by gathering and processing data so that the system can generate alerts when problems arise or suggest actions that can prevent faults. SHM systems are equipped with a set of sensors, e.g., to measure a bridge's extension, inclination, temperature, wind speed, and wind direction. Each sensor is connected to a data logger that converts the sensors analog signal into a digital one. The platform must collect, process and store data from the sensors. Figure 1 presents a context diagram of the Structural Health Monitoring Data Platform. This design is based on a real case study. Sensors provide data to different stakeholders, e.g., engineering experts monitoring the structure, data analysts, or the

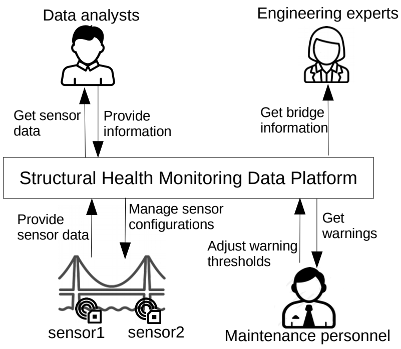

Figure 1: Context Diagram for Structural Health Monitoring Case Study.

maintenance personnel who manages the monitoring projects. The SHM system must meet the following functional requirements:

(1) The system must control several construction structures (e.g., bridges) using the same data platform.
(2) The system must be structured to support data storage, i.e., data must be saved in a way that allows for further data manipulation and analysis.
(3) The data platform must be able to maintain data from multiple sensors, users, projects, and organizations.
(4) The data platform must calculate the accumulated change for each data stream from a sensor, e.g., to gauge how far elements have moved when using extension sensors.
(5) The data platform must send customized alerts to users when thresholds are met, depending on individual sensors or sensor types. Thresholds can be used for determining the need for maintenance, or to call attention to ongoing events.
(6) The data platform must support plots providing statistical aggregates to help users spot meaningful events in time series. Besides, online plotting of recent raw sensor data is required to let personnel explore events interactively.
(7) The data platform must allow for browsing of live data from sensors, along with continuously derived equations, to provide a view of the current state of the structure.

2.2 Case Study 2: Beef Cattle Tracking and Tracing

Agricultural supply chains involve a complex network of producers, retailers, distributors, transporters, storage facilities, and consumers in the sale, delivery, and production of a particular product. Trackability and traceability are essential requirements in food marketing [24]. Tracking refers to following the path of an entity from the source to destination. Tracing refers to identifying original information regarding an entity and tracing it back in the system [37]. Systems for tracking and tracing agricultural products increase consumer confidence on provenance and quality of the food they buy, while at the same time helping retailers and certification authorities to monitor products.

The ability of IoT to collect data from sensors as well as trace entities is a crucial enabler for monitoring such chains. Systems that automate tracking and tracing in an agricultural supply chain should not only collect data, but also connect users and objects at any place and time. Data integration, processing, analysis and service support present many challenges in this context. For the sake of feasibility, we assume for this case study, similarly to other food tracing systems [33], that a global standard for supply chain messages, GS1 [32], is adopted by participants that connect with the IoT data platform. As such, we do not discuss the data integration problem in this paper.

Our case study refers to a part of the beef cattle supply chain, concentrating on cow tracking and meat product tracing, providing tracking information and helping consumers trace meat products. Figure 2 presents the entities interacting within the data platform. The system must provide multi-tenancy services to host the data of different participants and supply chains. From a high-level point of view, there are five kinds of tenants in our system: farmers, slaughterhouses, distributors, retailers, and consumers. Each involved part is the source of different types of data in the system to enable the tracing of the whole life-cycle of a given meat product.

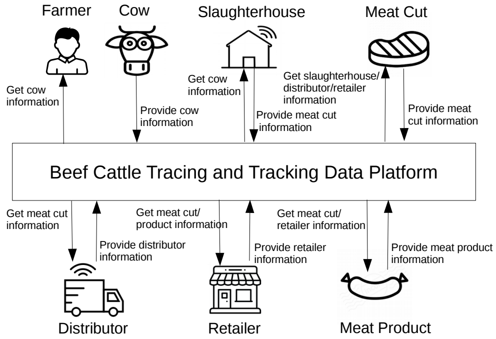
Farmer
Cow
Slaughterhouse
Meat Cut
Get cow information
Provide cow information
Get cow information
Provide meat cut information
Get slaughterhouse/ distributor/retailer information
Provide meat cut information
Beef Cattle Tracing and Tracking Data Platform
Get meat cut information
Provide distributor information
Get meat cut/ product information
Provide retailer information
Get meat cut/ retailer information
Provide meat product information
Distributor
Retailer
Meat Product

Figure 2: Context Diagram for Beef Cattle Tracking and Tracing Case Study.

Full-fledged cattle sensor-based systems involve the deployment of very many kinds of sensors – both in individual animals and in their environment. For instance, each animal has external sensors (e.g., collars, earrings) to measure movement, speed, location. Cattle often also have sensors inside their digestive tract (usually swallowed, sometimes implanted), to measure factors such as temperature, metabolic variables, or digestive characteristics. Environmental sensors may monitor factors such as cattle weight, or soil humidity. Additional sensors along the supply chain include devices that provide trackability (e.g., in transportation), but also traceability and quality (e.g., monitoring temperature inside warehouses). Without loss of generality, we have simplified this scenario to consider only a few of these sensing sources, keeping only enough distinct sensors to illustrate actual data and sampling rate heterogeneity. This simplified scenario must fulfill the following functional requirements:

(1) The data platform must store the data from animal and environment sensors, such as collars bound to each individual cow, to enable retrieval of location, motion, and other facts regarding traceable entities.
(2) Farmers need to track each cow's trajectory and behavior, and thus the data platform must record the locations of each cow over time. Geo-fencing can help identify whether a cow is in an appropriate area (e.g., when rotating pasture grounds) [20].
(3) Slaughterhouses wish to access services that provide information about cows that will be slaughtered. For instance, it must be possible to access tracing information such as the provenance of the cows and tracking information about where the meat cuts produced after slaughter are transferred to.
(4) Distributors wish to get tracing information of a meat cut and tracking information of where those cuts are to be sent to.
(5) Retailers aim to know the source of the meat cuts and manage their transformation into meat products for consumers.
(6) Consumers wish to get tracing information about meat products over the whole supply chain.

2.3 Challenges for IoT Data Platforms

The functional and non-functional requirements discussed in this section render the modeling and building of IoT data platforms a non-trivial undertaking. The construction of such a platform involves technical issues related to capturing, identifying and storing relevant events, managing associated constraints, processing varied types of queries, etc. Further complexity arises from taking into account the necessities of different stakeholders, and issues related to data precision, synchronization and availability.

Choosing the right database architecture is therefore a key decision for the success of an IoT data platform project. Virtualized deployment for efficiency and ease of scaling to large request volumes are significant obstacles [46]. Moreover, support for multi-tenancy needs to be carefully designed. While physical sharing of tenant data lowers overhead and increases efficiency [7], this strategy opens up security risks, which are related to the lack of modularity at the database level [47].

