algorithms

Article

An Actor-Based Formal Model and Runtime Environment for Resource-Bounded IoT Services [†]

Ahmed Abdelmoamen Ahmed

Department of Computer Science, Prairie View A&M University, Prairie View, TX 77446, USA; amahmed@pvamu.edu

† This paper is an extended version of our paper "A Model for Representing Mobile Distributed Sensing-Based Services" published in the proceedings of the IEEE International Conference on Services Computing (SCC 2018), San Francisco, CA, USA, 2–7 July 2018.

Abstract: With sensors becoming increasingly ubiquitous, there is tremendous potential for Internet of Things (IoT) services that can take advantage of the data collected by these sensors. Although there are a growing number of technologies focused on IoT services, there is relatively limited foundational work on them. This is partly because of the lack of precise understanding, specification, and analysis of such services, and, consequently, there is limited platform support for programming them. In this paper, we present a formal model for understanding and enabling reasoning about distributed IoT services. The paper first studies the key properties of the IoT services profoundly, and then develops an approach for fine-grained resource coordination and control for such services. The resource model identifies the core mechanisms underlying IoT services, informing design and implementation decisions about them if implemented over a middleware or a platform. We took a multi-agent systems approach to represent IoT services, broadly founded in the actors model of concurrency. Actor-based services can be built by composing simpler services. Furthermore, we created a proximity model to represent an appropriate notion of IoT proximity. This model represents the dynamically evolving relationship between the service's sensing and acting capabilities and the environments in which these capabilities are exercised. The paper also presents the design of a runtime environment to support the implementation of IoT services. Key mechanisms required by such services will be implemented in a distributed middleware.

Keywords: Internet of Things (IoT); actors; formal model; runtime environment; resource-bounded; IoT services

check for updates

Citation: Ahmed, A.A. An Actor-Based Formal Model and Runtime Environment for Resource-Bounded IoT Services. *Algorithms* **2022**, *15*, 390. https://doi.org/10.3390/a15110390

Academic Editor: Arun Kumar Sangaiah

Received: 24 September 2022
Accepted: 20 October 2022
Published: 23 October 2022

1. Introduction

An Internet of Things (IoT) system is an engineered system that integrates computational algorithms with physical sensing components and processes. The computational algorithms coordinate and communicate with sensors that monitor cyber and physical indicators, along with actuators that modify the cyber and physical environment [1]. Such smart IoT systems use sensors to connect all distributed intelligence in the environment to gain a more in-depth knowledge of the environment, which enables more accurate actions and tasks. IoT systems can scale to billions of end-devices (i.e., sensors and actuators) connected to gateways, which act as the aggregation points for a group of sensors and actuators to coordinate the connectivity of these devices to each other and to an external network [2].

IoT systems open up an opportunity to offer innovative IoT services that can take advantage of the data collected by different sensors across a wide variety of domains, including health-care, agriculture, entertainment, environmental monitoring, and transportation, etc. [3]. Consider a livestock-monitoring IoT service that utilizes a set of in-situ sensors mounted in a livestock farm to collect data regarding the location, well-being, and health of cattle. This information would help farmers identify diseased animals to be separated from the herd, thereby preventing the spread of diseases. IoT services could also help coordinate rescue efforts following natural disasters, such as the hurricanes Harvey and Irma in 2017 in Texas and Florida, respectively. Emergency agencies could use such services to gain insight into the situation on the ground. IoT services have a communication pattern in which contextual data offered by a number of contributors become the basis for services [4]. Increasingly, sensed data could inform decisions to activate actuators to carry out tasks automatically. A growing number of IoT-based home automation technologies offer good examples of such capability.

Although there is growing body of work on different aspects of IoT services (e.g., [5–7]), research in each of these areas so far has been largely focused on either technology addressing specific challenges [8,9] or particular applications [3,10]. Moreover, the technologies developed in one area do not readily migrate to the other because of the differences in the applications and device specifications involved in each area. Furthermore, developing such services is highly labor-intensive, where significant parts of the code have to be written from scratch. This is, in part, because of the lack of high-level language support for such services. In addition, IoT services often require complex communication and coordination mechanisms, which are not adequately supported by existing ones [11].

This paper proposes a formal model for representing resource-bounded IoT services, broadly founded in the actors model of concurrency [12]. The model allows key principles of IoT services to be rigorously studied. It provides a mechanism for specifying the creation and manipulation of IoT services. The model also identifies core mechanisms underlying IoT services, which inform design and implementation decisions about them if they are implemented over an IoT programming platform. The model also studies some important properties and propositions of IoT services and develops mechanisms for fine-grained resource coordination and control for such services. First, we precisely describe the syntax and operational semantics of the various types of distributed IoT services, and then develop a unifying model to better support them.

We present a runtime environment, which implements the formal model, which is most readily applicable to resource-bounded IoT services. It offers high-level primitives supported using a middleware to implement key mechanisms required by such services, allowing service designers to focus on application-specific concerns. The proposed distributed runtime environment supports the deployment and execution of IoT services. The distributed runtime environment will consist of connected runtime environments, which support the execution of individual service components and manage their communication.

The contributions of this paper are threefold. First, we present the syntax and operational semantics of a formal model for representing resource-bounded IoT services. The syntax defines the model configuration, and the semantics represents the meaning of IoT operations, including communication, computations, and resource acquisition operations. The proposed model can be used to understand and define the complex communication and coordination requirements of a broader class of sensor-based systems. We also present the syntax and operational semantics of a proximity model which enables reasoning about the dynamic and evolving relationship between IoT services. Third, the paper presents the design of a runtime environment which implements the actor-based model with artificial-intelligence (AI) capabilities.

The rest of the paper is organized as follows: Section 2 presents related work. Section 3 presents the properties and definition of an IoT service. Sections 4 and 5 present the operational semantics of the actor-based model for representing IoT services and the proximity model, respectively. Section 6 presents the design of the runtime environment for supporting IoT services. Finally, Section 7 summarizes the results of this work.

2. Related Work

Although there are some existing technologies focused on sensor-based systems [5–7,9], there is relatively limited foundational work. This is in part because of the lack of precise understanding, specification, and analysis of such systems, and consequently, there is limited platform support for programming them. Compared to the traditional design approach [13–15], which focuses mainly on an application-specific point design which allows little compelling heritage, the proposed model enables design reuse at different levels of abstraction, including middleware libraries. It creates a stable and unified runtime environment that can be rapidly extended and customized for a range of different sensors.

The programming required for offering a new IoT service can be significant if carried out from scratch. However, there is an opportunity created by the similarity in the patterns of communication required for IoT services where contextual data offered by a number of contributors become the basis for the application. We defined this pattern of communication in [11] as multi-origin communication. Multi-origin coordination mechanisms can be provided on a platform over which such a class of IoT applications could be implemented relatively easily. We also interpreted and implemented these mechanisms for the domain of crowd-sourced services [4,16] by implementing CSSWare, a middleware which provides domain-specific mechanisms to support initiating new services.

Harjula et al., presented a decentralized IoT edge nanoservice architecture [17] for future gadget-free computing, called nanoEdge. The proposed model enables on-demand service composition based on the hardware and software resources available at the service location. The IoT nodes contribute to nanoservices based on their hardware capacity, storage, security, and privacy services without relying on centralized entities. The nodes with more resources can participate more in service provisioning than those with fewer resources.

The architecture of distributed IoT systems involves a large number of end devices connected to gateways. In many cases, both end devices and gateways operate in resource-constrained concurrent environments with limited memory and power, and require real-time capabilities in some scenarios. To accommodate such requirements, small operating systems, called IoT OSs, are specially designed to manage the device's resources efficiently. Over the last decade, several OSs for IoT applications have emerged, such as TinyOS, FreeRTOS, RIOT and Android Things.

The runtime environment of IoT applications, which runs over the OS, is responsible for providing all necessary services and support for the execution of applications. There are a few related works in the literature which provide such support, such as PatRICIA [14], RapIoT [15], and Eclipse Kura [18]. PatRICIA is a framework for developing and deploying IoT applications on cloud platforms. However, this framework does not enable IoT applications to utilize the edge of the network because of the lack of code distribution mechanisms. RapIoT is a toolkit which supports rapid prototyping of IoT applications by non-expert developers. Unlike PatRICIA and RapIoT, the proposed approach supports the distributed execution of IoT applications over a large number of devices at the edge of the network.

