INTERNET OF THINGS

Module - 01

⇒ Overview of Internet of Things

We are at the beginning of an emerging era where ubiquitous communication and connectivity is neither a dream nor a challenge anymore. Subsequently, the focus has shifted toward a seamless integration of people and devices to converge the physical realm with human-made virtual environments, creating the so- called Internet of Things (IoT) utopia.

A closer look at this phenomenon reveals two important pillars of IoT: “Internet” and “Things” that require more clarification. Although it seems that every object capable of connecting to the Internet will fall into the “Things” category, this notation is used to encompass a more generic set of entities, including smart devices, sensors, human beings, and any other object that is aware of its context and is able to communicate with other entities, making it accessible at any time, anywhere. This implies that objects are required to be accessible without any time or place restrictions.

Ubiquitous connectivity is a crucial requirement of IoT, and, to fulfill it, applications need to support a diverse set of devices and communication protocols, from tiny sensors capable of sensing and reporting a desired factor, to powerful back-end servers that are utilized for data analysis and knowledge extraction.

⇒ Open source Semantic Web Infrastructure for Managing IoT Resources In the Cloud

Introduction

The cloud computing paradigm realizes and promotes the delivery of hardware and software resources over the Internet, according to an on-demand utility-based model. Depending on the type of computing resources delivered via the cloud, cloud services take different forms, such as Infrastructure as a service (IaaS), Platform as a service (PaaS), Software as a service (SaaS), Storage as a service (STaaS), and more. These services hold to promise to deliver increased reliability, security, high availability, and improved QoS at an overall lower total cost of ownership. At the same time, the IoT paradigm relies on the identification and use of a large number of heterogeneous physical and virtual objects (ie,

both physical and virtual representations), which are connected to the Internet. IoT enables the communication between different objects, as well as the in-context invocation of their capabilities (services) toward added-value applications.

Since the early instantiations and implementations of both technologies, it has become apparent that their convergence could lead to a range of multiplicative benefits. Most IoT applications entail a large number of heterogeneous geographically distributed sensors. As a result, they need to handle numerous sensor streams, and could therefore directly benefit from the immense distributed storage capacities of cloud computing infrastructures. Furthermore, cloud infrastructures could boost the computational capacities of IoT applications, given that several multisensor applications need to perform complex processing that is subject to timing and other QoS constraints. Also, a great deal of IoT services (eg, large-scale sensing experiments and smart-city applications) could benefit from a utility-based delivery paradigm, which emphasizes the on-demand establishment and delivery of IoT applications over a cloud-based infrastructure.

Background / Related Work

The proclaimed benefits of the IoT/cloud convergence have (early on) given rise to research efforts that attempted to integrate multisensory services into cloud computing infrastructures. Early efforts have focused on the development of pervasive (sensor-based) grid-computing infrastructures, which emphasized modeling sensors and their data as a resource, and, accordingly, enabling real-time access, sharing, and storage of sensor data. Sensor Grids have been used for a number of pervasive computing applications, notably community-sensing applications such as meteorology. With the advent of cloud computing, the convergence of the cloud computing with WSN infrastructures has been attempted, as an extension of the sensor grid concept in the scope of on-demand elastic cloud-based environments. The convergence of cloud computing with WSN aimed at compromising the radically different and conflicting properties of the two (ie, IoT and cloud) technologies. In particular, sensor networks are location-specific, resource constrained, expensive (in terms of development/deployment cost), and generally inflexible in terms of resource access and availability. On the contrary, cloud-based infrastructures are location-independent and provide a wealth of inexpensive resources, as well as rapid elasticity. Sensor clouds come to bridge these differences and endow WSN with some cloud properties. Other issues are associated with energy efficiency and the proper handling of service-level agreements. Most recent research initiatives are focusing on real-life implementation of sensor clouds, including open source implementations.

In addition to research efforts toward sensor-clouds, there are also a large number of commercial online cloud-like infrastructures, which enable end users to attach their devices on the cloud, while also enabling the development of applications that use those devices and the relevant sensor streams.

OpenIoT Architecture for IoT/Cloud Convergence

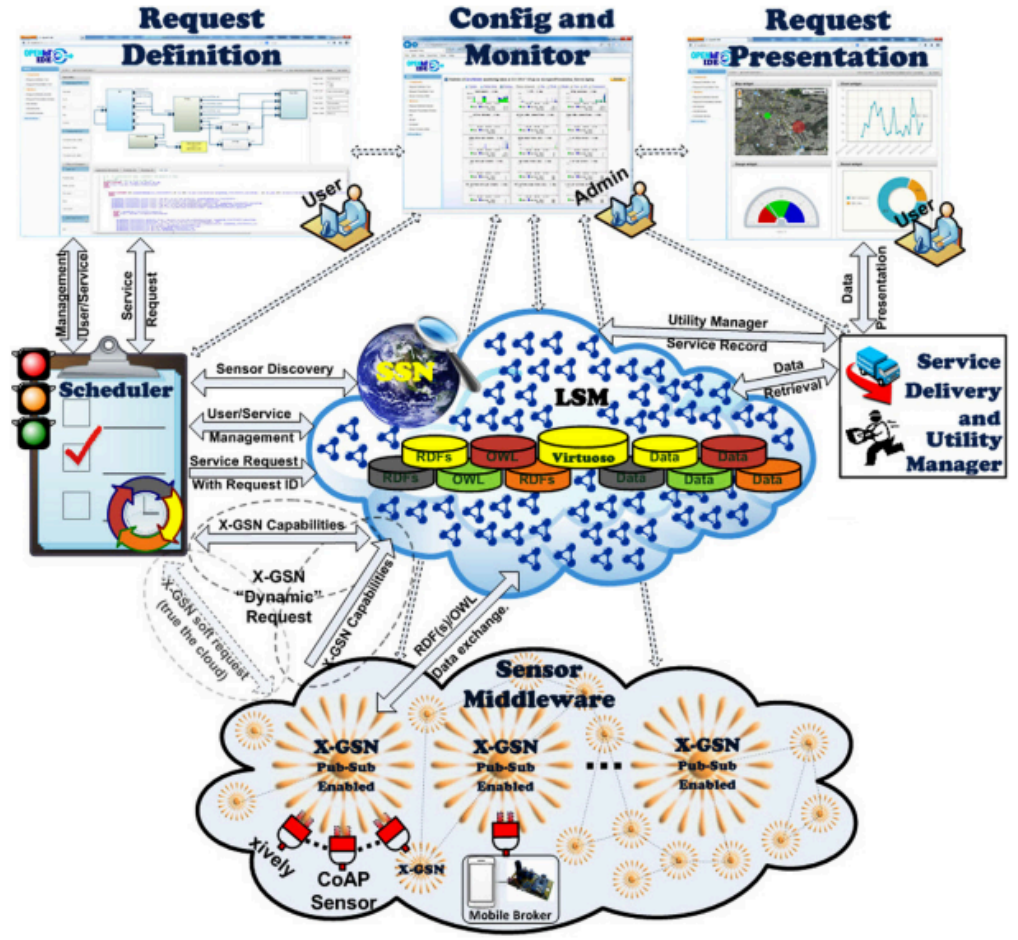

FIGURE 2.1 OpenIoT Architecture for IoT/Cloud Convergence

Our approach for converging IoT and cloud computing is reflected in the OpenIoT architecture, which is depicted in Fig. 2.1. The figure illustrates the main elements of the OpenIoT software architecture along with their interactions and functionalities, in particular:

- The Sensor Middleware, which collects, filters, and combines data streams stemming from virtual sensors (eg, signal-processing algorithms, information- fusion algorithms, and social-media data streams) or physical-sensing devices (such as temperature sensors, humidity sensors, and weather stations). This middleware acts as a hub between the OpenIoT platform and the physical world, as it enables the access to information stemming from the real world.