Thus, several questions need to be addressed when building IoT data platforms. First, a vast amount of data is concurrently generated from IoT devices. How can this data be managed and processed in the data platform? Second, how can data protection and access control across different entities be enforced while sharing data effectively? Third, our case studies suggest that a variety of queries, including analyses of time series from bridge sensors, spatial queries for cow locations, or graph navigation for tracing, need to be efficiently supported over IoT data. How can applications be modeled and built to support different types of queries? Fourth, it is necessary that the system be easily scaled without affecting functionality and performance. How can the platform be architected to easily scale out when it becomes necessary to manage more users and data? In this investigation, we observe that the issues regarding modularity and scalability pointed out in this section can be simultaneously addressed by AODBs [17]. We design cloud-centric actor-oriented database backends for the two IoT case studies introduced in this section. As a first step, we explain the motivation of taking an approach based on AODBs in the next section.

3 WHY ACTOR-ORIENTED DATABASES?

We argue that an actor-oriented database is the ideal organization for an IoT data platform, enabling fulfillment of all common non-functional requirements identified in Section 2. Moreover, AODBs ease the achievement of functional requirements by providing a modular, stateful, and scalable substrate for the modeling, design and implementation of an IoT data platform. The following characteristics of an AODB illustrate its suitability to address the challenges of IoT data platforms.

AODBs facilitate the management of distribution and the encapsulation of data. Actors are logically distributed, and can thus naturally map to dispersed entities such as sensors. The latter promotes the expression of parallelism in the application logic responsible for data ingestion into the platform. Moreover, an AODB-centric design functionally decomposes the data platform into different actors. State is encapsulated within each actor, and can only be communicated by asynchronous messages. As such, actors provide a mechanism for isolation of different functions and data, enabling efficient support for multi-tenancy.

Actor modularity in AODBs supports representation and sharing of heterogeneous data. Actors are the unit of modularity in an AODB. By encapsulating state and supporting specification of user-defined APIs, actors abstract heterogeneous data representations. Moreover, arbitrary data transformations can be coded in actor methods, enabling asynchronous exchange of data across heterogeneous actors. As such, actors offer an attractive model to capture heterogeneity of data formats and representations originating at multiple IoT devices.

AODBs employ multiple actor types and concurrent execution among actors to achieve scalability. The support for multiple actor types enables the representation of different kinds of entities in the IoT data platform. When a new entity type is added to the system, it is represented by a new actor type added to the data platform. AODBs thus support a gradual extension of the platform through new actors and actor types with minimal impact on existing components. The use of actors makes scaling out easier, since new actors can be deployed over additional hardware components to avoid violation of performance constraints. The resulting concurrent and distributed execution facilitates efficient use of computational resources to bolster scalability.

Parallelism across actors in AODBs allows for processing of massive amounts of concurrently generated data. By identifying tasks and associated logic among entities, we can model entities as independent actors, so that they can perform tasks concurrently. When independent tasks are then run in parallel across actors deployed on separate hardware components, data platform performance can be improved. Since sensors are naturally modeled as different actors, parallel execution can be leveraged in processing data from a large number of data streams.

Encapsulation and modularity in AODBs support data protection and access control. As data in one actor are invisible to others, access permissions can be checked when data are exchanged by asynchronous messaging [40]. In other words, data are protected inside an actor, and mechanisms for access control can ensure data are only shared with authorized users.

4 MODELING THE CASE STUDIES WITH ACTOR-ORIENTED DATABASES

AODBs provide scalable data processing, management, and storage over a set of application-defined actors [17]. However, to the best of our knowledge, there is scant guidance on how to model applications to reap the benefits of AODBs. In this section, we fill this gap by an in-depth modeling discussion of the two IoT case studies described in Section 2. We contribute to the construction of IoT data platforms in two ways: (1) we provide guidelines for modeling IoT data platforms with AODBs; and (2) we explain how an actor model affects the system both in design and implementation.

It is believed that application modeling helps to preserve and reuse information in other projects, as well as facilitates the automated generation of a system from models [51]. To support the latter aim, we leverage UML notation [53] to create models of actors, their encapsulated state and their operations. These data-centric models harken back to conceptual modeling approaches in databases, enabling both specifications of data requirements and, in the future, code generation for AODB platforms. To support the former aim, we focus on documenting database actors and their asynchronous interactions. In addition to database actors, the classic architecture of an IoT data platform contains a stateless tier that mediates the interaction with users or devices. The analysis of this tier is outside the scope of our work; we abstract its functionality as stateless actors operating as proxies and omit this tier from our models.

In the presented models, we represent the minimal necessary information to emphasize techniques for actor-oriented database design. Application details that would make the presentation unnecessarily complex are thus omitted, and simplifications are made where appropriate. In the sections that follow, we identify each core modeling question encountered in the two IoT data platform case studies and discuss lessons learned.

4.1 How can Actors be Identified?

A variety of entities exist in any system, and these entities either perform or collaborate to achieve different tasks. Moreover, these entities have different life cycles, distinct types, and varied needs regarding heavy computation or communication [14]. To take advantage of the actor programming model as well as achieve high availability and performance, it is an essential question to decide which entities or entity sets should be modeled as actors.

To appreciate a concrete example of this challenge, consider the beef cattle system introduced in Section 2, where many entities perform different tasks to satisfy multiple requirements. For instance, a collar sensor that is bound to a cow continuously collects real-time geo-data for this cow and sends it to the data platform. A historical trajectory for each individual cow is thus updated based on this sensor data. Besides, additional information may be needed, such as the cow's identifier or health-related data. In this sense, we can observe both collar and cow as separate entities engaged in cooperation to provide real-time and historical geo-information to farmers and slaughterhouses. The question is whether these two entities, the collar sensor, and the cow, should be one or two independent actors.

Actors comprise a model of computation for concurrency and distribution [1]. So not only should actors encapsulate state, as it is the case with non-actor objects, but they should also abstract concurrent tasks that need to be processed by the system. In our experience, we found it useful to answer the following questions when attempting to identify actors: (a) What services are provided by the system being modeled? (b) Who should provide these services and who are their users? (c) What is the output and input of every single task performed and can these tasks be executed concurrently?

Take our beef cattle tracking and tracing system as an example. Typically one actor is designed to carry out one specific real-world task with associated logic, such as slaughter or distribute. Different actors then capture simultaneous tasks. For instance, farmers would like to obtain information on cows and manage the herds that they own. Slaughterhouses would like to obtain information about cows that will be slaughtered and record how these cows get transformed into meat cuts. Farmers and slaughterhouses can thus be conceptualized as users of cow information services, which a cow ought to provide. Moreover, the interactions between farmers, slaughterhouses, and cows are concurrently executed by independent entities in the real world. In particular, farmers manage cows and their respective information, and slaughterhouses slaughter cows and record their transformation into meat cuts. Cows are associated with their sensor data, which are continuously updated by their collars. As such, we model farmers, slaughterhouses, and cows as independent actor types. Since each collar is bound to a cow, we encapsulate this sensor information inside cow actors.

Figure 3 shows the data platform model for the beef cattle system. Every actor encapsulates its state and communicates with other actors via asynchronous messages. Therefore, simple accesses to data in the state of an actor are rendered as asynchronous communication events across actors. As we can see from the figure, we model one cow as a *Cow* actor. A Cow actor has an aggregation relationship with many collar sensor readings indicating the GPS locations of the cow, and each such sensor reading is bound to exactly one cow. In other words, we use aggregation relationships to indicate that the objects of a non-actor class are encapsulated in the referred actors. Since cows are modeled as actors, real-time locations are reported to Cow actors, which serve this information to all interested readers along with other associated cow state data such as the cow identifier.