Compared to the traditional design approach [7,9,13,17,19], which focuses mainly on an application-specific point-design which allows little compelling heritage, the proposed approach offers the following salient advantages: (I) It provides a generic model for developing an efficient sensor-data processing system. Basing designs on a standard integration platform architecture with AI libraries of pre-proven components will provide a solid foundation and streamline the development and validation processes. (II) It enables the design reuse at different levels of abstraction, including middleware libraries. It creates a stable and unified framework which can be rapidly extended and customized for a range of different sensors.

3. IoT Service Definition

As illustrated in Figure 1, distributed IoT services perform three things: they sense interesting environments, they perform computation, and they act in some way to serve a purpose. This is not different from a classical view of computations where there is an input, required processing, and an output of the processing. What is, however, very different is the context in which these things happen. The application context is different because IoT applications are fundamentally distributed, requiring complex types of communication.

The execution environment is also different because of the distributed resources required and their characteristics.

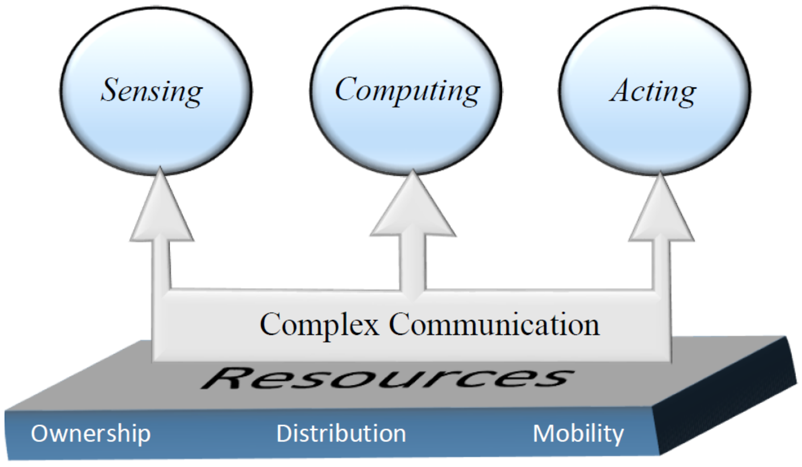


Figure 1. IoT service properties.

3.1. *Communication*

In addition to simple synchronous or asynchronous communication, distributed IoT applications often involve different types of group communications, with both multiple recipients as well as multiple senders. For example, consider a number of sensors autonomously sending their sensing feeds to a server to be used in aggregate form as the basis for a service. What is required to support this is high-level communication primitives, which do not mix functional concerns of services with complex communication concerns.

3.2. *Resources*

The resources required for distributed sensor-based services include processor, memory and network resources, as are typical of most distributed computations; in addition, also required are sensor and possibly actuator resources, where the latter can be more generally defined to include any resources required for producing the desired output. Some of these resources can be used in parallel by different services. For example, different parts of memory can simultaneously hold the states of different services, different cores of a multi-core processor can be simultaneously serving different services. However, all of these resources can be used concurrently by interweaving steps of the different services, in a time-multiplexed manner. In other words, schedulers can be used to periodically schedule the use of resources over a period of time. There are, however, three challenging aspects of resources that one must be contended with when dealing with distributed sensor-based services: ownership, distribution and mobility. We discuss them below, in turn.

3.2.1. Resource Ownership

Some of the most exciting opportunities for large-scale distributed sensor-based services come from the prospect of the bringing together of networked computational devices with embedded sensors, which already exist on people's persons, in their homes, and in social spaces, in the form of smartphones and wearable devices. However, these devices are typically owned by people, who pay to keep them charged and connected, and rely on them. They also hold private data, and serve their owners as their primary purpose. What is required, then, is a potent way for applications to negotiate usage rights for these devices with their owners.

3.2.2. Resource Distribution

Resources needed by IoT services are typically physically distributed in geographic space. This distribution does not just present a connectivity challenge; it actually is the leading enabler of various services that require the sensing of live data in different (or the same) geographic locations to create a coherent view of an interesting world. For example, significant computational resources may need to be located in close network proximity to high-fidelity sensors to pre-process data before being transmitted to a central server. Such requirements present challenges in determining which specific resources—from a variety of available ones—should be utilized for an application at a particular time.

3.2.3. Resource Mobility

The resources involved in distributed IoT services—the computing devices, the sensors, and the actuators acting on behaf of services—are all potentially mobile. Resources could be moving to be in optimal locations to serve a service (such as sensors actively pursuing their sensing goals), but more often, they are mobile because their wider service tasks are secondary to their primary use of serving the device's owner's own needs, such as computational, sensing, or connectivity. A resource that is stationary with respect to the device it is located on, which in turn is also stationary with respect to the owner of the device, could be mobile with respect to the physical surroundings because of the owner's movements.

Similarly, the phenomena or situations of interest to an application may also be mobile. Consider, for instance, a herd of endangered wild animals on the move in a forest, requiring tracking by scientists. What this means is that the observer–observee relationship between IoT services and what they are sensing is highly dynamic. As a result, an IoT service must dynamically evolve the mapping between the processing/sensing/acting tasks which need to be carried out and the physical placement of those tasks in execution environments that can sustain them.

3.3. IoT Service Definition

A service receives input contributions from some contributing source, processes them, and creates output contributions for some clients. We call these contributions *service feeds*. In a system of IoT services, we treat every client and contributor as a service: they are called the client service and the contributing service of the particular IoT service. In other words, the client service could simply be an end user receiving a feed from a service, but not necessarily producing a derivative service for another service; a contributing service, similarly, could be just a sensor without any service contributing to it.

We model an IoT service by a set of *ports* and a set of *agents*. Ports can be one of two types: input or output. A service receives input feeds from contributing services through its input ports and sends output feeds to client services through its output ports. Agents implement the service's logic and convert the feeds arriving from contributing services into the feeds required by client services. Our definition for an IoT service is as follows:

Definition 1 (Service)**.** *An IoT service has one or more input ports, which receive feeds from contributing services, and one or more output ports, which send feeds to client services, and a set of agents, which are responsible for implementing the service's logic.*

From this definition, a service has two types of components: ports and agents. Both ports and agents are active objects. Consequently, a service can be defined as a set of active objects. Input ports are receptionists of data. Therefore, contributing services must know the names of input ports of client services in order to send data messages—through their output ports—to them.

In a system of services, each service could be either a contributing or consuming service, or both at the same time. We consider each client and contributor in the system as a service, as they all fit Definition 1. A client is deemed a service because it has a non-empty finite set of input ports, at least one output port, and a non-empty finite set of agents. Similarly, a contributor, which has at least one input port, a non-empty finite set of output ports, and a non-empty finite set of agents, can also be considered as a service for the same reason. Furthermore, a single sensor can also be considered as a particular case of a contributing service because it has an empty set of input ports, a single output port, and a single agent, which is responsible for sampling sensor data at a particular sampling rate.

In a system of IoT services, the required communication between services is carried out by sending and receiving asynchronous messages. Messages can be of one of two types: control messages or data messages. Control messages (also called *inter-service* messages) are communicated between services for administrative purposes. In contrast, the data messages are used to send service feeds from contributing services to client services. Each service has a dedicated agent called the *service coordinator*, which is responsible for handling control messages between its home service and other services in the system. When a control message is sent to a service, the message is received by its coordinator agent. The rest of the agents implement the service logic and process the received data messages.

Figure 2 illustrates the interaction between services. A rectangle represents a service's boundary. Each service has a set of ports using which it could communicate with other services in the system. The figure also illustrates the components of a service. Ovals are agents serving the service, white circles are input ports, black circles are output ports, and the lines with arrows represent message flows; the service encapsulates a set of agents implementing its logic. These agents are invisible to other services. To interact with a service, both contributing and client services have to be connected to the service's ports: to send messages to the service, contributing services are connected to its input ports; to receive messages from the service, client services are connected to its output ports.