- The Cloud Computing Infrastructure, which enables the storage of data streams stemming from the sensor middleware, thereby acting as a cloud database. The cloud infrastructure also stores metadata for the various services, as part of the scheduling process, which is outlined in the next section. In addition to data streams and metadata, computational (software) components of the platform could also be deployed in the cloud in order to benefit from its elasticity, scalability, and performance characteristics
- The Directory Service, which stores information about all the sensors that are available in the OpenIoT platform. It also provides the means (ie, services) for registering sensors with the directory, as well as for the look-up (ie, discovery) of sensors. The IoT/cloud architecture specifies the use of semantically annotated descriptions of sensors as part of its directory service.
- The Global Scheduler, which processes all the requests for on-demand deployment of services, and ensures their proper access to the resources (eg, data streams) that they require. This component undertakes the task of parsing the service request, and, accordingly, discovering the sensors that can contribute to its fulfillment. It also selects the resources, that is, sensors that will support the service deployment, while also performing the relevant reservations of resources.
- The Local Scheduler component, which is executed at the level of the Sensor Middleware, and ensures the optimized access to the resources managed by sensor middleware instances (ie, GSN nodes in the case of the OpenIoT implementation).
- The Service Delivery and Utility Manager, which performs a dual role. On the one hand, it combines the data streams as indicated by service workflows within the OpenIoT system, in order to deliver the requested service.
- The Request Definition tool, which enables the specification of service requests to the OpenIoT platform. It comprises a set of services for specifying and formulating such requests, while also submitting them to the Global Scheduler.

Fig. 2.1 does not specify implementation technologies associated with the various components, thus providing an abstract presentation of the functional elements of the architecture. OpenIoT is, however, implemented on the basis of specific implementation technologies [such as GSN for the sensor middleware, W3C SSN for the directory service, and JSF (Java Server Faces) libraries (such as Primefaces)]. Alternative implementations based on alternate technologies are however possible.

Scheduling Process And IoT Services Lifecycle

The Global Scheduler component is the main and first entry point for service requests submitted to the cloud platform. It parses each service request and accordingly performs two main functions toward the delivery of the service, namely the selection of the sensors/ICOs involved in the service, but also the reservation of the needed resources. The scheduler manages all the metadata of the IoT services, including: (1) The signature of the service (ie, its input and output parameters), (2) the sensors/ICOs used to deliver the service, and (3) execution parameters associated with the services, such as the intervals in which the service shall be repeated, the types of visualization (in the request presentation), and other resources used by the service. In principle, the Global Scheduler component keeps track of and controls the lifecycle of IoT services, which is depicted in Fig. 2.2.

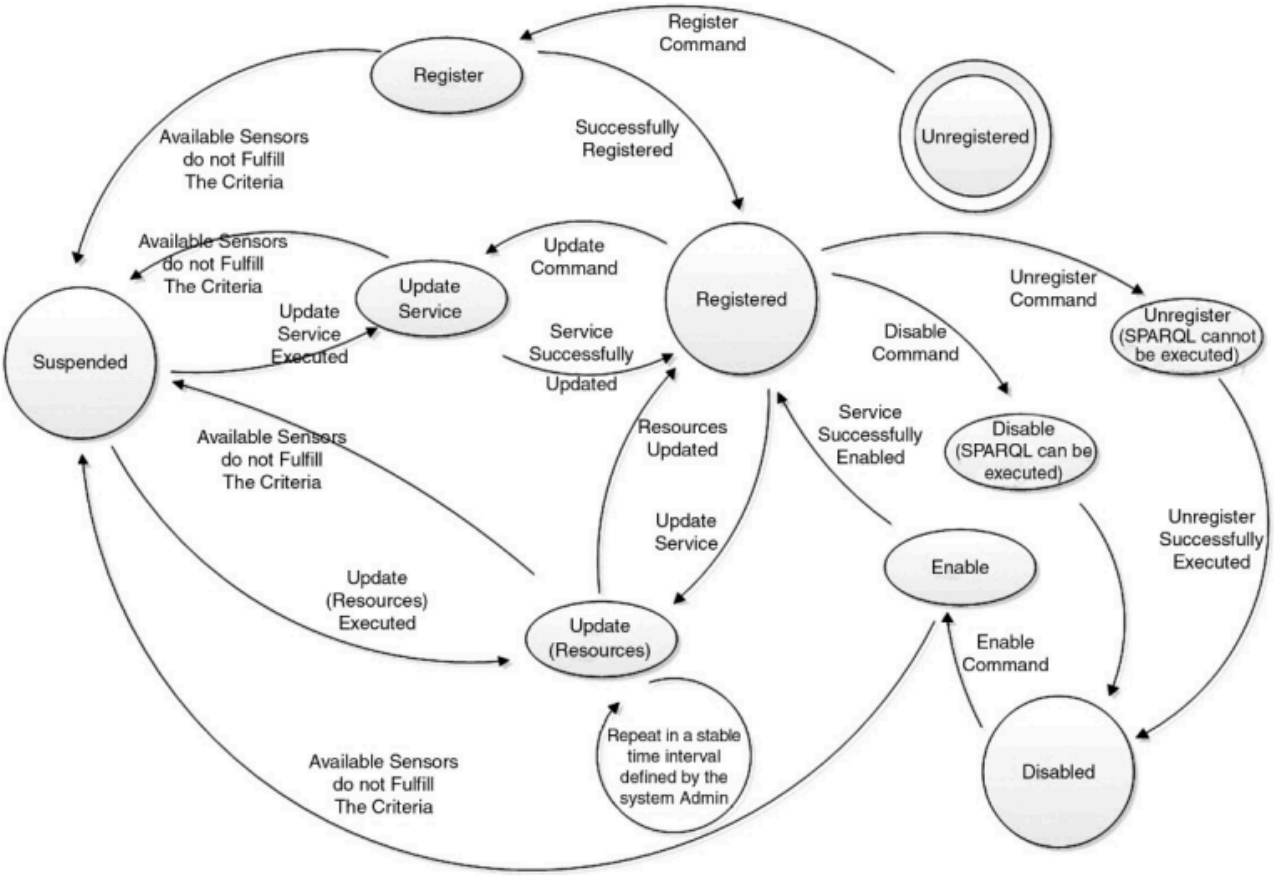

FIGURE 2.2 State Diagram of the OpenIoT Services Lifecycle Within the Scheduler Module

- Resource Discovery: This service discovers a virtual sensor's availability. It therefore provides the resources that match the requirements of a given request for an IoT service.
- Register: This service is responsible for establishing the requested service within the cloud database. To this end, it initially identifies and logs (within the cloud) all the

sensors/ICOs, which are pertinent and/or needed for delivering the requested IoT service.

- Unregister: In the scope of the unregister functionality for given IoT service (identified through its ServiceID), the resources allocated for the service (eg, sensors used) are released (ie, disassociated from the service).
- Suspend: As part of suspend functionality, the service is deactivated and therefore its operation is ceased. Note however, as part of the suspension the platform does not release the resources associated with the service.
- Enable from Suspension: This functionality enables a previously suspended service. The data structures holding the service's metadata in the cloud are appropriately updated.
- Enable: This service allows the enablement of an unregistered service. In practice, this functionality registers the service once again in the platform, through identifying and storing the required sensors/ICOs.
- Update: This service permits changes to the IoT service. In particular, it allows for the updating of the service's lifecycle metadata (ie, signature, sensors/ICOs, execution parameters) according to the requested changes.

In order to support the lifecycle services outlined previously, the scheduler provides an appropriate API, which also enables the services' transition between the various lifecycle states. The platform also supports baseline authentication and access-control mechanisms, which allows specific users to access resources and services available within the platform. Note that only registered users are able to leverage these aforementioned lifecycle management functionalities.