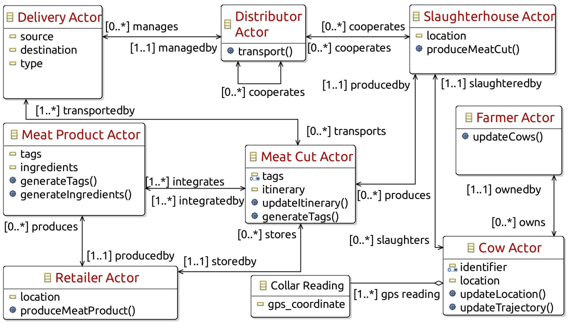

Figure 3: Actor Model of Beef Cattle Tracking and Tracing Data Platform.

We model one farmer or several farmers who work together (e.g., a cooperative) as one single *Farmer* actor because the state of this farmer or these farmers is organized as a unit.[1] One cow is owned by one farm unit, but one farm unit can own many cows. The Farmer actor can read the properties of any Cow actor that is associated with it through message passing. If such messages are exchanged under a security model with authentication, then we can enforce that cow information is only visible to its owner farmer tenant or properly authorized slaughterhouse.

A physical slaughterhouse is modeled as a *Slaughterhouse* actor. A cow can only be slaughtered once in exactly one slaughterhouse, but a slaughterhouse is responsible for slaughtering many cows. This constraint is reflected in the association between Cow and Slaughterhouse actors, and as above a Slaughterhouse actor can read data from any Cow actor via asynchronous messaging. The Slaughterhouse actor processes such data to derive *Meat Cut* actors, which represent units of beef to be distributed as a whole.

A Meat Cut actor processes updates to its itinerary property generated by *Delivery* actors as meat cuts are transported. In our model, a *Distributor* actor manages multiple Delivery actors, which themselves manage a transportation process with different source and destination locations. For example, a logistics company is modeled as a Distributor actor, and transportation processes in this company are modeled as various Delivery actors managed by the Distributor actor. A Delivery actor tracks a meat cut delivery from a source to a destination location using a given vehicle at a well-defined time. A meat cut can be delivered many times by one or several distributors during the whole itinerary, and a distributor is responsible for delivering many meat cuts.

We model the final destination of a meat cut to be a retailer, e.g., a supermarket chain, whose information is managed by a *Retailer* actor. Retailer actors can create *Meat Product* actors by disaggregating or combining meat cuts. Thus, Meat Product actors have a many-to-many association with Meat Cut actors.

Based on the above modeling process, we can summarize a general **principle of how to identify actors**:

> Typically, one actor is designed to carry out one specific real-world task with associated logic. Different actors capture different simultaneous tasks.

[1] Notice that this unit can be broken down into smaller units at will, depending on the focus of an application – e.g., individual farmers unite to provide beef, but a cooperative handles each farmer individually.

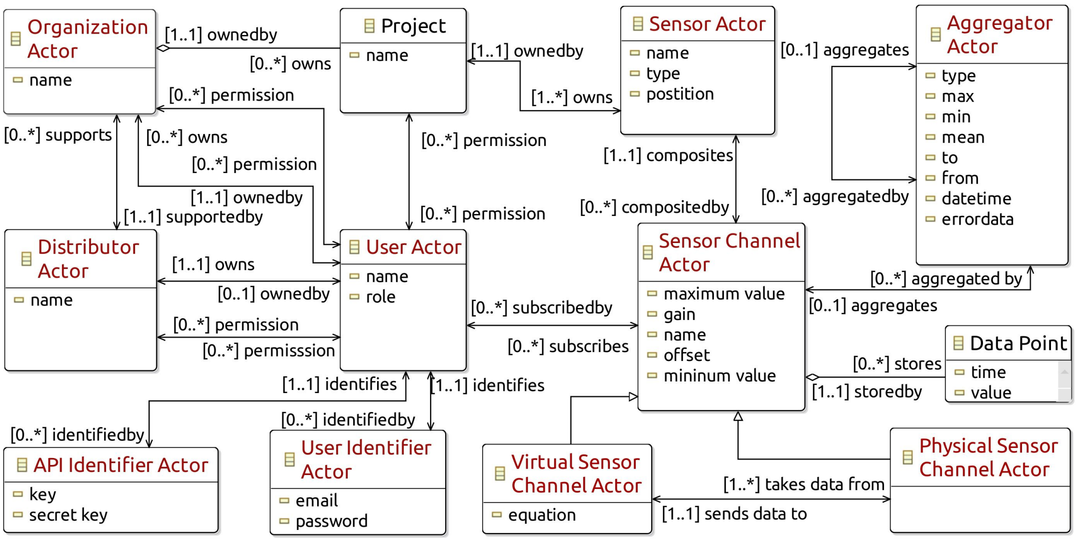

Figure 4: Actor Model of Structural Health Monitoring Data Platform.

4.2 What should the Granularity of Actor State be?

In an AODB, we allocate different tasks into separate actors, as this organization can help increase concurrency. However, if a single actor concentrates too much state or too much of the application logic, i.e., if an actor is *coarse-grained*, then it becomes increasingly difficult to reap the benefits of concurrency stemming from the application of the actor model. At the other extreme, since actors do not share state and communicate only through asynchronous messaging, an excessively *fine-grained* actor design can introduce unnecessary overheads in state manipulation as well as increased communication overhead. Moreover, a fine-grained design may cause the actors in the system to explode in number, e.g., one actor per data item or record in the system, which can challenge efficiency in an AODB platform. So deciding the granularity of actors is an essential problem when modeling any application with actor-oriented databases.

Previously, we have formulated a principle to identify actors out of the entities in an application scenario. However, we should balance this principle against the potential effect of actor granularity on application performance. In particular, we wish to keep concurrency high, but at the same time avoid unnecessary overheads and reduce the complexity of application modeling. To balance these goals, our experience has been that it is natural to make actors more fine-grained when they represent *active entities* for which detailed tracking is required by the application.

Figure 4 presents the data platform model for the structural health monitoring system. We observed during modeling that organization entities own project entities representing different constructions and that each such construction project is associated with some installed sensors. Note that only organizations are active, as they initiate and manage construction activities, while projects are passive structural schemes used by organizations. As such, we create *Organization* actors that encapsulate project information, as displayed by an aggregation relationship with a non-actor Project class, instead of utilizing separate actors. This modeling decision minimizes message exchange when there is no clear advantage in having the two entities run concurrently.

This notwithstanding, sensors are themselves active entities in that they may be relocated, leading to change of position, and also may generate multiple data streams originating from different physical sensor channels (e.g., if we consider a regular smartphone as a sensor, then the accelerometer and microphone would be sensor channels). Moreover, messaging is minimal between sensors and sensor channels, as data streams arriving at the platform can be disaggregated by proxies directly by sensor channel instead of being relayed through sensor entities. As such, we model separate *Sensor* and *Sensor Channel* actors. Sensor Channel actors hold a window of data points originating in the respective data stream. The data points are captured as non-actor objects since these entities are not active.

To help structure information about data points, additional actors are included. First, Sensor Channel is specialized into *Physical Sensor Channel* and *Virtual Sensor Channel* actor classes. Whereas the former represents a channel in a physical sensor, the latter represents a computation over potentially multiple physical channels (e.g., in our smartphone example, an equation merging the data from accelerometer and microphone sensor channels). While a virtual sensor channel provides data at the finest level of detail, it is necessary to provide statistical aggregates for online queries posed by data analysts at various levels of detail (e.g., per hour, day, or month). Since there can be parallelism in computing these aggregations across levels of detail (e.g., hourly aggregates serving as input to daily aggregates), it is useful to conceptualize them as active entities. We thus introduce *Aggregator* actors in the model.