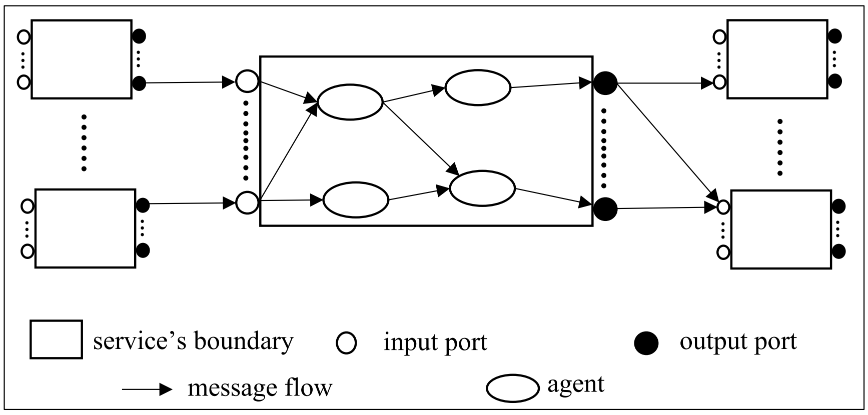


Figure 2. Communication between services.

4. Operational Semantics

This section presents the operational semantics for the proposed model. We define the state of an IoT service as follows:

Definition 2 (Service State)**.** *A service is denoted as s and is written as:*

$$[\![I : \alpha : O]\!]_s \tag{1}$$

where s is the service's unique name, I is a set of input ports of the service, α is a set of agents of the service, and O is a set of output ports of the service.

Here, we are only interested in modeling the interactions between services, so we do not show data messages with local recipients within a service. As agents are modeled as actors, α can be considered as an actor map which maps a finite set of actor addresses to their behaviors. A coordinator actor of a service s, written as $a_s \in \alpha$, receives all messages sent to s. Note that there is a one-to-one mapping between s and a_s (i.e., for each service s, there is exactly one coordinator agent).

Definition 3 (Model Configuration)**.** *The instantaneous snapshot of a system of services is called a service model configuration. This configuration represents the state of a finite set of long-lived IoT services, a finite set of contracts between services which define how services are connected to each other, and a finite set of control messages between services. A contract is negotiated between two services when one of them wants to consume service feeds produced by the other.*

The model configuration can be represented by a 3-tuple:

$$\langle S \mid C \mid M \rangle \tag{2}$$

S is a set of services. C is the set of contracts between the services, where each contract $c \in C$ has the form $(s_1,\ s_2,\ map)$, where s_1 is the name of the first service, s_2 is the name of the second service, and map is a name table which says which output ports of the first service are connected to which input ports of the second service. The connections are represented using $(o\ ;\ i)$ pairs where $o \in O$ is an output port of s_1 and $i \in I$ is an input port of s_2. The contracts involving a service $s \in S$ can be written as $co(s) \subset C$. M is a finite set of control (inter-service) messages in the system which are communicated between services and are handled by coordinator actors of the communicating services.

Our model assumes that there is a special service, called the *directory service*, which has up-to-date information about the capabilities of all contributing IoT services in the system. For example, the service capability of a sensor service is determined by the maximum sampling rate of that sensor. The directory service fits the service definition (`Definition 1`) because it has non-empty finite sets of input ports, output ports and agents. The directory service can be implemented as a federated hierarchy of directory services, which can be distinguished by different metrics such as geographical locations, type of services, etc. Each contributing service in the system must be registered with the directory service in order to participate in serving client services. Note that the names and capabilities of all contributing services registered with the directory service are all part of the directory service's state.

4.1. Service Request

One way to represent the requirements of an IoT service is to represent them as sets of timed data feeds. For each of these feeds, the service needs to secure appropriate sensor data feeds and carry out the required aggregations and customizations for different clients. This representation of a service request gives a client the ability to represent their service requirements without being too rigid. If the client is too rigid in defining their request, the service request is likely to be rejected if the directory service is too busy at these points of time.

A service request is represented by ρ and is defined as follows:

Definition 4 (Service Request)**.** *A service request ρ is a set of sets of timed service data feeds. Each set $\rho_i \in \rho$ has a sufficient number of service feeds for serving that request.*

A service request is represented formally as follows:

$$\rho = \{\rho_1, \ldots, \rho_n\} \ \ , \ \ \rho_i = \langle \zeta_1, \ldots, \zeta_m \rangle \tag{3}$$

where $n \geq 0$ is the number of sets in ρ, $m \geq 0$ is the number of service feeds in ρ_i, and ζ represents a single timed service feed.

Only one set $\rho_{selected}$ out of ρ needs to be served. The selection of this $\rho_{selected}$ can be said to be carried out by a function f as follows:

$$\rho_{selected} = f(\rho) \tag{4}$$

The necessary and sufficient condition for accepting a service request is stated in `Axiom 1`, as follows:

Axiom 1: Accepting a service request. A service request ρ can be accepted in the system if and only if at least one member of ρ, $\rho_{selected} \in \rho$, can be served by a set of contributing services, $S_{\rho_{selected}} \subset S$.

We consider a continuous-time model for representing services. This continuous-time modeling approach provides more flexibility in describing IoT services because of their real-time requirements, represented as sets of temporally defined feeds. These feeds are constructed by aggregating sensor events —such as user activities, change in a geographical location, etc., which occur at particular points in time and lead to a change in the state of interest of an IoT service.

4.2. Transition Rules

We use transition rules to describe the progress of a system of IoT services. Next, we present transition rules of the proposed actor-based model for representing IoT services.

4.2.1. Sending a Service Request

An IoT service is initiated by a client service which sends a service request ρ to the directory service to create a new service. The client's requirements are expressed in that request, ρ. The same service request can also be used to subscribe to an existing service.

The following transition shows how a client service sends a service request to the directory service:

$$\begin{gathered}\langle [\![I:\ [R[send(s_d,(\rho,s))]]_{a_s}\ ,\ \alpha\ :\ O]\!]_s,\ S \mid C \mid M\rangle \to \\ \langle [\![I:\ [R[nil]]_{a_s}\ ,\ \alpha\ :\ O]\!]_s,\ S \mid C \mid \langle s_d \Leftarrow (\rho,s)\rangle,\ M\rangle\end{gathered} \tag{5}$$

where s is a client service, s_d is the directory service, ρ is a service request, a_s is s's coordinator actor, and $send(s_d,(\rho,s))$ sends message (ρ,s) containing the received ρ and the client's name to the directory service s_d. This leads to the creation of message $\langle s_d \Leftarrow (\rho,s)\rangle$ on the right hand side, and actor a_s continues execution.

4.2.2. Search Function

For convenience, we define a function *search* which is used by the directory service to determine the opportunity to serve a new service request ρ by selecting one existing service matching the requirements of ρ, or a set of contributing services which could collectively contribute to serving ρ. The *search* function takes as parameters a service request ρ and the name of the client service s requesting ρ, and returns one of three pairs: (1) $(\{s_m\},c)$, a pair of the name of an existing service s_m which matches ρ's requirements and a new contract c created between s_m and s; (2) $(S_\rho,\emptyset)$, a pair of a set of potential contributing services S_ρ which could collectively contribute to serving ρ based on their capabilities and an empty set $\emptyset$ indicating that no contract is created at this point; or (3) $(\emptyset,\emptyset)$ to indicate that there is no way of serving ρ. The *search* function is defined as follows:

$$search(\rho,s) = (\{s_m\},c) \mid (S_\rho,\emptyset) \mid (\emptyset,\emptyset) \tag{6}$$

4.2.3. Receiving a Service Request

On receiving a service request ρ, the directory service uses the search function to determine the opportunity to serve ρ. This transition is written as:

$$\begin{gathered}\langle [\![I:[R[ready(search)]]_{a_{s_d}},\alpha:O]\!]_{s_d},\ S \mid C \mid \langle s_d \Leftarrow (\rho,s)\rangle, \\ M\rangle \to \langle [\![I:[search(\rho,s)]_{a_{s_d}},\alpha:O]\!]_{s_d},\ S \mid C \mid M\rangle\end{gathered} \tag{7}$$

where s_d is the directory service, a_{s_d} is s_d's coordinator actor, s is the name of the client service which sent ρ, $\langle s_d \Leftarrow (\rho,s)\rangle$ is a service request message sent to s_d, and ρ is the service request. As a result of delivery of this message to actor a_{s_d}, a_{s_d} uses the *search* function to first search for an existing service matching the requirements of ρ, or a set of contributing services which could collectively contribute to serving ρ.