The following figures illustrate the details of some lifecycle-management use cases within the Scheduler component. In particular, Fig. 2.3 illustrates the main workflow associated with the service registration process. In the scope of this process, the Scheduler attempts to discover the resources (sensors, ICO) that will be used for the service delivery. In case there are no sensors/ICOs that can fulfill the request, the service is suspended. In case a set of proper sensors/IOCs is defined, the relevant data entities are updated (eg, relationship of sensors to services) and a SPARQL script associated with the service is formulated and stored for later use. Following the successful conclusion of this process, the servicer enters the "Registered" state and is available for invocation.

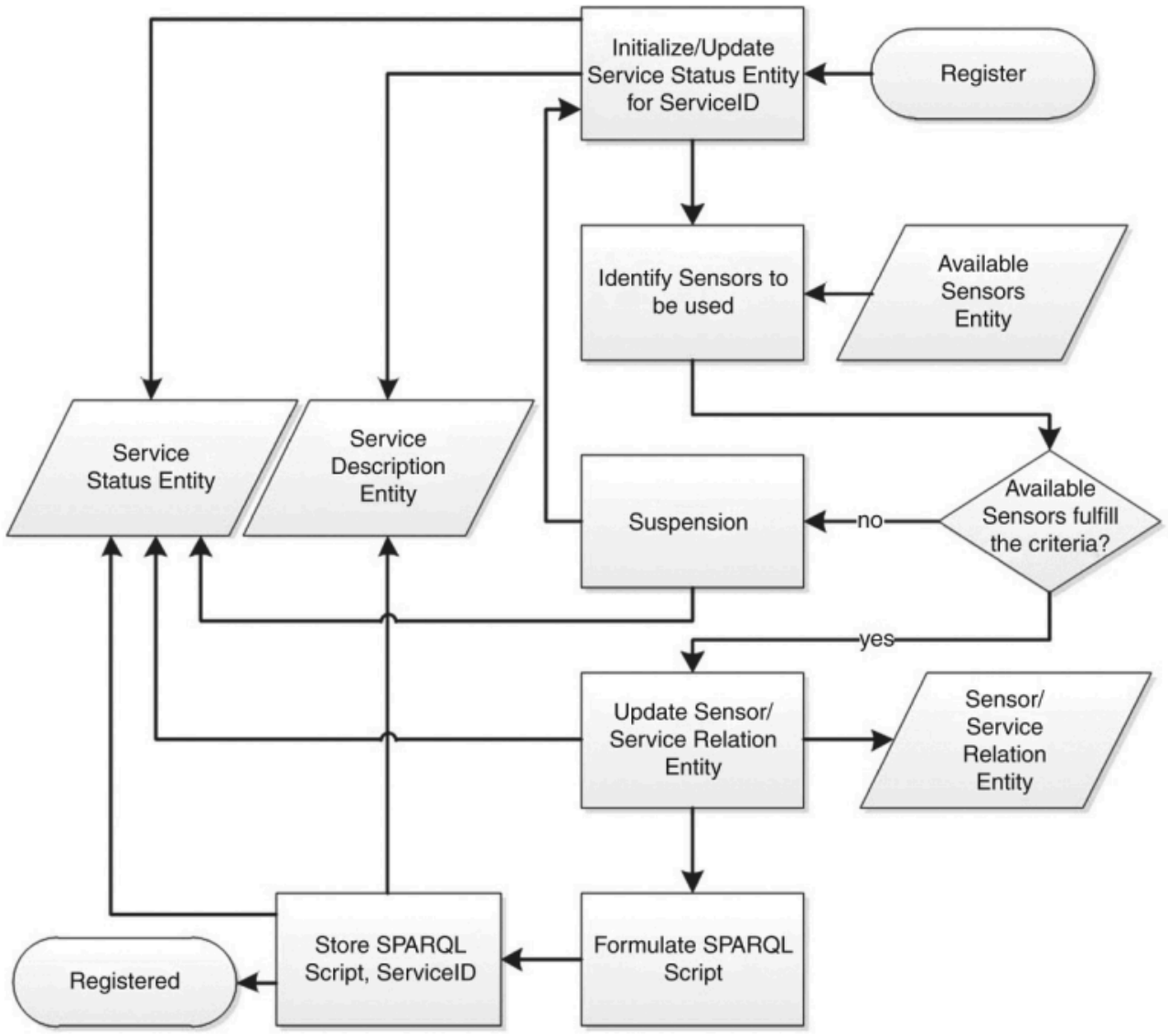

FIGURE 2.3 "Register Service" Process Flowchart

Likewise, Fig. 2.4 illustrates the process of updating the resources associated with a given service. As already outlined, such an update process is particularly important when it comes to dealing with IoT services that entail mobile sensors and ICOs, that is, sensors and ICOs whose location is likely to change within very short timescales (such as mobile phones and UAVs). In such cases, the Update Resources process could regularly check the availability of mobile sensors and their suitability for the registered service whose resources are updated. The workflow in Fig. 2.4 assumes that the list of mobile sensors is known to the service (ie, the sensors' semantic annotations indicate whether a sensor is mobile or not). Even though the process/functionality of updating resources is associated with the need to identify the availability and suitability of mobile sensors, in principle the Update process can be used to update the whole list of resources that contribute to the

given service. Such functionality helps the OpenIoT platform in dealing with the volatility of IoT environments, where sensors and ICOs may dynamically join or leave.

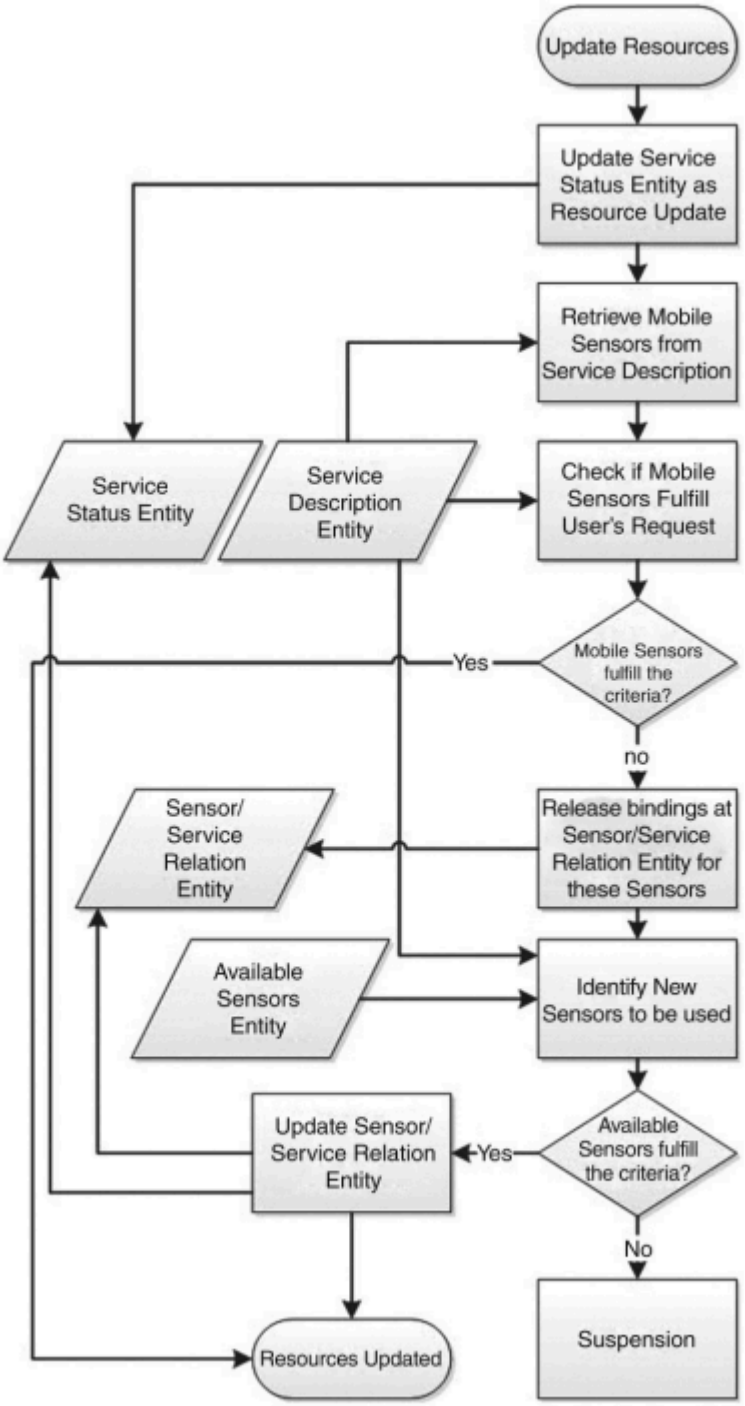

FIGURE 2.4 "Update Resources" Service Flowchart

Scheduling and Resource Management

The OpenIoT scheduler enables the availability and provision of accurate information about the data requested by each service, as well as about the sensors and ICOs that will be used in order to deliver these data. Hence, a wide range of different resource management and optimization algorithms can be implemented at the scheduler component of the OpenIoT architecture. Furthermore, the envisaged scheduling at the local (ie, sensor

middleware) level enables resource optimization at the sensor data acquisition level (ie, at the edges of the OpenIoT infrastructure).