Based on the above modeling process in the context of our case studies, we summarize a general **principle of how to decide on actor granularity**:

> An actor should represent the functionality of one active entity for which detailed tracking is required.

4.3 What is the Trade-Off between Employing Actors or Non-actor Objects for Frequently Accessed Entities?

We discussed the issue of actor granularity, which may result in decisions where entities from the domain are modeled as actors or alternatively as non-actor objects. The modeling principle for actor granularity calls our attention to active entities. By contrast, there are a number of entities that store data but do not proactively perform tasks. We call them *inanimate entities*, and they are exemplified in the beef cattle tracking and tracing case study by meat cuts and meat products. In Figure 3, we model these inanimate entities as actors. However, these actors only encapsulate state and manage corresponding queries and updates originating from active entities such as slaughterhouse, distributor or retailer, e.g., when meat cuts and products are created or transported. As such, a natural question is whether these inanimate entities could have been modeled as non-actor objects instead of actors.

For example, suppose a distributor wishes to obtain information about a meat cut that it transports. The corresponding Distributor actor would have to send a message to the respective Meat Cut actor to fetch this information. Furthermore, when a meat cut is transported, the Meat Cut actor has to communicate with a number of other source or destination actors, such as Slaughterhouse, Distributor, or Retailer actors. As such, a Meat Cut actor frequently interacts with other actors in the system. Since all information on meat cuts needs to be exchanged across actors through asynchronous messaging, casting meat cuts as actors can generate a considerable communication overhead.

To explore this question, we have created an alternative model for the beef cattle tracking and tracing case study (cf. Figure 5). In this alternative model, we capture inanimate, but frequently updated entities, such as meat cuts and meat products, as non-actor objects instead of actors. Actors are marked in red in Figures 3, 4 and 5, while the non-actor objects are marked in black. The non-actor objects represent a state and thus cannot exist in an AODB independently of some actor. To capture state mutation as meat cuts and products move across the supply chain, we create object versions that are always associated with a responsible actor at every stage. Consider how a meat cut is transferred from a slaughterhouse to a distributor. The meat cut is the same real-world entity, but the slaughterhouse and distributor may identify the meat cut differently. Upon transfer, the object representing the meat cut will be copied from the Slaughterhouse actor to the Distributor actor, where this new object version can be updated. Since each actor keeps a separate object version of the meat cut throughout the supply chain, communication to obtain meat cut information is obviated. All the actor logic that reads this information can now access the encapsulated entities in the respective actor state. For frequently accessed entities, this reduction in communication may pay off with respect to the overhead of copying non-actor objects. Furthermore, potentially more concurrency can be exploited in reading local object versions across several actors independently. However, some degree of data redundancy may be introduced in the model.

Based on the above modeling process, we can summarize a general **principle of when to model frequently accessed entities as non-actor objects instead of actors**:

> Frequently accessed entities can be modeled as actors or non-actor objects, and the latter representation should be preferred when reductions in communication overheads and gains from concurrency offset the disadvantages of copying overhead and data redundancy.

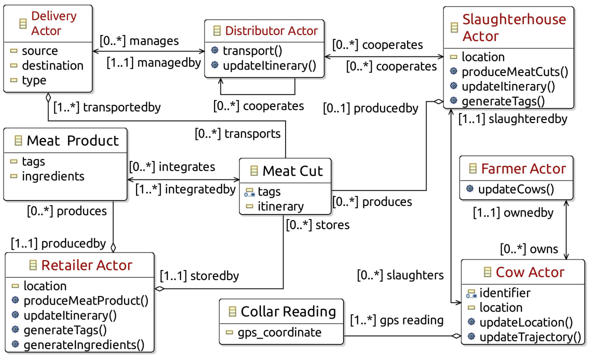

Figure 5: Alternative Actor Model of Beef Cattle Tracking and Tracing Data Platform.

4.4 How can Relationship Constraints be Enforced across Actors?

Because actors encapsulate state and only communicate through asynchronous messaging, relationships between actors are conceptually distributed. For instance, in the model of Figure 3, a farmer may own many cows, but a cow belongs to at most one farmer. In a typical implementation, each direction of this relationship would be represented as properties in Cow and Farmer actors. When performing updates to this relationship, we need to update both sides and make sure these two properties in different actors remain consistent. In particular, when a farmer sells a cow, the Cow actor should have its *ownership* relationship changed to the next owner, and the properties in the two affected Farmer actors ought to reflect that only one farmer retains the ownership of that cow. Since communication between actors is asynchronous, it is a challenge to keep consistency across actors in the presence of updates.

The consistency problem can be addressed by a transaction facility in the AODB, when available, or alternatively by a workflow that ensures that all actors in a relationship change are eventually updated to a consistent state. These options are similar in spirit to the proposal for indexing support in AODBs [17]. Since some actor systems, such as Akka, no longer support transactions [52], and update workflows operate under relaxed consistency, a final alternative is to keep all data related to a relationship, or more generally constraint, encapsulated in a single actor. This discussion leads us to our final **principle of how to enforce constraints when using actors**:

> Employ transactions to update data across actors consistently; however, in the absence of transactions, keep data related to a constraint in a single actor or design a multi-actor workflow for updates.

5 IMPLEMENTATION

In this section, we discuss the implementation of the IoT data platform for the first case study of Section 2 with an AODB. We choose the Structural Health Monitoring Data Platform (SHMDP) since the resulting implementation has been transitioned to the company SenMoS. However, the lessons learned and discussion extend more broadly to the applicability of AODBs to IoT data platforms in other domains, e.g., in beef cattle supply chains, among others.

Choice of AODB. Our implementation of the structural health monitoring data platform was based on the model of Figure 4 [18].

The first implementation challenge to be overcome was to find an appropriate platform supporting actor and non-actor object constructs, as well as the AODB approach. The vision for AODBs [17] was proposed in the context of the Orleans project [16], and we thus elect this actor runtime for the SHMDP. Orleans has also been used successfully in the context of other scalable applications [38], and can thus support real-world deployments. Unlike many other actor programming languages or frameworks such as Erlang [29] or Akka [3], Orleans employs the concept of *virtual actors*, i.e., named actors that are logically in perpetual existence. The Orleans actor runtime automatically creates activations of these virtual actors for processing whenever functions are asynchronously invoked on them, and eliminates activations when there is pressure on resources. As such, virtual actors simplify actor lifecycle management for an application built on Orleans.

In addition to virtual actors, Orleans provides an explicit storage model for actor state. In particular, actors run in a stateful middle-tier that can be conceptualized as an in-memory cache of actor state enriched with application code expressed as actor functions. Whenever persistence of actor state is required, a cloud storage system is employed by Orleans. The concrete storage system is specified through annotations in actor code. To meet the vision for AODBs, additional features are currently being implemented in Orleans to close the gap between actor runtime and DBMS functionality, e.g., indexing [17] and ACID transactions across actors [27].