4.2.4. Subscribing to an Existing Service

If a matching service is found, then the directory service tells the client service about this service, and a new contract is signed between the client service and the found service.

If $search(\rho, s)$ evaluates to $(\{s_m\}, c)$, then the transition rule for subscribing to an existing service is as follows:

$$\begin{aligned} &search(\rho, s) \xrightarrow{\lambda}_X (\{s_m\}, c) \Rightarrow \\ &\quad \langle [\![I : [R[search(\rho, s)]]_{a_{s_d}}, \alpha : O]\!]_{s_d},\ S \mid C \mid M \rangle \rightarrow \\ &\quad \langle [\![I : [R[(\{s_m\}, c)]]_{a_{s_d}}, \alpha : O]\!]_{s_d},\ S \mid C' \mid \langle s \Leftarrow s_m \rangle,\ M \rangle \end{aligned} \tag{8}$$

where $X = \{a_{s_d}\}$ is the context in which $search(\rho, s)$ is reduced to $(\{s_m\}, c)$; s_d is the directory service; a_{s_d} is s_d's coordinator actor; ρ is the service request; s is the name of the client service which sent ρ; s_m is the name of an existing service s_m which matches ρ's requirements; c is a new contract created between s_m and s, which has the form $(s_m,\ s,\ (o_{s_m}\ ;\ i_s))$ where $o_{s_m} \in O_{s_m}$ is an output port of s_m and $i_s \in I_s$ is an input port of s; and $C' = C\ \cup\ \{c\}$. This transition happens if and only if $search(\rho, s)$ is evaluated to $(\{s_m\}, c)$. This leads to the creation of message $\langle s \Leftarrow s_m \rangle$ containing s_m which is sent to s, and the creation of a new contract c created between s_m and s on the right hand side.

4.2.5. Creating a New Service

Here, we present the composition of running contributing services to create a new service based on the definition of ρ.

Consider a function *create*: When providing as parameters a service request ρ; a set of contributing services, S_ρ, which could collectively contribute to serving ρ; and the name of the client service s requesting ρ, *create* creates a new service which satisfies ρ's requirements. Then, the *create* function returns the name of the newly created service s', and a set of contracts C_ρ between s' and S_ρ. The *create* function is defined as follows:

$$create(\rho, S_\rho, s) = (\{s'\}, C_\rho, c) \tag{9}$$

If there exist contributing services whose feeds are sufficient for generating ρ, then the service request is accepted. This operation is shown in the following:

$$\begin{aligned} &search(\rho, s) \xrightarrow{\lambda}_X (S_\rho, \emptyset) \Rightarrow \\ &\langle [\![I :\ [R[search(\rho, s)]]_{a_{s_d}}\ ,\ \alpha\ :\ O]\!]_{s_d},\ S \mid C \mid M \rangle \rightarrow \\ &\qquad \langle [\![I :\ [R[(S_\rho, \emptyset)]]_{a_{s_d}}\ ,\ \alpha\ :\ O]\!]_{s_d},\ S \mid C \mid M \rangle \end{aligned} \tag{10}$$

where $X = \{a_{s_d}\}$ is the context in which $search(\rho, s)$ is reduced to $(S_\rho, \emptyset)$, s_d is the directory service, a_{s_d} is s_d's coordinator actor, ρ is the service request, and s is the name of the client service which sent ρ. This transition happens if and only if $search(\rho, s)$ is evaluated to $(S_\rho, \emptyset)$.

The directory service then uses the *create* function to create a new service utilizing the contributing services. The transition rule for creating a new service is as follows:

$$\begin{aligned} &create(\rho, S_\rho, s) \xrightarrow{\lambda}_X (\{s'\}, C_\rho, c) \Rightarrow \\ &\quad \langle [\![I : [R[create(\rho, S_\rho, s)]]_{a_{s_d}}, \alpha : O]\!]_{s_d}, S \mid C \mid M \rangle \rightarrow \\ &\quad \langle [\![I : [R[(\{s'\}, C_\rho, c)]]_{a_{s_d}}, \alpha : O]\!]_{s_d}, S' \mid C' \mid \langle s \Leftarrow s' \rangle, M \rangle \end{aligned} \tag{11}$$

$$\{s'\}\ ,\ C_\rho \text{ and } c \text{ are fresh}$$

where $X = \{a_{s_d}\}$ is the context in which $create(\rho, S_\rho, s)$ is reduced to $(\{s'\}, C_\rho, c)$; s_d is the directory service; a_{s_d} is s_d's coordinator actor; s is the client service which sent the service request ρ; s' is the newly created service producing ρ from C_ρ's contributions; C_ρ is a new set of contracts created between s' and S_ρ; c is a new contract created between s' and s, which has the form $(s',\ s,\ (o_{s'}\ ;\ i_s))$ where $o_{s'} \in O_{s'}$ is an output port of s' and $i_s \in I_s$ is an input port of s; $C' = C\ \cup\ C_\rho\ \cup\ \{c\}$; and $S' = S \cup \{s'\}$. This transition happens if and only if $search(\rho, s)$ is evaluated to $(S_\rho, \emptyset)$. This leads to the creation of message $\langle s \Leftarrow s' \rangle$ containing the name of the new service s', the creation of new sets of contracts C_ρ and $\{c\}$, and the creation of a new service s' on the right hand side.

4.2.6. Rejecting a Service Request

A service request is rejected if none of the existing services match the target service's requirements expressed in ρ, and there do not exist sufficient contributing services for serving ρ (i.e., there are no sufficient contributing services for serving any member of ρ). If these two conditions hold, a request rejection message is sent to the client service which sent ρ.

If $search(\rho, s)$ evaluates $(\emptyset, \emptyset)$, then the transition rule for rejecting a service request is as follows:

$$\begin{array}{l} search(\rho, s) \xrightarrow{\lambda}_X (\emptyset, \emptyset) \\ \quad \langle [\![I:\ [R[search(\rho, s)]]_{a_{s_d}}\ ,\ \alpha\ :\ O]\!]_{s_d},\ S \mid C \mid M \rangle \rightarrow \\ \quad \langle [\![I:\ [R[(\emptyset, \emptyset)]]_{a_{s_d}}\ ,\ \alpha\ :\ O]\!]_{s_d},\ S \mid C \mid \langle s \Leftarrow \emptyset \rangle,\ M \rangle \end{array} \tag{12}$$

where $X = \{a_{s_d}\}$ is the context in which $search(\rho, s)$ is reduced to $(\emptyset, \emptyset)$, s_d is the directory service, a_{s_d} is s_d's coordinator actor, and s is the client service which sent service request ρ. This transition happens if and only if $search(\rho, s)$ is evaluated to $(\emptyset, \emptyset)$. This leads to the creation of message $\langle s \Leftarrow \emptyset \rangle$ which is a rejection message sent to s indicating that there is no way of serving ρ.

5. The Proximity Model

In this section, we study some important properties and propositions of IoT services. We also present a proximity model to represent an appropriate notion of proximity for IoT services. Developing such a model requires more than simply representing geographic locations and specifications of sensors and actuators. Particularly, proximity may be geographical or logical, and is often dependent on the context. For example, consider a camera capable of capturing high-quality images in good lighting, but limited in its range and resolution in dark or foggy conditions. In this paper, we address these challenges by representing a sufficiently rich and context-sensitive notion of proximity.

5.1. Properties and Propositions

The necessary and sufficient condition for serving an IoT service request is stated in `Axiom 2`, as follows:

Axiom 2: Serving a service request. A service request set to $\rho_i \in \rho$ can be served by a set of contributors $C_{\rho_i} \in C$ if and only if each member of ρ_i can be served by a subset of C_{ρ_i}.

The necessary and sufficient condition for serving a client tuple is stated in `Axiom 3`, as follows:

Axiom 3: Serving a client tuple. A client tuple $\zeta \in \rho_i$ can be served by a set of contributors C_ζ if and only if C_ζ has the sensing capabilities to serve the requirements of ζ, and ζ can be accommodated in C_ζ's sensing schedules.

There are two types of deadlines set by the directory service in each contributor's request to serve ρ_i: (i) the offer response's deadline by which a contributor has to respond to the directory service's request; and (ii) the offer acceptance's deadline by which the directory service has to accept the contributor's response to its request.