In terms of specific optimizations that can be implemented over the introduced IoT/cloud infrastructure are optimization techniques that have their roots in the WSN literature, where data management is commonly applied as a means to optimize the energy efficiency of the network. In particular, the scheduler components (at both the global and the local levels) maintain information that could optimize the use of the network on the basis of aggregate operations, such as the aggregation of queries to the sensor cloud. Furthermore, a variety of in-network processing and data management techniques can be implemented in order to optimize processing times and/or reduce the required access to the sensor network. The criteria for aggregating queries and their results could be based on common spatial regions (ie, data aggregation based on sensors residing in geographical regions of high interest).

Another class of optimizations that are empowered by the introduced scheduling approach are caching techniques, which typically reduce network traffic, enhance the availability of data to the users (sink), and reduce the cost of expensive cloud-access operations (eg, I/O operations to public clouds). The caching concept involves maintaining sensor (data streams) data to a cache memory in order to facilitate fast and easy access to them. In the case of OpenIoT, caching techniques could obviate the need to execute the results of previously executed SPARQL queries.

Validating Applications and Use Cases

The overall scheduling infrastructure has also been validated through the development of proof-of-concept IoT applications, which comprise multiple IoT services that have been integrated based on the OpenIoT infrastructure. One of the proof-of-concept applications falls in the wider realm of smart-city applications, aimed at providing a set of smart services within a student campus in Karlsruhe, Germany.

The second use-case concerns the implementation of IoT services for manufacturing, and, more specifically, for the printing and packing industry, with emphasis on the production of boxes. The IoT services support key production processes in this industry, such as printing on paper sheets and die-cutting (for perforation of the sheets), as well as gluing and folding the pieces of a box. A variety of sensors are employed to facilitate production-line automation and quality-control checks, including laser sensors, high-speed 1D/2D barcode verification cameras, weight sensors, contrast and color sensors (for marking code identification), as well as ultrasonic sensors (for measuring heights and material-reel diameters). In this environment, the OpenIoT infrastructure is used to enable the dynamic

on-demand formulation, calculation, and visualization of KPIs (Key Performance Indicators) about the manufacturing processes. Interesting KPIs include, for example: (1) in the area of materials consumption, the rate of consumption and how much scrap is produced; (2) in the area of machine performance, how fast each machine is working, what is the rate of product/shipping container production, and the overall efficiency of the machines; (3) in the area of labor activity and performance, how much time is spent setting up/repairing the machine; and (4) in the area of machine operation, an interesting KPI relates to tracking the time that machines spend in their various modes (ie, setup/repair/idle/operation).

To this end, high-level events captured, based on the processing of the aforementioned sensors, are announced as virtual sensors to the W3C SSN directory through the X-GSN middleware. In particular, KPI calculations are implemented as a set of X-GSN virtual sensors, and, accordingly, are published to the sensor cloud and made available for semantic querying.

Future Research Directions

An open source implementation of the introduced scheduling concepts is available as part of the OpenIoT Open Source Project. We expect this chapter to motivate the open-source community toward providing other implementations of scheduling concepts for IoT services, including their integration with cloud computing infrastructures. Such concepts could deal with caching of IoT service requests and response data, taking into account the spatiotemporal characteristics of the respective IoT queries. In terms of the OpenIoT middleware infrastructure, future research work includes the development of tools for visualizing IoT resources, as part of the Integrated Development Environment (IDE) of the OpenIoT project.

⇒ Device / Cloud Collaboration Framework for Intelligence Applications

Introduction

Cloud computing is now an established computing paradigm that offers on-demand computing and storage resources to the users who cannot afford the expenditure on computing equipment and management workforce. This computing paradigm first led to the notable commercial success of Amazon's EC2 [1] and Microsoft's Azure [2]. These companies have adopted the business model of renting out their virtualized resources to the public. More recently, Google and Facebook are now utilizing their data-center-based clouds to internally run machine-learning algorithms based on the large volume of data collected from their users.

Background and Related Work

Public cloud-vendors have attracted institutions who have little incentive to pay the upfront cost of IT infrastructure, especially when the applications and services they host do not require high compute and storage capacity. However, once their applications and services become popular, the demand for higher compute and storage capacity may suddenly soar. This is the point when these institutions may not feel that public cloud is cost-effective anymore, and consider opting for a different cloud computing model. Some organizations have considered forming a community that shares cloud infrastructure and management resources. In this model, the community members can share their cloud resources with one another. In addition, the shared cloud resources can be offered to nonmembers to drive extra revenue. Recently, companies such as IBM and Samsung have teamed up to push for community cloud-computing initiatives.

Device / Cloud Collaboration Framework

1. Powerful Smart Mobile Devices

In 2014, more than 1 billion smartphones were sold worldwide, and more than 2 billion consumers are expected to get a smartphone by 2016 [10]. Smartphonesa nowadays have enough computing capacity to process various computing tasks. However, the device usage can be constrained by limited battery life and network connectivity. Therefore, we can consider utilizing the highly available cloud resources in addition to the device.

In Fig. 3.1, we have illustrated a high-level layout of our device/cloud collaboration framework. In the following section, we will explain how the framework functions to address the aforementioned concerns on performance and privacy.