Data Platform Architecture. A second implementation challenge was to architect an IoT data platform based on AODBs that fulfills all of the non-functional requirements of Section 2. Ideally, an AODB should handle online data ingestion and querying as well as analyses of historical data. However, as pointed out in Section 2, declarative querying functionality is still incomplete in AODBs currently [17]. Thus, we identify three core components for the SHMDP: actor runtime, cloud storage system, and analytical database system. The actor runtime was implemented by Orleans and provides the virtual actor abstraction. It also keeps any necessary in-memory data structures for online data processing and analysis as expressed in the model of Figure 4. The storage system provides durability of actor state, and allows large amounts of historical data to be archived. A key-value database system with efficient data ingestion [36] is useful for this purpose. Finally, data recorded in the storage system can be exported into a classic star schema implemented in the analytical database [34]. The latter component is targeted at analytical queries over historical data, and its description is outside the scope of this paper. The former two components comprised the online data ingestion, processing, and analysis functions of the SHMDP.

Support for Non-Functional Requirements. The AODB architecture supports the non-functional requirements listed in Section 2 as follows:

(1) **Data ingestion from endpoints**. Data from different endpoints was managed by distinct actors in Orleans, and recorded in the cloud storage system for durability.
(2) **Multi-tenancy**. Modularity, data encapsulation, and asynchronous communication were provided by virtual actors in Orleans, allowing isolation of functions and data sensitive to different users.
(3) **Support for heterogeneous data**. Orleans virtual actors support a number of data types and structures, e.g., representing simple alerts or real-time derived data for virtual sensors. In addition, Orleans was used to query time ranges of raw data, and to build aggregates for low latency requests over time periods. The problem of using Orleans for these functionalities was that declarative queries cannot access data across actors, and thus needed to be decomposed by the developer.
(4) **Cloud-based deployment**. Orleans was built to scale out on servers, and extend over multiple geographical locations. It is, moreover, open-source and designed with cloud deployment as a primary target.
(5) **Scalable data platform**. Modularization allows scalability in the number of actors, thus easily enabling the addition of more endpoints or users to the data platform.
(6) **High efficiency**. All processing in virtual actors occurs in-memory. Orleans employs multi-core and multi-server architectures to execute application logic in different actors in parallel.
(7) **Access control and data protection**. Authentication and access control were implemented at the application level by building on actor modularity features.

Virtual actor durability and deployment. Further implementation challenges arise from ensuring that the IoT data platform can effectively ingest and process the large number of concurrent update streams originating from devices. Two issues may impact performance substantially: enforcing durability and deploying actors over multiple machines in a cloud infrastructure.

Orleans virtual actors are called grains, and managed automatically by the Orleans runtime. When a grain has work to do, the grain is activated; when the grain has been standing idle for too long, the grain's resources are reclaimed by the system, removing it from memory. To provide durability, grains in Orleans may have a state storage class. This class defines all variables the developer wishes to store persistently. The developer can force the current state to persistent storage by invoking the WriteStateAsync grain method or configure the grain class to store state persistently when Orleans deactivates a grain. Whenever the Orleans runtime re-activates a grain, the runtime retrieves by default the latest grain state from cloud storage, if available. As such, Orleans lets the developer decide when state is written to persistent state storage.

In the SHMDP, durability requirements may vary depending on the task being implemented. Certain tasks require that the state of actors be immediately made durable, e.g., for creating structural entities such as organizations, sensors, projects, and sensor channels. Other tasks, such as gathering sensor data, can collect a window of updates before forcing them to storage. For example, in the Great Belt Bridge [50], the structural health monitoring project consisted of more than 200 sensor channels, with a typical requirement for live data being a reporting rate of one packet of readings per sensor per second. So if we wrote state to persistent storage after each request, we would need 200 write requests every second to the cloud storage system.

Activated grains in Orleans get distributed across a set of silos, where each silo is typically deployed in a server in a cluster of machines. The distribution of grain activations to silos is by default random, which is adequate for most use cases since it will spread load. However, this actor deployment can increase the cost of communication when certain actors interact frequently. Orleans suggests using prefer-local activation in these cases. For our data platform, we have had to change the activation placement strategy away from random placement for our sensor channels and aggregators. The prefer-local placement in these instances minimizes the need to perform remote procedure calls when processing incoming requests.

6 EXPERIMENTAL EVALUATION

Our goal in the experiments is to assess if the AODB-based implementation of our model from Figure 4 yields an IoT data platform that can scale in the number of sensors simulated and at the same time support low-latency online query functionality. In the following, we present our setup and the obtained results.

6.1 Setup

Benchmarking Tool. To stress-test the SHMDP, we created a command line tool in .NET that uses the Orleans framework client directly. This tool simulates data requests from sensors and users in order to generate variable load for the data platform. Sensors are simulated by tasks that each call a sensor grain and insert 10 data points. This procedure is repeated each second if all sensors have finished their calls, so as to adhere to the behavior expected in the real scenario based on our experience.

Even though we simulate sensors for experimentation with the benchmarking tool above, we envision that ingestion of sensor data points will be based on a REST interface in a production deployment. This way, sensors can send HTTP calls to the data platform. As part of data ingestion, message queues can be employed to accommodate for bursty behavior in sensor measurements [6]. To limit the scope of our evaluation, however, we focus on stressing only the virtual actor implementation of the IoT data platform, and not other layers related to communication with sensor devices.

The benchmarking tool stores data from each request sent to the data platform in a log. Each log entry includes the latency for the request, which request was sent (data insertion, live user data, or user data request), the sampling rate, and a timestamp. With this information, we can derive detailed throughput and latency statistics for the experiments.

Summary of Software. We needed the execution of several components for the experiments. The first one was the *Orleans silo*, typically with one instance deployed per server, where virtual actors are activated and run all application logic. We also employed *Amazon RDS* [12] for Orleans system storage, which keeps track of silo instances, reminders, and general system state. *Amazon DynamoDB* [9] was used for Orleans grain state storage. Besides, the C# benchmarking tool described above is invoked to generate load to silos.

Cloud Service and Deployment. To characterize the SHMDP's data ingestion and processing capabilities, we set up our benchmarking environment on Amazon AWS [8], employing the Amazon DynamoDB and RDS services [9, 12] as stated above and EC2 on-demand instances [10] for all remaining components. Given our budget, two types of instances were employed: T2 for low cost and burst performance features as well as M5 for more stable performance. All instances were running Windows Server 2012 R2 and Orleans 1.5.0. The configuration, unless otherwise mentioned, was designed to simulate a possible future production deployment of the data platform based on our previous experience with the project for the Great Belt Bridge [50]: m5.xlarge instances were employed for the Orleans silos, RDS db.t2.small for Orleans system storage, DynamoDB with 200 writes and 200 reads per second for Orleans grain storage, and an m5.2xlarge instance for the benchmarking tool.

Environment Configuration. For the experiments, we simulate sensors with two sensor channels each; every tenth sensor has a virtual sensor channel that is a summation of the two other sensor channels on the corresponding sensor. The latter choice reflects that only a subset of sensor data require additional processing to create a derived virtual sensor stream, which is close to the real life scenario from the Great Belt Bridge. We populated our actor-oriented database with synthetic data for users, organizations, projects, sensors, and sensor channels simulating a realistic scenario. For every 100 sensors, a new organization was constructed with a single user and a single project. Following the sensor configuration, these 100 sensors represent 210 sensor channels in total, out of which 200 are physical sensor channels and 10 are virtual sensor channels. This structure was used for all experiments, so that we can calculate exactly how many organizations, projects, users, and sensor channels are created given a number of sensors. Employing 100 sensors with 210 sensor channels in total is a configuration similar in size to the one in our previous experience with the Great Belt Bridge.