The following axiom describes the conditional contributor commitment to contribute in serving the service request:

Axiom 4: Conditional contributor commitment. A contributor $c \in C_{\rho_i}$ conditionally commits to serving its part, $Z^c_\rho \in \rho$, if it indicates its capability to serve Z^c_ρ to the directory service, and Z^c_ρ can be accommodated in c's sensing schedule, with the condition that the directory service accepts the assignment of Z^c_ρ to c by a deadline $d_{Z^c_\rho}$.

We assume that the directory service guarantees to send a confirmation message to the contributor service before the deadline $d_{Z^c_{\rho_i}}$ if the directory service has accepted c's offer.

The following axiom describes the unconditional contributor commitment to contribute in serving the service request:

Axiom 5: Unconditional contributor commitment. A contributor c's conditional commitment to serve $Z^c_{\rho_i}$ becomes unconditional when the server accepts c's offer by the deadline $d_{Z^c_{\rho_i}}$.

Theorem 1. *If the directory service accepts a request ρ, it is served.*

Proof. According to `Axiom 1`, if a request ρ is accepted in the system, there must be at least one set $\rho_i \in \rho$ which can be served by the directory service, and the service can schedule the processing time required to serve ρ_i. According to `Axioms 2 and 5`, if ρ_i can be served by the directory service, the directory service must have received unconditional offers from a set of contributors $C_{\rho_i} \in C$ to participate in serving ρ_i. To obtain an unconditional commitment from a contributor $c \in C_{\rho_i}$, according to Axiom 4, c must have checked its sensing schedule and temporarily reserved a time slot for serving its part in ρ_i. c would have sent an offer to the server by the deadline, and this offer would have been accepted by the server by c's deadline. The directory service would have sent a confirmation message to c. Then, c would have, finally, sent a message to the directory service conforming that c has scheduled its part in ρ_i for execution. If all members in C_{ρ_i} have sent a confirmation message to the directory service, this means each member $c \in C_{\rho_i}$ must have scheduled its part, $Z^c_{\rho_i} \in \rho_i$, for execution. In other words, C_{ρ_i} has unconditionally committed to collectively serve a set of client tuples Z_{ρ_i} which covers all members in ρ_i. Therefore, ρ has a commitment to be served by both a set of contributors and the directory service. This means that a request ρ is served if the directory service accepts it, which proves `Theorem 1`. □

There are two types of delay set by the client in this request, which tell the directory service about the following requirements: (i) the freshness of the sensed data; and (ii) the time ranges of receiving service updates from the server.

The following definition describes the compatible delay tolerance of the sensing requests served by a single contributor service.

Definition 5 (Compatible delay tolerance)**.** *The delay tolerances of multiple sensing requests, which are parts of contributor tuples, are compatible if the hosting contributor can opportunistically merge these requests to form one sensing group so that one optimal sampling instance would satisfy them all.*

Axiom 6: Tolerant sensor sampling. If multiple sensing requests have compatible delay tolerances at a contributor c, then c can opportunistically merge these requests to form one sensing group so that one optimal sampling instance would satisfy all of them.

Axiom 6 serves as the foundation for our ShareSens approach [20]. It is used when the optimized sensing option has been selected. ShareSens assumes that requests pick from a limited number of sampling rates available, selected both to increase opportunity for sharing sampled data as well as to enhance the performance of the scheduling mechanism, which implicitly encourages selecting a delay tolerance when submitting a sensing request. The quality of service determines the cost/price of the service in our pricing model. ShareSens provides two options of quality of service for serving a sensing request when dealing with tolerant sensor sampling: (i) the same time lag delays all samples; or (ii) samples are provided within a range, but there is no promise to have an equal time distance between them.

Definition 6 (Contributor service ability). *A contributor c indicates its ability to serve its part in ρ_i, $Z^c_{\rho_i}$, if c has the sensing capabilities to serve the requirements of $Z^c_{\rho_i}$, and the sensing requests in $Z^c_{\rho_i}$ can be served using a mix of the following options:*

1. *Some sensing requests in $Z^c_{\rho_i}$ can join some of the sensing groups already being served at c if the sensing schedule of those groups is not altered after the arrival of the incoming requests;*
2. *Some sensing requests in $Z^c_{\rho_i}$ can be merged with some of the sensing groups already being served at c if they have compatible delay tolerances with those groups so that the existing sensing schedule at c is altered in order to accommodate the incoming requests;*
3. *Some sensing requests in $Z^c_{\rho_i}$ are separately scheduled in a new sensing group.*

Choosing between the above options depends on the complexity of the reasoning process for accommodating the incoming requests. If the reasoning cost of option 2 is high, then it may be more efficient to use option number 3 directly. In other words, there should be a balance between cost effectiveness and the expected benefits. Therefore, optimizing the schedule has to be weighed against the benefits of the optimal schedule, which may depend on the duration of the sensing request.

Axiom 7: Opportunistic merging of sampling requests. An incoming sensing request can partially or completely share the same sampling stream with a sensing group already being served at a contributor *c* if and only if it has a delay tolerance which is compatible with that group.

Definition 7 (Client tuple). *A client tuple ζ represents a single timed service feed, which is formally represented as:*

$$\zeta = \langle Y, s, td, d_\zeta \rangle \tag{13}$$

where Y denotes an expression that is evaluated by the server to obtain the sensing task (i.e., the sensing parameters) which is carried out by the contributors. s represents the client service, td is the delivery time, and d_ζ is the maximum time delay between the sensing time and the delivery time. If the sensed data is a result of aggregating multiple sensing feeds, then d_ζ becomes the maximum time delay between the oldest feed and the delivery time of the aggregate.

Contributor tuples are created using a function f_Ξ as follows:

$$f_\Xi(\rho_i, S_{\rho_i}) \rightarrow \Xi_{\rho_i} = \{\xi_1, \ldots, \xi_k\} \tag{14}$$

where Ξ_{ρ_i} denotes the set of all contributor tuples which are sufficient to serve ρ_i, S_{ρ_i} is the set of contributors who collectively participate in serving ρ_i, ξ_i is a single contributor tuple generated from a contributor s_i, and k is the size of the Ξ_{ρ_i} set.

A contributor tuple ξ_i is represented as follows:

$$\xi_i = \langle s, si, \tau, d_\xi, \omega \rangle \tag{15}$$

where s denotes a contributing service's unique name (i.e., actor name), si represents the sensing instructions, τ is the sensing time, d_ξ is the maximum time delay between the sensing time and sending data to the service coordinator, and ω represents the sensing data. Note that ω is empty before filling the sensing data by the contributor service.

The following transition rule shows the process of assigning the sensing tasks represented in the contributor tuples Ξ_{ρ_i} to S_{ρ_i}:

$$\begin{gathered}\langle [\![I:\ [R[multicast(S_{\rho_i}, \Xi_{\rho_i})]]_{a_{s_d}}\ ,\ \alpha\ :\ O]\!]_{s_d},\ S \mid C \mid M \rangle \\ \rightarrow \langle [\![I:\ [R[nil]]_{s_d}\ ,\ \alpha\ :\ O]\!]_{s_d},\ S \mid C \mid \langle S_{\rho_i} \Leftarrow M_\Xi \rangle,\ M \rangle\end{gathered} \tag{16}$$

where s_d is a directory service; S_{ρ_i} is a set of contributing services which will participate in serving ρ_i, Ξ_{ρ_i} is the set of the contributor tuples (i.e., sensing tasks); M_Ξ is a set of actor messages where each $m \in M_{\Xi}$ is a message sent to $s_j \in S_{\rho_i}$, where $m = \langle s_j \Leftarrow \Xi_{s_j} \rangle$; and $\Xi_{s_j} \subset \Xi_{\rho_i}$ such that Ξ_{s_j} is s_j's sensing assignment.

On receiving its assigned set of contributor tuples Ξ_c, a contributor c adds them to its schedule, as follows:

$$\begin{aligned} &\langle s \mid \{(recieve)_c, C\} \mid \Theta \mid P|m, \mu|t\rangle \rightarrow \\ &\quad \langle s \mid [add\_to\_schedule(\Xi_c, st)_c]_c, C \mid \Theta \mid P \mid \mu|t\rangle \end{aligned} \tag{17}$$

where $m = \langle c \Leftarrow \Xi_c \rangle$ is a schedule message, Ξ_c is the contributor tuples assigned to contributor c, and st_c is the contributor's state.