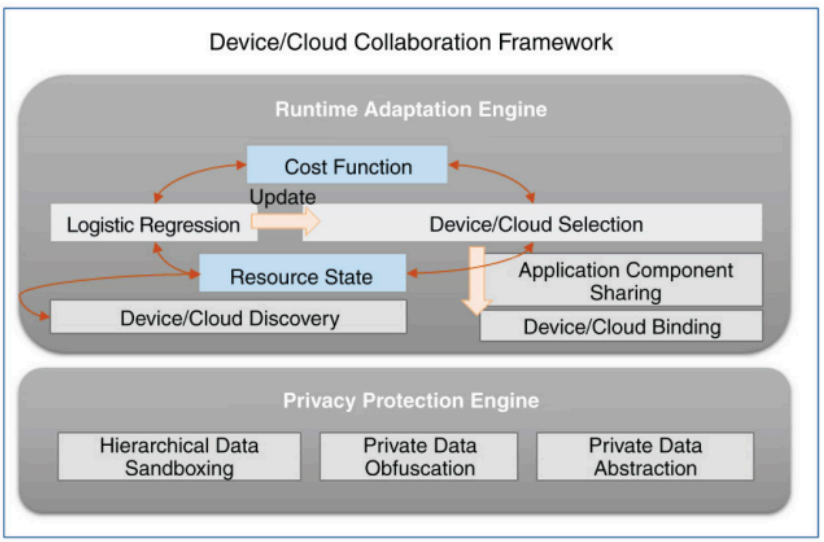

FIGURE 3.1 High-Level Layout of the Device/Cloud Collaboration Framework

2. Runtime Adaptation Engine

Given a computation task, our framework first faces a problem of choosing the entity to execute the task. Suppose we are given a task of processing a query issued over voice, and assume that we have a lightweight mobile version of a voice-query processing engine that is embedded in a smartphone. This mobile engine can answer the given query without the cost of transferring the voice data to the cloud over the network. However, it will consume the limited battery life of the device, and/or the accuracy of the result may not necessarily be as good as that of the cloud-based query processing engine, which runs on resources with higher compute capacity. If the lightweight voice-query engine returns a poor result, then the user may have to issue the query redundantly to the cloud-based query processing engine with the hope of getting a better result. This may hurt the overall quality of experience (QoE). This calls for a decision mechanism that automatically selects a better agent that can execute a given task. With the automatic selection process in place, users do not have to worry about going through extra interaction cycles for determining where to run a job. Note that the Runtime Adaptation Engine (RAE) sits at the core of our framework, as shown in Fig. 3.1. The RAE maintains a list of available devices and cloud to utilize Device/Cloud Discovery, and monitors the state of their available resources. The RAE employs a Logistic Regression algorithm to learn the most cost-effective policy for distributing tasks among devices and cloud, given the resource state. The mechanism of the RAE is actually an autonomous agent, which can be deployed on each device and cloud. RAEs communicate with each other to transparently share the resource state for determining the ways to distribute a given workload.

3. Privacy Protection Solution

So far, we have focused on the aspects of the device/cloud collaboration framework that are concerned with performance and efficiency. Now, we turn our attention to the privacy-protection problem. Suppose we want to provide a location-based service to the users. To provide this service, location data such as the visited GPS coordinates, the point of interest (POI), and the time of visit have to be collected first. Based on these location data, the mobility pattern and the interest of individual users can be inferred. However, these data contain personal information. Hence, for these data, we can ask the user to decide whether to transfer them to cloud or not when accepting the location-based service. Based on the decision made by the user, our framework can assess the expected service-quality for the users. Suppose a user

wants to transfer to the cloud a blurred location log with entries that are simply labeled either or . Given this log with little detail, it is difficult to expect a service provider to offer a useful location-based recommendation. In such a case, our framework warns the user that disallowing the sharing of detailed private information would result in poor service quality. Our framework employs the technology of protecting privacy by sandboxing hierarchically organized application-data. This technology, implemented in the Hierarchical Data Sandboxing module (in Fig. 3.1), supports the user to explicitly specify a group of data in the hierarchy to be shared with the cloud or not. The group of data that is set to be kept only within a device will be protected by sandboxing.

Applications of Device/Cloud Collaboration

1. Context Aware Proactive Suggestion

Based on the personal data collected on each mobile device, we have devised Proactive Suggestion (PS), an application that makes context-aware recommendations. In Fig. 3.2, the individual components of the PS are laid out. Analytics engines of PS produce hierarchical personal data that are interdependent to each other. Raw data such as GPS coordinates, call logs, application usage, and search queries are fed to a Cooccurrence Analysis engine, which is responsible for identifying activities that occurred at the same time. For example, the cooccurrence analysis engine may recognize that a user listens to live streaming music while walking in the park. Given such cooccurrence data, the Sequence Mining engine can infer causal relationships between personal activities that occurred over time. The recognized sequential patterns can be fed into the Predictive Analysis engine to assess the probability of a particular activity taking place in a certain context.

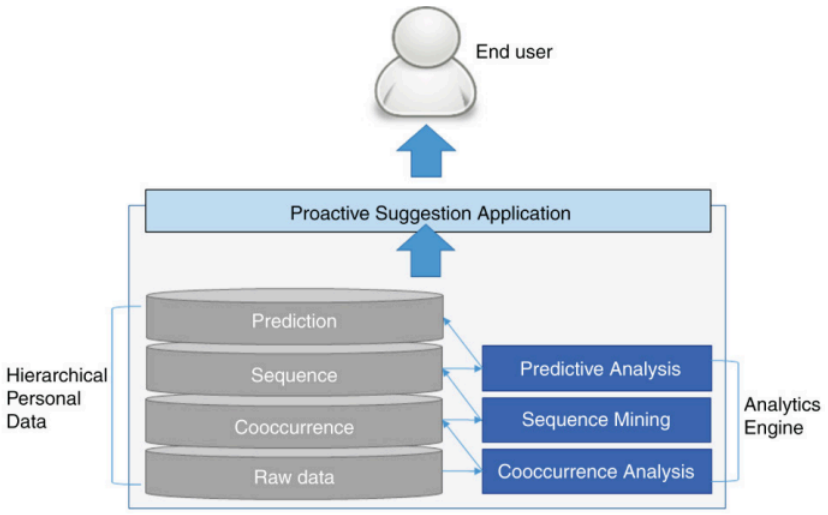

FIGURE 3.2 High-Level Layout of the Core Components for the Proactive Suggestion Application

2. Semantic QA Cache

Semantic QA cache is a mobile application that retrieves answers to a given query from the cache filled with answers to the semantically similar queries issued in the past. Semantic QA cache can be useful when there is no Internet connectivity or when the user is not in favor of transferring private queries to the cloud. Fig. 3.4 illustrates how the semantic QA cache is managed. Semantic QA cache returns a list of similar queries and the associated answers. Semantic QA cache constantly updates ranking function based on the word-translation table as explained in [20]. The ranking function measures the similarity between a newly issued query and the queries measured in the past.

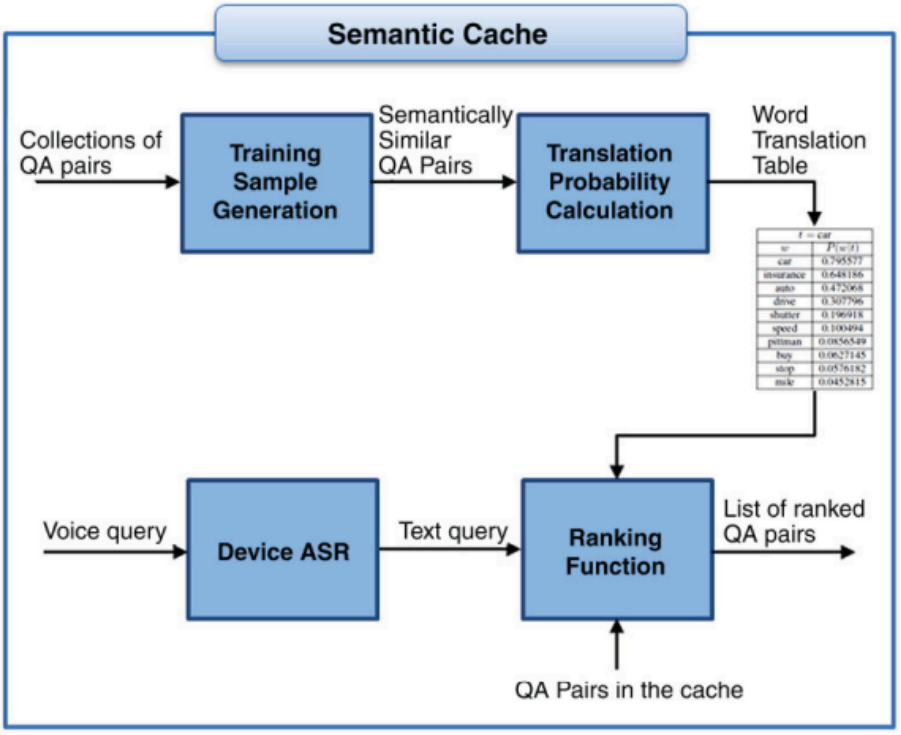

FIGURE 3.4 Illustrations of the Technique to Cluster Semantically Similar QA Pairs for Retrieving an Answer for a Newly Given Query Without Asking the QA Engine on the Cloud Side

Note that we have adapted the framework to compute the probability of the on-device semantic QA cache to answer a given query correctly. If the probability is high enough, the Device/Cloud Selection module will take the risk of looking up the semantic QA cache for an answer. If the cache does not return the right answer and forces the user to ask the cloud again, then our framework will adjust the probability accordingly.