To achieve our experimental goal related to low latency queries, the upload of data points to the grain state storage has been configured to only happen when the Orleans silo service is shut down. This configuration ensures that we are not benchmarking DynamoDB storage, but rather the execution of in-memory actors. When using the system in production, the grains have to be configured to store data points to grain storage at an acceptable rate as explained in the considerations for durability in Section 5.

Load was offered to the SHMDP by sending requests with 20 data points for each sensor currently being simulated (i.e., 10 data points were generated for each physical sensor channel in each sensor). The requests were sent at a rate of 1 request per second. This frequency simulates sensors sampling data at 10 Hz, as specified in the Great Belt Bridge project for most sensors. As an example, consider that we wish to simulate 500 sensors: this number of sensors would correspond to 1,000 physical sensor channels and 50 virtual sensor channels. Thus, the resulting load would be of 500 requests per second being used to transmit 10,000 data points per second, and leading to the calculation of 500 virtual data points each second.

For each experiment, Figures 6, 7, 8 and 9 present the results. A single point on the figures aggregates 10 minutes of the whole service configuration running. The data was split into windows of 1 minute, and the first minute was removed to make sure the platform had started up correctly before measurement. In addition, the last minute was removed to ensure that only whole minutes were used. The average latency or throughput was then calculated as a measurement, and depicted along with standard deviation as error bars where appropriate.

6.2 Experimental Results

How many sensor readings can the SHMDP ingest using a single cloud server? In our first experiment, we aimed at establishing a relationship between the number of simulated sensors and the hardware utilization at the data platform, so that we can create a baseline load for the other experiments. In particular for these measurements, we employed the smallest VM size in the M5 series, the m5.large instance, and observed when the instance cannot process any more data insertion requests. We have chosen the smallest server size so that the experiment can be used for both scale up and scale out baselines.

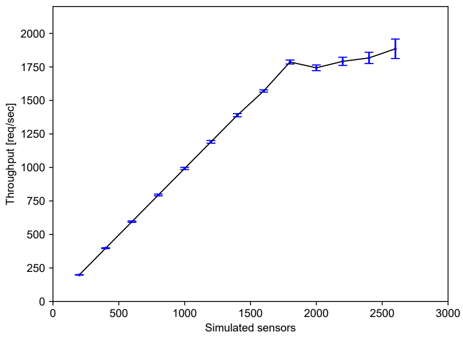

Figure 6: Single-server throughput experiment.

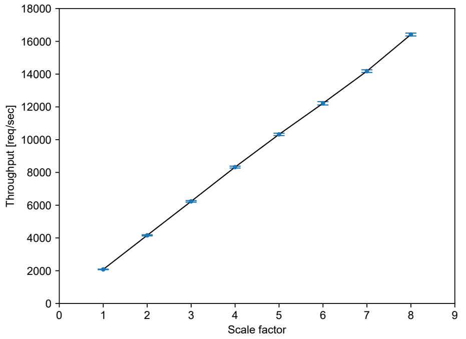

Figure 7: Scale out experiment over multiple servers.

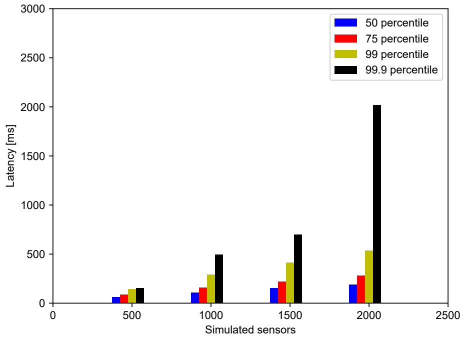

Figure 8: Latency percentiles for raw sensor channel data point time range requests.

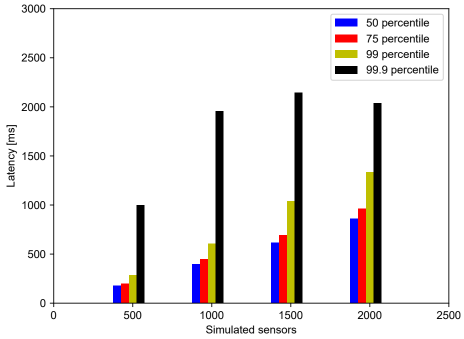

Figure 9: Latency percentile for organization live data requests.

Figure 6 shows the results from this single-server throughput experiment. Because each simulated sensor in the experiment was configured to send one request every second, note that the SHMDP deployment was processing all requests as long as the throughput is equal to the amount of simulated sensors. We observe that the ratio between simulated sensors and throughput is close to one until the number of simulated sensors reaches 2,000. At that point, throughput ceases to increase even if more load is offered. By monitoring the VM instance when performing the experiment, we have remarked that CPU Usage in Windows Task Manager was at 100% when the number of simulated sensors was above 1,800.

Does the SHMDP scale simultaneously on the number of sensors and servers? Our second aim was to verify whether the data platform can scale out in the number of data requests that it can ingest from simulated sensors by utilizing the computing power of more servers. To simulate a production environment, we employed larger m5.xlarge VMs as described in the experimental setup. From our single-server throughput experiments, we can estimate a baseline load to be offered per server. Based on this baseline, we can proportionally scale the load, number of servers, and organization structure in the experiment.

To estimate baseline load, we note that in a production environment, we wish to leave some CPU resources for user interaction. We chose to leave roughly 20% utilization for handling user online query requests and creating statistical aggregates. From the single-server throughput experiment of Figure 6, we know that roughly 1,800 requests per second can be processed by a m5.large instance. By removing 20% and rounding to the nearest 100 requests per second, we obtain 1,400 requests per second. Now, we can scale that number by the difference in computing power between the m5.large and m5.xlarge instances, which is estimated by their EC2 Compute Unit (ECU) values to be of a factor 1.5x. So the baseline for a single server corresponds to the load offered by 2,100 simulated sensors. This configuration is employed for a scale factor of one. As the scale factor is increased, we proportionally increase the number of simulated sensors and the number of servers used for Orleans silos.

Figure 7 shows that the throughput sustained by the data platform scales close to linearly with the scale factor. To illustrate this observation, consider that at a scale factor of five, we have five server instances and 10,500 simulated sensors. We observe as expected a throughput above 10,000 requests per second. Similarly, for a scale factor of eight, we have eight server instances and 16,800 simulated sensors, and a throughput above 16,000 requests per second is observed.

The results indicate that the data platform can potentially scale out even further than the 8 servers used in this experiment, since we did not hit any bottlenecks. We expect that the behavior can be maintained as we add a larger number of servers, since there are no dependencies across organizations and there is enough processing slack left to support eventual online user queries and calculation of statistical aggregates.

Does the SHMDP deliver low latency on online query functions concurrently with data ingestion? We have simplified the previous experiments by removing any user interactions, and made all sensors sample data at 10 Hz sending 1 request each second to the data platform. This scenario is close enough to our experience with a real deployment that we can observe how the data platform scales as we increase the number of sensor insertion requests. However, we still need to show that the 80% utilization rate chosen earlier will indeed leave enough room for the processing of online user queries. Furthermore, we aimed at better characterizing user request latencies under this target utilization level to make informed decisions when creating a production environment in the future.

To simulate user requests to the data platform, we estimated a relationship between simulated sensor requests and user interaction requests. We know from the requirements for the SHMDP that requests for live data as well as raw data kept in the sensor channel actors need to be supported. Requests for live data retrieved the most recent values from all sensor channels of a given organization, while requests for raw data retrieved the time series in a given sensor channel actor in an organization. From actual user interactions observed at the Great Belt Bridge project, we expect these online queries to be generated by at most one person looking at live data for each organization requesting data once every second, and at most one request for raw data a second for each organization. Since a deployment in that project would have around 100 sensors, we thus generate roughly 1% of the requests for live data from all sensor channels in a organization, 1% for raw data, and the remaining 98% as sensor data insertions.