Each contributor c senses/executes the tuples that have a sensing time equals to t. After the sensing computation in ξ_τ is executed, d is filled in with the sensing data, and a serve-next message is sent to the contributor itself in order to serve the next message or sensing task. The contributor then sends the sensed data d to the server when $t \geq \tau + \delta$, as described by the following transition rule:

$$\begin{aligned} &\langle [I : [sense(\xi)]_a, \alpha : \mu : O]_{s_1}, S|C|\mu|t\rangle \rightarrow \\ &\quad \langle [I : (receive)_a, \alpha : \mu : O]_{s_1}, S|C|m, \mu|t\rangle \end{aligned} \tag{18}$$

where s_1 is a contributing service, s_2 is a consuming service, $\xi = \langle s_2, si, \tau, d_\xi, nil\rangle$, $\xi' = \langle s_2, si, \tau, d_\xi, \omega\rangle$, and $m = \langle s_2 \Leftarrow \omega\rangle$ is a data contribution message.

On receiving a contribution message $m = \langle s \Leftarrow \omega\rangle$, the server actor proceeds to aggregate the message and update the request's state, as follows:

$$\begin{aligned} &\langle (receive)_s|C|\Theta|\{\langle \theta, \sigma, td, \pi\rangle, P\}|m, \mu|t\rangle \rightarrow \\ &\quad \langle [aggr(d, \theta)]_s|C|\Theta|P|\mu|t\rangle \end{aligned} \tag{19}$$

where $m = \langle s \Leftarrow d\rangle$ is a contribution message, where θ is the current state of the request ρ_s.

If the aggregation condition is met and $td \geq t + d_\xi$, the server notifies a service update to the client:

$$\begin{aligned} &\langle [aggr(d, \theta)]_s|C|\Theta|\{\langle \theta, \sigma, td, \pi\rangle, P\}|\mu|t\rangle \rightarrow \\ &\quad \langle (receive)_s|C|\Theta|\{\langle \theta', \sigma, td, \pi\rangle, P\}|m, \mu|t\rangle \end{aligned} \tag{20}$$

where $m = \langle \sigma \Leftarrow \theta'\rangle$, θ is the current state of the request ρ_s, and θ' is the new state of the request.

If the aggregation condition is not met, the server updates the request's state:

$$\begin{aligned} &\langle [aggr(d, \theta)]_s|C|\Theta|\{\langle \theta, \sigma, td, \pi\rangle, P\}|\mu|t\rangle \rightarrow \\ &\quad \langle (receive)_s|C|\Theta|\{\langle \theta', \sigma, td, \pi\rangle, P\}|\mu|t\rangle \end{aligned} \tag{21}$$

Theorem 2. *If there is a delay which can be tolerated to a group of sensing requests at a contributor c, one sampling instance is carried out for all of them.*

Proof. According to Axiom 6, if the delay in sampling can be tolerated to a group of sampling requests at a contributor c, c can adjust their sampling rates within a predefined delay so that they all can share the same sampling instance. In this case, there must be a set of contributors $C_{\rho_i} \in C$ that have conditionally committed to collectively serve a set of client tuples Z_{ρ_i}, which is a superset of ρ_i. According to Axiom 3, if a contributor $c \in C_{\rho_i}$ conditionally commits to serve its part $\xi_c \in \rho_i$, this means c has indicated its ability to serve ξ_c to the server, and xi_c can be accommodated in c's sensing schedule.

- c can determine the lowest sensing rate that would satisfy all sensing requests;
- All requests have compatible delay tolerances with those of requests already being served.

□

5.2. Fine-Grained Resource Model for IoT Services

Our model assumes that IoT services are owned by their initiator. This capability can also inform the pricing of using services. The directory service can set the price of creating a new service, and the initiator service can charge the cost of using its services.

Property 1. *For an accepted request ρ, the time delay to send any aggregated update θ in ξ to a client σ does not exceed the max-delay of the client tuple, d_ξ, specified by σ.*

Proof. For the same client tuple ξ, the max delay of the contributor tuples Ξ_ξ mean it is limited by the oldest tuple in the aggregate. This is assuming that the sensed data in any contributor tuple ξ, including the oldest feed, must be collected and sent to the server no later than $d_\xi - \delta$, where δ is the maximum time delay between the sensing time and sending data to the server. Therefore, θ is sent to σ no later than d_ξ. This proves the property. □

Property 2. *There are no wasted contributions in the system.*

Proof. Assume that a request ρ is accepted in the system. According to Axiom 1, if a request ρ is accepted in the system, there must be at least one set $\rho_i \in \rho$ that can be served by the server. According to Axioms 2 and 5, there must be a set of contributors $C_{(\rho_s)}$ which have unconditionally committed to serve a set of client tuples, $Z_{(\rho_i)}$, which is exactly equal to the set ρ_i so that each and every member in $Z_{(\rho_i)}$ is a member in ρ_i. Therefore, there are no wasted contributions in the system. This proves the property. □

A contributor context sampling is a process of obtaining a contributor context by the mobile device. To quantitatively describe the consumption of resources, particularly the energy consumed by sensors involved in the sampling, in the proposed IoT system, we adopt the following energy model. Each of the sampling methods supported by the mobile device is associated with a resource cost ω required for its invocation for context sampling and is determined by the following function:

$$\omega = f_{cost}(E, s) \tag{22}$$

A total cost of the IoT system functioning is determined by costs of the individual contributor context samplings and is represented as the sum of their individual costs:

$$\Omega_{sampling} = \sum_{i=0}^{|c|} \omega_i \tag{23}$$

Given a service list, the goal of the IoT system is to define the function $f_{sampling}$ that maximizes the number of IoT services that are hosted at the contributor's device, and minimizes the total cost Ω of consumed resources.

We use a sliding-window concept to aggregate the most recently collected sensor data from contributors as input to the aggregation function. We collect a series of sensor feeds from different contributors at the same time, denoted as $\Xi = \{\xi_1, \xi_2, \dots \xi_n\}$. Each service consumes ε computational resource units from each contributor during data acquisition. The total computation cost can be expressed by:

$$\Omega_{comp} = \sum_{i=0}^{\Theta^s} \varepsilon_i \tag{24}$$

We define β to denote the communication cost of unit data, and the total communication cost can be expressed by:

$$\Omega_{comm} = \beta \sum_{i=0}^{\Theta^s} d_i \tag{25}$$

where d_i is the size of the sensing data.

We consider three types of overhead costs in the IoT system: context sampling, computation, and communication. The total cost within a long time period $[1, T]$ can be calculated by:

$$\Omega_{total} = \sum_{t=1}^{T} \Omega_{sampling}(t) + \Omega_{comp}(t) + \Omega_{comm}(t) \tag{26}$$

Our objective is to calculate the amount of resources Ω_{total} a service uses.

We represent the availability of each resource (e.g., sensor sampling, CPU cycles, network utilization, etc.) in a system of services by a resource term $\epsilon = [r, l_r]^{\lambda}_{\tau}$, where:

- $\langle r, l_r \rangle$ denotes the located type of the specified resource. It contains both the type of the resource r and the relative location (locality) of the resource l_r. The locality could be relative or absolute based on the type of the service. Note that the location where the resource is residing is dynamic, as we deal with mobile resources.
- λ represents the rate of availability of the resource, in quantity or time. For example, if the specified resource is a contributor's sensor, then this parameter represents the sampling rate of that sensor which is the time between any two consecutive context samplings. The higher its value, the more frequent the context is sampled. In addition, this parameter adjusts the required accuracy when sampling the context. The higher its value, the more precise the sampled context is.
- τ is the time interval during which the resource exists, where $\tau = \langle t_{start}, t_{end} \rangle$

For example, consider that a GPS sensor at a contributor is sampled at rate 10 Hz and available at a time interval $\langle 0, 5 \rangle$, and can be represented as $[GPS, l_{GPS}]^{10}_{\langle 0,5 \rangle}$.

As each resource term is associated with a time interval τ, relationships between time intervals must be defined before we can discuss the operations on resource terms. We use interval algebra to formalize relations between two time intervals, as shown in Figure 3.