3. Image and Speech Recognition

Automatically recognizing images and speech can greatly enhance a user's experience in using applications. For example, with automatic image recognition, photos taken by a user can be automatically tagged with metadata and catalogued

more easily. Similar to Amazon's Firefly [21], we have developed an application called Watch&Go, which lets users obtain detailed information about a product upon taking a photograph. Fig. 3.6 shows the snapshot of Watch&Go that guides users to properly focus on some electronics products, and automatically retrieve information such as type, vendor, model name, and the result of social sentiment about the product.

Practicality of these recognition applications has greatly improved, thanks to the recent advancement of Deep Learning (DL). The DL follows the approach of learning the correlation between the parameters across multiple layers of perceptron [22]. However, DL model training methods usually suffer a slow learning curve compared to the other conventional machine-learning methods. Although it is generally believed that the larger DL model improves the recognition accuracy through a set of well-refined training data, it has been challenging to acquire adequate parameters when we train multiple layers at the same time

Future Work

Various ways of componentizing a given intelligence application come with different trade-offs. A more fine-grained componentization, as shown in the PS application, may yield more efficient task distribution among neighboring devices and cloud. Also, fine-grained privacy-protection policies can be applied to these components. However, computing more efficient task distribution incurs additional overhead. It may cost relatively lower overhead to determine execution entities for the coarse-grained. But it can be nontrivial to port a cloud-side coarse-grained component to a resource-limited device when it is deemed necessary.

Module - 02

⇒ Introduction to Fog Computing

Internet of Things (IoT) environments consist of loosely connected devices that are connected through heterogeneous networks. In general, the purpose of building such environments is to collect and process data from IoT devices in order to mine and detect patterns, or perform predictive analysis or optimization, and finally make smarter decisions in a timely manner. Data in such environments can be classified into two categories.

- Little Data or Big Stream: transient data that is captured constantly from IoT smart devices
- Big Data: persistent data and knowledge that is stored and archived in centralized cloud storage

IoT environments, including smart cities and infrastructures, need both Big Stream and Big Data for effective real-time analytics and decision making. This can enable real-time cities that are capable of real-time analysis of city infrastructure and life, and provides new approaches for governance.

Motivation Scenario

A recent analysis [3] of Endomondo application, a popular sport-activity tracking application, has revealed a number of remarkable observations. The study shows that a single workout generates 170 GPS tuples, and the total number of GPS tuples can reach 6.3 million in a month's time. With 30 million users (as shown in Fig. 4.1), the study shows that generated data flows of Endomondo can reach up to 25,000 tuples per second. Therefore, one can expect that data flows in real-time cities with many times more data sources—GPS sensors in cars to air- and noise-pollution sensors—can easily reach millions of tuples per second. Centralized cloud servers cannot deal with flows with such velocity in real time. In addition, a considerable numbers of users, due to privacy concerns, are not comfortable to transfer and store activity-track-data into the cloud, even if they require a statistical report on their activities. This motivates the need for an alternative paradigm that is capable of bringing the computation to more computationally capable devices that are geographically closer to the sensors than to the clouds, and that have connectivity to the Internet.

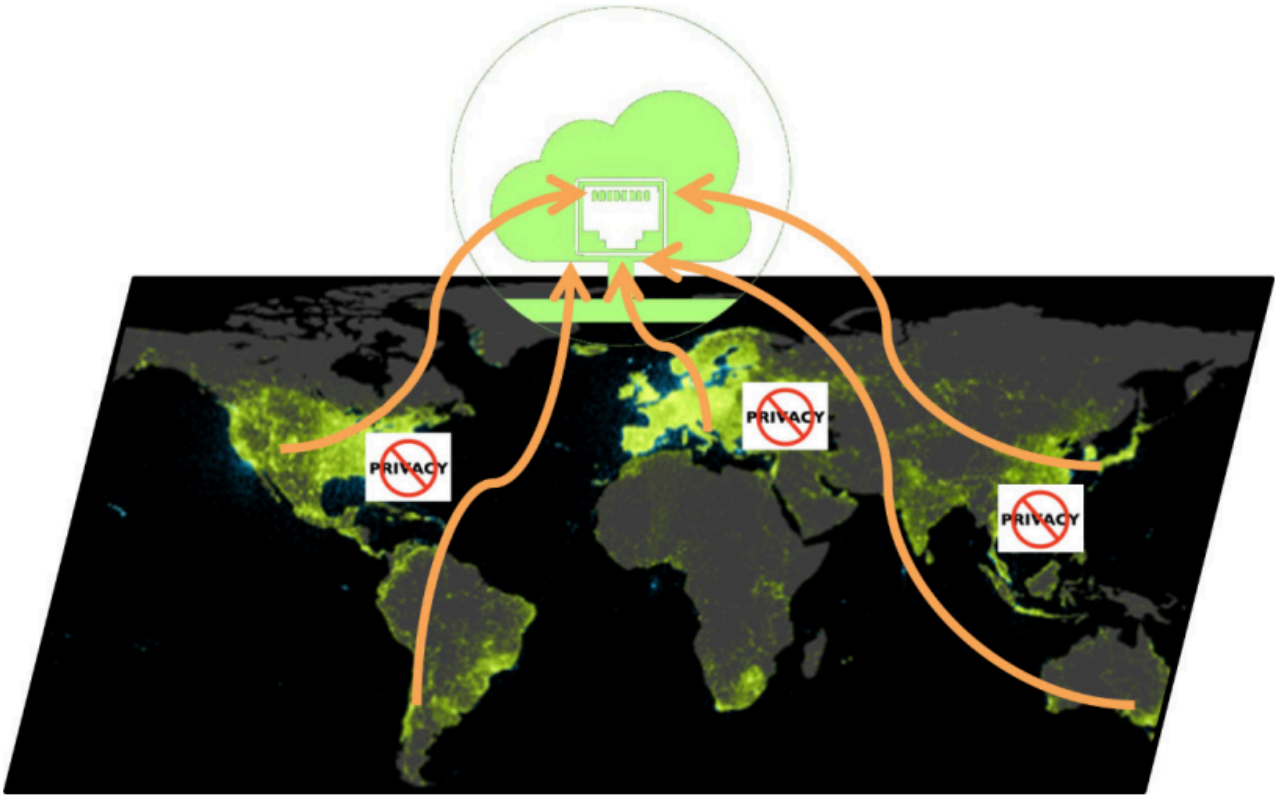

FIGURE 4.1 Endomondo has 30 Million Users Around the Globe, Generating 25,000 Records per Second

Definition and Characteristics

We define Fog computing as a distributed computing paradigm that fundamentally extends the services provided by the cloud to the edge of the network. It facilitates management and programming of compute, networking, and storage services between data centers and end devices. Fog computing essentially involves components of an application running both in the cloud as well as in edge devices between sensors and the cloud, that is, in smart gateways, routers, or dedicated fog devices.

Fog computing supports mobility, computing resources, communication protocols, interface heterogeneity, cloud integration, and distributed data analytics to address requirements of applications that need low latency with a wide and dense geographical distribution. Advantages associated with Fog computing including the following:

- Reduction of network traffic: Cisco estimates that there are currently 25 billion connected devices worldwide, a number that could jump to 50 billion by 2020. The billions of mobile devices such as smart phones and tablets already being used to generate, receive, and send data make a case for putting the computing capabilities closer to where devices are located, rather than having all data sent over networks to central data centers. Depending on the configured frequency, sensors may collect data every few seconds. Therefore, it is neither efficient nor sensible to send all of

this raw data to the cloud. Hence, fog computing benefits here by providing a platform for filter and analysis of the data generated by these devices close to the edge, and for generation of local data views.