Figures 8 and 9 show that the latency of online query requests increase, as expected, for higher percentiles of the latency distribution. This growth is especially pronounced for 99.9th percentile latency; however, even these extreme tail latencies can be ameliorated if utilization is reduced in the machine by offering load from less sensors. For example, for 500 simulated sensors, 99.9th percentile latency is minimal for raw data requests, and under 1 sec for live data requests. It is expected that latency of user interactions on the website be kept within a few seconds. This requirement can be fulfilled by the data platform even with the targeted 80% utilization load offered by 2,000 simulated sensors and at extreme latency percentiles. Moreover, the latency of raw data requests is often substantially below 0.5 sec, while that of live data requests is often below 1 sec at 2,000 simulated sensors.

7 RELATED WORK AND DISCUSSION

This section discusses research efforts related to our work. To the best of our knowledge, the literature lacks contributions explicitly justifying why and discussing how AODBs meet the challenges of IoT data platforms. However, earlier approaches have explored how to support different aspects of IoT data management employing a variety of data-centric system abstractions.

Approaches based on data stream management systems (DSMSs), in particular, are a commonly used solution in the context of IoT systems [11, 21, 31, 48]. DSMSs are apt at transforming multiple input streams, through a topology of data flow operators, into output streams containing, e.g., alerts and notifications for further processing. One challenge in these systems has been flexibility in responding to dynamically changing conditions as typical in IoT, e.g., through the addition or removal of input sources [49]. Actor-based streaming implementations have been proposed to address these concerns [5, 39], as adaptability is a built-in feature of the actor model [1]. However, a problem with data streaming approaches has been to additionally provide for data storage and online queries [22, 26]. In the context of IoT, AllJoyn Lambda explored a lambda architecture for IoT data storage analytics. Adnan et al. combined streaming and historical data to perform predictions in IoT systems based on machine learning models [2]. In contrast to AODBs, which abstract storage management with virtual actors and storage annotations, these approaches require developers to master complex APIs, often spanning across data stream and database systems. Moreover, while these systems provide for low-latency alerts, online queries are non-trivial to support efficiently. By contrast, an AODB acts as an in-memory, programmable cache where complex analyses can be executed in parallel over the encapsulated state of multiple actors employing user-defined methods.

Another class of solutions explored by previous work is that of cloud-centric actor-based IoT middleware, such as Ptolemy Accessor [42] and Calvin [41]. In these systems, every IoT device is modeled as an actor so that those multiple IoT components can be easily integrated into a potentially complex edge-cloud system. However, these middleware platforms lack integration with data management features that are central to an IoT data platform, such as efficient data storage with support for multi-tenancy and data protection. In addition to middleware, specific IoT applications have also been directly built over actor runtimes [4, 38]. For example, Pegasus is a cloud-based project aimed at gathering data with high-altitude balloons [23]. The system employs the Orleans actor runtime so as to simplify the development process of building a parallel, interactive and dynamic cloud service [15]. In contrast to our work, these previous implementation efforts do not provide any insights on data modeling decisions, nor do they analyze case studies to connect requirements for IoT data platforms with the necessary support from an actor-based solution. Even though there have been explorations of how to employ actors as a modeling construct for cyber-physical systems [25], none of these investigations fully satisfy our data platform requirements, namely storing, managing and processing large-scale data as well as providing for high scalability, real-time computation, data protection, and access control.

In line with the vision of Bernstein et al. [17], we argue that the integration of data management features into actor runtimes can help meet the increasing demand for scalable, low-latency data platforms. Recently, a relational actor programming model has been proposed for in-memory databases and realized in ReactDB [46]. Even though ReactDB shows that the actor model can be used to provide for low latency in databases, we did not consider it as a possible option for our data platform because it is a research prototype and currently not available for production use. Furthermore, in previous work combining actors and databases, there is no systematic review of how to model and structure IoT data platforms, nor discussion of the implementation of such IoT platforms employing an AODB approach. Our work matches the characteristics of an AODB with the requirements and challenges of IoT data platforms, showing how recent research on AODBs can be the basis for a new methodology to model and build IoT data platforms.

8 CONCLUSIONS AND FUTURE WORK

IoT systems require adequate data platforms for handling data storage, management, query and preservation. The modeling and deployment of these platforms remain an open research challenge. In this paper, we presented a generic actor-oriented data platform modeling approach for IoT data platforms, showing how actor-oriented databases can address challenges in the management of IoT data. Our discussion of challenges and their solution was showcased via two distinct case studies, specifically systems for structural health monitoring and beef cattle tracking and tracing. Our contribution covered the detailed modeling of these two real-world case studies and presented the entities and the patterns used to represent their dynamic behavior. This was accompanied by a discussion of modeling challenges, together with our recommendation of technologies and methodologies to address these challenges. As part of this work, we developed a prototype of a structural health monitoring system, which was transitioned to SenMoS. This prototype was validated through experiments demonstrating scalability as more simulated sensors are added as well as low latency in interactive query functions.

We believe that adopting AODBs for IoT systems can help attain the full potential of IoT by extending the reach, scalability, and maintainability of IoT data platforms. As future work, we plan to explore data integration issues in IoT data platforms modeled with the AODB approach, and devise approaches to enforce constraints in AODBs.

ACKNOWLEDGMENTS

Work partially conducted in the context of the Future Cropping partnership [30], supported by Innovation Fund Denmark. Experimental evaluation supported by the AWS Cloud Credits for Research program. In addition, this work was partly supported by the International Network Programme project "Modeling and Developing Actor Database Applications", funded by the Danish Agency for Science and Higher Education (number 7059-000528) and by FAPESP CEPID CCES 13/08293- 7. Additional funding provided by FAPESP project 17/02325-5 and by CNPq-Brazil.

REFERENCES

[1] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

[2] Adnan Akbar, Abdullah Khan, Francois Carrez, and Klaus Moessner. 2017. Predictive analytics for complex IoT data streams. *IEEE Internet of Things Journal* 4, 5 (2017), 1571–1582.

[3] Akka 2018. Akka Documentation. https://doc.akka.io/docs/akka/1.3.1/Akka.pdf.

[4] Akka IoT cases 2018. Akka Documentation, Version 2.5.17, IoT example use case. https://doc.akka.io/docs/akka/2.5/guide/tutorial.html.

[5] Akka Streams 2018. Akka Streams version 2.5.18. https://doc.akka.io/docs/akka/2.5/stream/.

[6] AQStreamProvider 2019. Azure Queue (AQ) Stream Provider. https://dotnet.github.io/orleans/Documentation/streaming/stream_providers.html?q%3Dqueue%23azure-queue-aq-stream-provider%0A.

[7] Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. 2008. Multi-tenant Databases for Software As a Service: Schema-mapping Techniques. In *Proc. ACM International Conference on Management of Data (SIGMOD).* 1195–1206.

[8] AWS 2017. Amazon Web Services. https://aws.amazon.com/.

[9] AWS DynamoDB 2018. Amazon DynamoDB. https://aws.amazon.com/dynamodb/.

[10] AWS EC2 2018. Amazon Web Services EC2 Instances. https://aws.amazon.com/ec2/instance-types/.