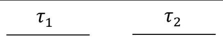
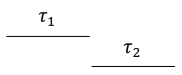
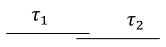
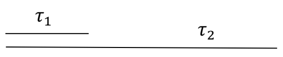
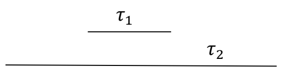
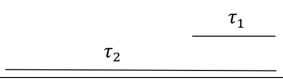
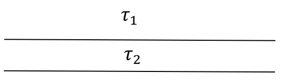
Relation	**Illustration**	**Interpretation**
$\tau_1 < \tau_2$		τ_1 takes place before τ_2
$\tau_1\ m\ \tau_2$		τ_1 meets τ_2
$\tau_1\ o\ \tau_2$		τ_1 overlaps τ_2
$\tau_1\ s\ \tau_2$		τ_1 starts τ_2
$\tau_1\ d\ \tau_2$		τ_1 during τ_2
$\tau_1 f \tau_2$		τ_1 finishes τ_2
$\tau_1 = \tau_2$		τ_1 equals τ_2

Figure 3. Possible relations between two time intervals.

If two resource terms in a resource set have the same located types and overlapping time intervals, they can be combined as follows:

$$\epsilon_1 \cup \epsilon_2 = [r, l_r]^{\lambda_1}_{\tau_1} \cup [r, l_r]^{\lambda_2}_{\tau_2} = \{[r, l_r]^{\lambda_1}_{\tau_1 \setminus \tau_2}, [r, l_r]^{\lambda_2}_{\tau_2 \setminus \tau_1}, [r, l_r]^{\lambda_2 + \lambda_2}_{\tau_2 \cap \tau_1}\} \tag{27}$$

where for any interval for which they overlap, their rates are added, and for remaining intervals, they are represented separately in the set, and $\setminus$ is a relative complement operation.

Resource requirements of service requests: A service request consumes resources at every step of its execution. We represent request's computations in terms of the resources they consume. Request's computations in our model can be divided into three categories: sensing computation at a contributor device, CPU processing at the server side, and network utilization.

We represent the resource requirements for a contributor tuple ξ as follows:

$$[R]_{\xi}^{(s,d)} = \Phi(\xi, s, d), R = \{r_1, \ldots, r_n\}, r_i = [q]_{\langle r,l \rangle} \tag{28}$$

where Φ is a function which takes a contributor tuple ξ as a parameter and returns a set of resource amounts representing the required resources for serving that tuple, s is the earliest start time of executing ξ (i.e., sensing time), d is the deadline by which the sensing task must be completed, and q is the quantity of resource required. For example:

$$\begin{aligned} \Phi(\xi,5,8) = [R]_{\xi}^{(5,8)} = [\{[20k]_{\langle CPU, l_C PU \rangle}, [100]_{\langle GPS, l_G PS \rangle} \\ , [1k]_{\rangle Acc, l_{Acc} \rangle}, [100ms]_{\langle net, l_{net} \rangle}]^{(5,8)} \end{aligned} \tag{29}$$

Consider a function f, which, when a resource set E and the resource requirement of a contributor tuple $[R]_{\xi}^{(s,d)}$ are provided as parameters, returns a Boolean value true or false, indicating whether or not the contributor tuple can be served given the available set of resources:

$$f(E, \Phi(\xi, s, d)) = (\bigcup_{s}^{d} E \geq \Phi(\xi)) = true | false \tag{30}$$

where $\bigcup_s^d E$ gives the union of all resources in $E = \{\epsilon_1, \ldots, \epsilon_n\}$ which exist in the interval (s, d). For each member r in R, f searches in the set of available resources E for the amount of resources required to service r.

Axiom 8: Single contributor tuple accommodation. A contributor tuple ξ can be accommodated by a system if, by time s, the system satisfies the resource requirement $\Phi(\xi, s, d)$, i.e., $(E, \Phi(\xi, s, d)) = true$, where E is the available resources of the system.

Axiom 9: Sequential contributor tuples accommodation. A system with resources E can accommodate a sequential contributor tuples, represented in a single client tuple ξ, if the system can satisfy the simple resource requirements for each contributor tuple in ξ.

We represent the requirements of a set of contributor tuples in ξ sequentially, as follows:

$$\Phi(\xi, s, d) ::= \Phi(\xi_1, s, t_1) \cup \Phi(\xi_2, t_1, t_2) \cup \ldots \cup \Phi(\xi_m, t_{m-1}, d) \tag{31}$$

where $s \leq t_1 \leq t_2 \leq \ldots \leq t_{m-1} \leq d$.

A concurrent computation consists multiple actors (e.g., contributors). Resource requirements of a concurrent computation $\Phi(\rho_i, s, d)$ can be satisfied by satisfying resource requirements of the individual actors, as follows:

$$\Phi(\rho_i, s, d) = \Phi(\varsigma_{c_1}, s, d) \cup \Phi(\varsigma_{c_2}, s, d) \cup \ldots \cup \Phi(\varsigma_{c_n}, s, d) \tag{32}$$

where ς_{c_n} represents computations carried out by actor c_n.

6. A Distributed Runtime Environment for IoT Services

To narrow the gap between the generation of sensor data and its interpretation, novel artificial-intelligence (AI) methods are critically needed to smartly analyze and process the vast amount of sensor data on the network's edge. This section presents the design of an AI-powered runtime environment for smart sensing to support the programmability of IoT services.

The proposed runtime environment offers an adaptable, configurable, highly reliable, and reuse-driven sensing platform that can be easily reused in the development of IoT services. This platform will open new opportunities to explore various sensor fusion architectures, including sensory front end, read-out electronics, power systems with respect to volume, mass, and power consumption. Figure 4 illustrates the software stack of the proposed AI-power platform for smart sensing, which will include three layers:

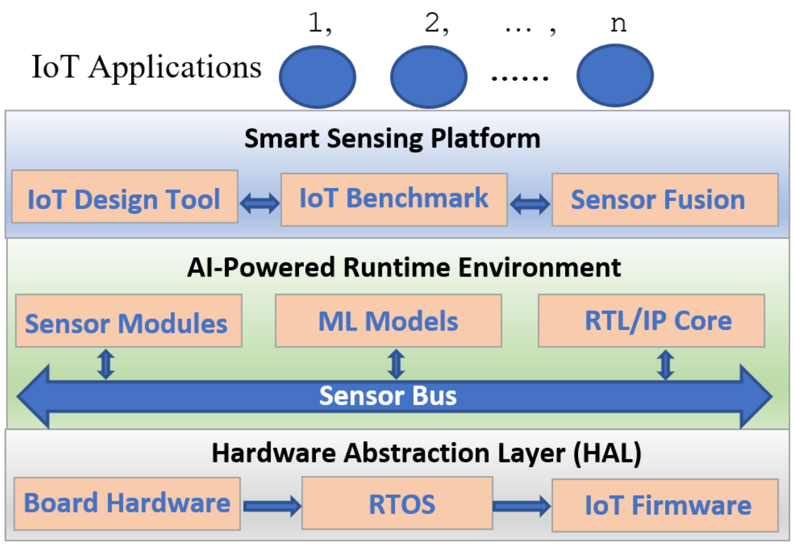


Figure 4. The proposed AI-power platform for smart sensing.

Hardware abstraction layer (HAL): HAL is a software layer which enables access to the hardware components of IoT devices such as random-access memory (RAM), read-only memory (ROM), and serial interfaces, etc. This layer provides an innovative hardware building block with computation, storage, communication, and re-programmability, as well as various heterogeneous computing elements of a GPU, a central processing unit (CPU), digital signal processing (DSP), field programmable gate arrays (FPGAs), and a tensor processing unit (TPU) to meet various IoT sensing requirements. It also includes real-time operating systems (RTOS) and IoT firmware that are particularly suited for small and constrained devices, and that can provide IoT-specific capabilities.

AI-powered runtime environment: The AI-enabled runtime environment will run over the HAL layer. It hosts several sensor modules which can interact with the ML models through a senor bus over the network. This layer also provides a register-transfer-level (RTL) intellectual property (IP) cores library. The RTL/IP core implements a set of parameterizable DSP algorithms commonly used by IoT sensors and other communication IP cores for building a complete sensor-data processing system.