- Suitable for IoT tasks and queries: With the increasing number of smart devices, most of the requests pertain to the surroundings of the device. Hence, such requests can be served without the help of the global information present at the cloud. For example, the aforementioned sportstracker application Endomondo allows a user to locate people playing a similar sport nearby
- Low-latency requirement: Mission-critical applications require real-time data processing. Some of the best examples of such applications are cloud robotics, control of fly-by-wire aircraft, or antilock brakes on a vehicle. For a robot, motion control depends on the data collected by the sensors and the feedback of the control system. Having the control system running on the cloud may make the sense-process-actuate loop slow or unavailable as a result of communication failures. This is where fog computing helps, by performing the processing required for the control system very close to the robots—thus making real-time response possible.
- Scalability: Even with virtually infinite resources, the cloud may become the bottleneck if all the raw data generated by end devices is continually sent to it. Since fog computing aims at processing incoming data closer to the data source itself, it reduces the burden of that processing on the cloud, thus addressing the scalability issues arising out of the increasing number of endpoints.

Reference Architecture

Fig. 4.3 presents a reference architecture for fog computing. In the bottommost layer lays the end devices (sensors), as well as edge devices and gateways. This layer also includes apps that can be installed in the end devices to enhance their functionality. Elements from this layer use the next layer, the network, for communicating among themselves, and between them and the cloud. The next layer contains the cloud services and resources that support resource management and processing of IoT tasks that reach the cloud. On top of the cloud layer lays the resource management software that manages the whole infrastructure and enables quality of Service to Fog Computing applications. Finally, the topmost layer contains the applications that leverage fog computing to deliver innovative and intelligent applications to end users.

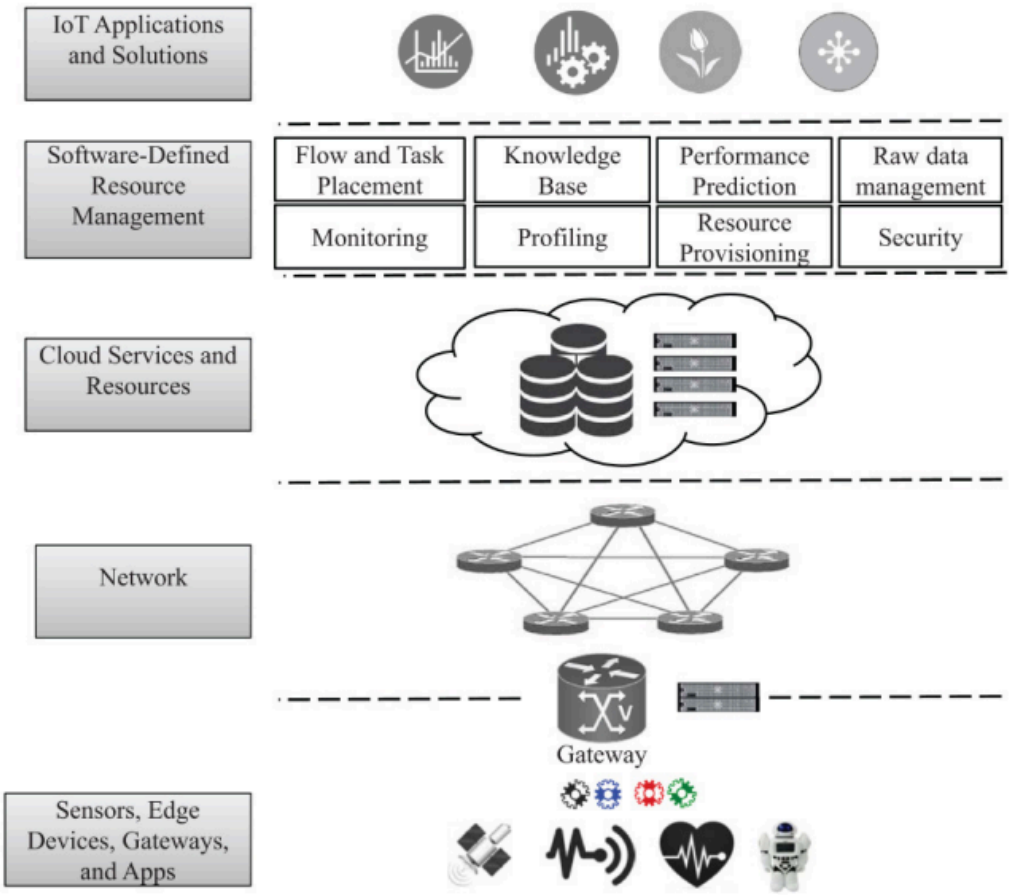

FIGURE 4.3 Fog Computing Reference Architecture

Looking inside the Software-Defined Resource Management layer, it implements many middlewarelike services to optimize the use of the cloud and Fog resources on behalf of the applications. The goal of these services is to reduce the cost of using the cloud at the same time that performance of applications reach acceptable levels of latency, by pushing task execution to Fog nodes. This is achieved with a number of services working together, as follows.

- Flow and task placement: This component keeps track of the state of available cloud, Fog, and network resources (information provided by the Monitoring service) to identify the best candidates to hold incoming tasks and flows for execution. This component communicates with the Resource-Provisioning service to indicate the current number of flows and tasks, which may trigger new rounds of allocations if deemed too high.
- Knowledge Base: This component stores historical information about application demands and resource demands that can be leveraged by other services to support their decision-making process.
- Performance Prediction: This service utilizes information of the Knowledge- Base service to estimate the performance of available cloud resources. This information is used by the ResourceProvisioning service to decide the amount of resources to be

provisioned, in times where there are a large number of tasks and flows in use or when performance is not satisfactory.

- Raw-Data Management: This service has direct access to the data sources and provides views from the data for other services. Sometimes, these views can be obtained by simple querying (eg, SQL or NOSQL REST APIs), whereas other times more complex processing may be required (eg, MapReduce).
- Monitoring: This service keeps track of the performance and status of applications and services, and supplies this information to other services as required.
- Profiling: This service builds resource- and application-profiles based on information obtained from the Knowledge Base and Monitoring services.
- Resource Provisioning: This service is responsible for acquiring cloud, Fog, and network resources for hosting the applications. This allocation is dynamic, as the requirements of applications, as well as the number of hosted applications, changes over time.

Applications

1. Healthcare

Cao et al. propose FAST, a fog-computing assisted distributed analytics system, to monitor fall for stroke patients. The authors have developed a set of fall-detection algorithms, including algorithms based on acceleration measurements and time-series analysis methods, as well as filtering techniques to facilitate the fall-detection process. They designed a real-time fall-detection system based on fog computing that divides the fall-detection task between edge devices and the cloud. The proposed system achieves a high sensitivity and specificity when tested against real-world data. At the same time, the response time and energy consumption are close to the most efficient existing approaches.

Another use of fog computing in healthcare has been brought out by Stantchev et al. They proposed a three-tier architecture for a smart-healthcare infrastructure, comprised of a role model, layered-cloud architecture, and a fog-computing layer, in order to provide an efficient architecture for healthcare and elderly-care applications.

2. Augmented reality

Augmented reality applications are highly latency-intolerant, as even very small delays in response can damage the user experience. Hence, fog computing has the potential to become a major player in the augmented reality domain. Zao et al. [6] built an Augmented Brain Computer Interaction Game based on Fog Computing and

Linked Data. When a person plays the game, raw streams of data collected by EEG sensors are generated and classified to detect the brain state of the player. Brain-state classification is among the most computationally heavy signal-processing tasks, but this needs to be carried out in real time. The system employs both fog and cloud servers, a combination that enables the system to perform continuous real-time brain-state classification at the fog servers, while the classification models are tuned regularly in the cloud servers, based on the EEG readings collected by the sensors

3. Caching and Preprocessing

Zhu et al. [8] discuss the use of edge servers for improving web sites' performance. Users connect to the Internet through fog boxes, hence each HTTP request made by a user goes through a fog device. The fog device performs a number of optimizations that reduces the amount of time the user has to wait for the requested webpage to load. Apart from generic optimizations like caching HTML components, reorganizing webpage composition, and reducing the size of web objects, edge devices also perform optimizations that take user behavior and network conditions into account. For example, in case of network congestion, the edge device may provide low resolution graphics to the user in order to reach acceptable response times. Furthermore, the edge device can also monitor the performance of the client machines, and, depending on the browser rendering times, send graphics of an appropriate resolution.