[11] AWS IoT 2018. AWS IoT Core. https://aws.amazon.com/iot-core/.

[12] AWS RDS 2018. Amazon Relational Database Service. https://aws.amazon.com/rds/.

[13] Debasis Bandyopadhyay and Jaydip Sen. 2011. Internet of things: Applications and challenges in technology and standardization. *Wireless Personal Communications* 58, 1 (2011), 49–69.

[14] Philip A. Bernstein. 2018. Actor-Oriented Database Systems. In *Proc. IEEE International Conference on Data Engineering (ICDE).* 13–14.

[15] Philip A Bernstein and Sergey Bykov. 2016. Developing cloud services using the orleans virtual actor model. *IEEE Internet Computing* 5 (2016), 71–75.

[16] Philip A Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. 2014. Orleans: Distributed virtual actors for programmability and scalability. *MSR-TR-2014–41* (2014).

[17] Philip A Bernstein, Mohammad Dashti, Tim Kiefer, and David Maier. 2017. Indexing in an Actor-Oriented Database.. In *Proc. Biennial Conference on Innovative Data Systems Research (CIDR).*

[18] Kasper Myrtue Borggren. 2018. *Scalable Structural Health Monitoring Data Platform using Actors as a Database.* Master's thesis. University of Copenhagen, Copenhagen Denmark.

[19] Shawn Bowers and Bertram Ludäscher. 2005. Actor-oriented design of scientific workflows. In *Proc. International Conference on Conceptual Modeling (ER).* Springer, 369–384.

[20] Thomas B Breen. 2009. System and method for updating geo-fencing information on mobile devices. US Patent 7,493,211.

[21] Jean-Paul Calbimonte, Oscar Corcho, and Alasdair JG Gray. 2010. Enabling ontology-based access to streaming data sources. In *Proc. International Semantic Web Conference (ISWC).* 96–111.

[22] Sirish Chandrasekaran and Michael J. Franklin. 2004. Remembrance of Streams Past: Overload-Sensitive Management of Archived Streams. In *Proc. International Conference on Very Large Data Bases (VLDB).* 348–359.

[23] Athima Chansanchai. 2018. Pegasus II mission sends balloon high above Earth and invites you along for an Internet of Things ride. https://news.microsoft.com/features/pegasus-ii-mission-sends-balloon-high-above-earth-and-invites-you-along-for-an-internet-of-things-ride/.

[24] Mario GCA Cimino, Beatrice Lazzerini, Francesco Marcelloni, and Andrea Tomasi. 2005. Cerere: an information system supporting traceability in the food supply chain. In *Proc. IEEE International Conference on E-Commerce Technology Workshops.* 90–98.

[25] Patricia Derler, Edward A. Lee, and Alberto L. Sangiovanni-Vincentelli. 2012. Modeling Cyber-Physical Systems. *Proc. IEEE* 100, 1 (2012), 13–28.

[26] Nihal Dindar, Peter M. Fischer, Merve Soner, and Nesime Tatbul. 2011. Efficiently correlating complex events over live and archived data streams. In *Proc. ACM International Conference on Distributed Event-Based Systems (DEBS).* 243–254.

[27] Tamer Eldeeb and Phil Bernstein. 2016. *Transactions for Distributed Actors in the Cloud.* Technical Report MSR-TR-2016-1001. Microsoft Research. https://www.microsoft.com/en-us/research/publication/transactions-distributed-actors-cloud-2/

[28] Embrapa 2018. Brazilian Agricultural Research Corporation - A Embrapa. https://www.embrapa.br/en/international.

[29] Erlang 2017. Build massively scalable soft real-time systems. https://www.erlang.org/.

[30] Future Cropping 2017. Future Cropping partnership website. https://futurecropping.dk/en/.

[31] Google 2018. Google IoT Core. https://cloud.google.com/iot-core/.

[32] GS1 2018. Global Standards One. https://www.gs1.org/.

[33] IBM 2018. IBM Food Trust: trust and transparency in our food. https://www.ibm.com/blockchain/solutions/food-trust.

[34] Ralph Kimball and Margy Ross. 2013. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling* (3rd ed.). Wiley Publishing.

[35] Andréia Akemi Kondo, Claudia Bauzer Medeiros, Evandro Bacarin, and Edmundo Roberto Mauro Madeira. 2007. Traceability in Food for Supply Chains.. In *Proc. International Conference on Web Information Systems and Technologies (WEBIST).* 121–127.

[36] Chen Luo and Michael J. Carey. 2018. Efficient Data Ingestion and Query Processing for LSM-Based Storage Systems. *CoRR* abs/1808.08896 (2018). arXiv:1808.08896

[37] A Mousavi, M Sarhadi, A Lenk, and S Fawcett. 2002. Tracking and traceability in the meat processing industry: a solution. *British Food Journal* 104, 1 (2002), 7–19.

[38] Orleans 2018. Who Is Using Orleans? http://dotnet.github.io/orleans/Community/Who-Is-Using-Orleans.html.

[39] Orleans Streams 2018. Orleans Streams. https://dotnet.github.io/orleans/Documentation/streaming/index.html.

[40] OrleansGrainCallFilters 2018. Grain Call Filters. https://dotnet.github.io/orleans/Documentation/grains/interceptors.html.

[41] Per Persson and Ola Angelsmark. 2015. Calvin–merging cloud and iot. *Procedia Computer Science* 52 (2015), 210–217.

[42] Ptolemy 2018. The Ptolemy Project: Accessors. https://ptolemy.berkeley.edu/accessors/.

[43] Daniel Diaz Sanchez, R Simon Sherratt, Patricia Arias, Florina Almenarez, and Andres Marin. 2015. Enabling actor model for crowd sensing and IoT. In *Proc.IEEE International Symposium on Consumer Electronics (ISCE).* 1–2.

[44] SEGES 2018. SEGES Landbrug & Fødevarer F.m.b.A. website. https://www.seges.dk/en.

[45] SenMos 2018. SenMoS: your sensor monitoring system. https://senmos.dk/.

[46] Vivek Shah and Marcos Antonio Vaz Salles. 2018. Reactors: A Case for Predictable, Virtualized Actor Database Systems. In *Proc. ACM International Conference on Management of Data (SIGMOD).* 259–274.

[47] Vivek Shah and Marcos Vaz Salles. 2018. Actor-Relational Database Systems: A Manifesto. *CoRR* abs/1707.06507 (2018).

[48] Zhitao Shen, Vikram Kumaran, Michael J Franklin, Sailesh Krishnamurthy, Amit Bhat, Madhu Kumar, Robert Lerche, and Kim Macpherson. 2015. CSA: Streaming Engine for Internet of Things. *IEEE Data Eng. Bull.* 38, 4 (2015), 39–50.

[49] Ayush Singhal, Rakesh Pant, and Pradeep Sinha. 2018. AlertMix: A Big Data platform for multi-source streaming data. *arXiv preprint arXiv:1806.10037* (2018).

[50] Storebælt 2018. Facts and History. https://www.storebaelt.dk/english/bridge.

[51] Toby J Teorey. 1999. *Database modeling & design.* Morgan Kaufmann.

[52] Transactors dropped 2018. Akka Migration Guide 2.3.x to 2.4.x. https://doc.akka.io/docs/akka/2.4/project/migration-guide-2.3.x-2.4.x.html.

[53] UML 2018. OMG Unified Modeling Language (OMG UML) Version 2.5.1. https://www.omg.org/spec/UML/2.5.1/PDF.