Smart sensing platform: IoT applications are deployed on top of the runtime environment. Applications are defined as a workflow of functions, where each function can provide data generation (e.g., in-situ sensors), processing, and data consumption (e.g., actuators). Function-code execution is triggered by specific events such as receiving new sensing data from a sensor locally or remotely, or detecting a change in the state of a sensor. Furthermore, this layer includes an IoT benchmark for performance validation and resource usage estimation, and a collection of test procedures, scripts, test benches, and other monitoring and debugging utilities that can be reused for algorithm development and the IoT ecosystem implementation. In addition, we developed an integrated IoT toolchain which will provide a complete design flow starting from dataflow architectural exploration, model development, resource optimization to physical implementation based on the platform components.

The runtime environment enables programmers to write a single actor program which will automatically become distributed to run on many devices at the edge of the network. Actors are the building blocks for an application which will be deployed on the top the runtime environment. The actors are connected in a dataflow to form the application. This simplifies actor migration between runtimes and the matching of actor requirements with runtime capabilities. If the runtime does not meet the requirements posed by a deployed actor, then the actor will be automatically migrated to a runtime that can satisfy the requirements.

7. Summary

This paper presents a formal model for representing resource-bounded IoT services. The model relies on the actor model to define services. We precisely described the syntax and operational semantics of the model, in which it represents the requirements of an IoT service as sets of temporally defined feeds, each with its currency and contextual constraints. For each feed, the service provider secures appropriate sensor data feeds and carries out the required aggregations and customizations for different clients or client groups. The paper also presented a proximity model which enables the representation of and reasoning and decision making about the dynamically evolving relationship between a service's sensing and acting capabilities and the environments in which these capabilities are exercised. The decision-making capability, in essence, will be used to drive mobility. Furthermore, we present the design of an innovative runtime environment for implementing IoT services with AI capabilities. The proposed runtime environment offers high-level primitives supported by a middleware implementing key mechanisms required by IoT services, allowing application developers to focus on application-specific concerns. The presented model and runtime environment aims to support the implementation of innovative sensor-based IoT services involving distributed and mobile devices of various types by providing high-level programming constructs.

In on-going work, we are working on evaluating the developed runtime environment experimentally to assess its performance and scalability. In addition, we will implement deep-learning models into the IoT runtime environment to enable the intelligence of sampling, processing, analyzing and communications of sensor data. Finally, we plan to apply an AI-guided system-level optimization to this new IoT system to dynamically allocate the resources and adjust parameters based on the application scenarios.

Funding: This research work is supported in part by the National Science Foundation (NSF) under grant # 2011330. Any opinions, findings, and conclusions expressed in this paper are those of the authors and do not necessarily reflect NSF's views.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data and source code are available upon request.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Abdelmoamen, A.; Jamali, N. A Model for Representing Mobile Distributed Sensing-Based Services. In Proceedings of the IEEE International Conference on Services Computing, SCC'18, San Francisco, CA, USA, 2–7 July 2018; pp. 282–286.
2. Moamen, A.A.; Jamali, N. Opportunistic Sharing of Continuous Mobile Sensing Data for Energy and Power Conservation. *IEEE Trans. Serv. Comput.* **2020**, *13*, 503–514. [CrossRef]
3. Ahmed, A.A.; Echi, M. Hawk-Eye: An AI-Powered Threat Detector for Intelligent Surveillance Cameras. *IEEE Access* **2021**, *9*, 63283–63293. [CrossRef]
4. Moamen, A.A.; Jamali, N. An Actor-Based Middleware for Crowd-Sourced Services. *EAI Endorsed Trans. Mob. Commun. Appl.* **2017**, *3*, e1. [CrossRef]
5. Hwang, J.; Aziz, A.; Sung, N.; Ahmad, A.; Le Gall, F.; Song, J. AUTOCON-IoT: Automated and Scalable Online Conformance Testing for IoT Applications. *IEEE Access* **2020**, *8*, 43111–43121. [CrossRef]
6. Hwang, J.; Nkenyereye, L.; Sung, N.; Kim, J.; Song, J. IoT Service Slicing and Task Offloading for Edge Computing. *IEEE Internet Things J.* **2021**, *8*, 11526–11547. [CrossRef]
7. Li, Y.; Zhuang, Y.; Hu, X.; Gao, Z.; Hu, J.; Chen, L.; He, Z.; Pei, L.; Chen, K.; Wang, M.; et al. Toward Location-Enabled IoT (LE-IoT): IoT Positioning Techniques, Error Sources, and Error Mitigation. *IEEE Internet Things J.* **2021**, *8*, 4035–4062. [CrossRef]
8. Ahmed, A.A.; Reddy, G.H. A Mobile-Based System for Detecting Plant Leaf Diseases Using Deep Learning. *AgriEngineering* **2021**, *3*, 478–493. [CrossRef]

9. lv, H.; Ge, X.; Zhu, H.; Wang, C.; Yuan, Z.; Zhu, Y. Design and Implementation of Reactive Distributed Internet of Things Platform based on Actor Model. In Proceedings of the 2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC), Chengdu, China, 15–17 March 2019; pp. 1993–1996. [CrossRef]
10. Ahmed, A.A.; Olumide, A.; Akinwa, A.; Chouikha, M. Constructing 3D Maps for Dynamic Environments using Autonomous UAVs. In Proceedings of the 2019 EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous'19), Houston, TX, USA, 12–14 November 2019; pp. 504–513.
11. Moamen, A.A.; Jamali, N. Coordinating Crowd-Sourced Services. In Proceedings of the 2014 IEEE International Conference on Mobile Services, Anchorage, AK, USA, 27 June–2 July 2014; pp. 92–99.
12. Agha, G. *Actors: A Model of Concurrent Computation in Distributed Systems*; MIT Press: Cambridge, MA, USA, 1986.
13. Tan, W.; Fan, Y.; Zhou, M.; Tian, Z. Data-Driven Service Composition in Enterprise SOA Solutions: A Petri Net Approach. *IEEE Trans. Autom. Sci. Eng.* **2010**, *7*, 686–694. [CrossRef]
14. Nastic, S.; Sehic, S.; Vögler, M.; Truong, H.L.; Dustdar, S. PatRICIA—A Novel Programming Model for IoT Applications on Cloud Platforms. In Proceedings of the 2013 IEEE 6th International Conference on Service-Oriented Computing and Applications, Koloa, HI, USA, 16–18 December 2013; pp. 53–60. [CrossRef]
15. Mora, S.; Gianni, F.; Divitini, M. RapIoT Toolkit: Rapid Prototyping of Collaborative Internet of Things Applications. In Proceedings of the 2016 International Conference on Collaboration Technologies and Systems (CTS), Orlando, FL, USA, 31 October–4 November 2016; pp. 438–445. [CrossRef]
16. Moamen, A.A.; Jamali, N. An Actor-Based Approach to Coordinating Crowd-Sourced Services. *Int. J. Serv. Comput. (IJSC)* **2014**, *2*, 43–55. [CrossRef]
17. Harjula, E.; Karhula, P.; Islam, J.; Leppänen, T.; Manzoor, A.; Liyanage, M.; Chauhan, J.; Kumar, T.; Ahmad, I.; Ylianttila, M. Decentralized Iot Edge Nanoservice Architecture for Future Gadget-Free Computing. *IEEE Access* **2019**, *7*, 119856–119872. [CrossRef]
18. Kura. Eclipse IoT Framework. 2022. Available online: https://www.eclipse.org/kura/ (accessed on 10 October 2022).
19. Mayer, J.; Pampana, V.; Bernard, M.; Bytschkow, D.; Stohl, T.; Gupta, P.; Duchon, M. Holonic architectures for IoT-empowered energy management in districts. In Proceedings of the 2021 IEEE 7th World Forum on Internet of Things (WF-IoT), New Orleans, LA, USA, 14 June–31 July 2021; pp. 189–194. [CrossRef]
20. Moamen, A.A.; Jamali, N. ShareSens: An Approach to Optimizing Energy Consumption of Continuous Mobile Sensing Workloads. In Proceedings of the 2015 IEEE International Conference on Mobile Services (MS'15), New York, NY, USA, 27 June–2 July 2015; pp. 89–96.