One of the major advantages of fog computing is linking IoT and cloud computing. This integration is not trivial and involves several challenges. One of the most important challenges is data trimming. This trimming or pre-processing of data before sending it to the cloud will be a necessity in IoT environments because of the huge amount of data generated by these environments.

⇒ TinyOS - NesC

TinyOS is an open-source operating system designed for embedded systems and specifically for wireless sensor networks (WSNs). It is lightweight, modular, and event-driven, making it ideal for resource-constrained devices typical in the Internet of Things (IoT). NesC (Network Embedded Systems C) is the programming language developed specifically for TinyOS. NesC extends the C language with components and event-driven programming concepts, enabling efficient development and deployment of TinyOS applications.

Key Features of TinyOS

1. **Lightweight and Modular**:
   TinyOS's modular structure allows developers to include only the necessary components for an application, reducing memory and computational requirements.
2. **Event-Driven Execution**:
   TinyOS operates on an event-driven model instead of a traditional multithreaded approach. Events and tasks are scheduled for execution based on priorities, minimizing resource contention.
3. **Concurrency Management**:
   Tasks in TinyOS are non-preemptive but can interact with asynchronous events. This model simplifies concurrency management and avoids common issues like race conditions.
4. **High Energy Efficiency**:
   Designed for WSNs, TinyOS emphasizes low-power operation to extend the lifespan of devices powered by limited energy sources such as batteries.

NesC: The Programming Framework for TinyOS

NesC is a component-based programming language specifically designed for embedded systems and TinyOS. It builds upon C by introducing new constructs to facilitate modular programming and efficient hardware interaction.

Core Concepts of NesC:

1. Components:
   - Components are the building blocks of NesC applications. They are of two types:
     - Modules: Contain the application logic, including implementation details of interfaces.
     - Configurations: Connect and wire different components together.
   - Each component has defined interfaces that specify the services it provides and the services it requires.
2. Interfaces:
   - Interfaces define a contract between components. They consist of commands (provided services) and events (required services).

    - Commands are invoked by the user of the interface, while events are implemented by the user and triggered by the component.
3. Event-Driven Model:
    - NesC supports asynchronous event handling, which is critical for applications with real-time constraints, such as sensor networks.
    - Events like data reception, timer expirations, or hardware interrupts are handled efficiently without needing a heavy multitasking framework.
4. Static Memory Allocation:
    - To minimize overhead, NesC uses static memory allocation. This is especially useful for embedded systems where dynamic memory management might lead to fragmentation and inefficiency.
5. Concurrency Control:
    - NesC incorporates mechanisms to detect and avoid data races during compilation, making it safer for concurrent operations.

⇒ Programming Framework for Internet of Things

Introduction

IoT devices are generally characterized as small things in the real world with limited storage and processing capacity, which may not be capable of processing a complete computing activity by themselves. They may need the computational capabilities of Cloud-based back-ends to complete the processing tasks and web-based front-ends to interact with the user. The Cloud infrastructure complements the things [1], by supporting device virtualization, availability, provisioning of resources, data storage, and data analytics. The IoT by its nature will extend the scope of Cloud computing to the real world in a more distributed and dynamic way. With new opportunities, IoT also puts forth a major set of challenges for IoT application developers. Heterogeneity and the volume of data generated are two of the biggest concerns. Heterogeneity spans through hardware, software, and communication platforms. The data generated from these devices are generally in huge volume, are in various forms, and are generated at varying speeds. Since IoT applications will be distributed over a wide and varying geographical area, support for corrective and evolutionary maintenance of applications will determine the feasibility of application deployment.

Background

During the lifecycle of IoT applications, the footprint of an application and the cost of its language runtime play a huge role in the sustainability of an application. C has been used predominantly in embedded applications development due to its performance; moreover, it can occupy the same position in IoT programming too. Further, the choice of communication protocols also has a huge implication in the cost of IoT applications on devices. Remote Procedure Calls (RPC), Representational state transfer (REST), and Constrained Application Protocol (CoAP) are some of the communication methods that are being currently incorporated into IoT communication stacks. A complete programming framework in a distributed environment requires not only a stable computing language such as C, but also a coordination language that can manage communications between various components of an IoT ecosystem. An explicit coordination language can tackle many of the challenges. It can manage communication between heterogeneous devices, coordinate interaction with the Cloud and devices, handle asynchronous data arrival, and can also provide support for fault tolerance.

Embedded Device Programming Languages

Although there are various programming languages in the embedded programming domain, the vast majority of projects, about 80%, are either implemented in C and its flavors, or in a combination of C and other languages such as C++. Some of the striking features of C that aid in embedded development are performance, small memory foot-print, access to low-level hardware, availability of a large number of trained/experienced C programmers, short learning curve, and compiler support for the vast majority of devices. The ANSI C standard provides customized support for embedded programming. Many embedded C-compilers based on ANSI C usually:

- support low-level coding to exploit the underlying hardware,
- support in-line assembly code,
- flag dynamic memory-allocation and recursion,
- provide exclusive access to I/O registers,
- support accessing registers through memory pointers, and
- allow bit-level access.

1. NesC

    nesC is a dialect of C that has been used predominantly in sensor-nodes programming. It was designed to implement TinyOS [6], an operating system for

sensor networks. It is also used to develop embedded applications and libraries. In nesC, an application is a combination of scheduler and components wired together by specialized mapping constructs. nesC extends C through a set of new keywords. To improve reliability and optimization, nesC programs are subject to whole program analysis and optimization at compile time.

2. Keil C

Keil C is a widely used programming language for embedded devices. It has added some key features to ANSI C to make it more suitable for embedded device programming. To optimize storage requirements, three types of memory models are available for programmers: small, compact, and large. New keywords such as alien, interrupt, bit, data, xdata, reentrant, and so forth, are added to the traditional C keyword set. Keil C supports two types of pointers:

- generic pointers: can access any variable regardless of its location
- memory-specific pointers: can access variables stored in data memory

3. Dynamic C

Some key features in Dynamic C [8] are function chaining and cooperative multitasking. Segments of code can be distributed in one or more functions through function chaining. Whenever a function chain executes, all the segments belonging to that particular chain execute. Function chains can be used to perform data initialization, data recovery, and other kinds of special tasks as desired by the programmer. The language provides two directives #makechain, #funcchain and a keyword segchain to manage and define function chains.

Message Passing Devices

In this section, we review some of the communication paradigms and technologies such as RPC, REST, and CoAP that can be used in resource-constrained environments.

1. RPC

RPC is an abstraction for procedural calls across languages, platforms, and protection mechanisms. For IoT, RPC can support communication between devices as it implements the request/response communication pattern. Typical RPC calls exhibit synchronistic behavior. When RPC messages are transported over the network, all the parameters are serialized into a sequence of bytes. Since serialization of primitive data types is a simple concatenation of individual bytes, the

serialization of complex data structures and objects is often tightly coupled to platforms and programming languages.

Lightweight Remote Procedure Call (LRPC) was designed for optimized communication between protection domains in the same machine, but not across machines. Embedded RPC (ERPC) in Marionetteuses a fat client such as a PC and thin servers such as nodes architecture.

2. REST

Roy Fielding in his PhD thesis proposed the idea of RESTful interaction for the Web. The main aim of the REST was to simplify the web-application development and interaction. It leverages on the tools available on the Internet and stipulates the following constraints on application development:

- Should be based on client-server architecture and the servers should be stateless
- Support should be provided for caching at the client side
- The interface to servers should be generic and standardized (URI)
- Layering in the application architecture should be supported and each of the layers shall be independent
- Optional code-on demand should be extended to clients having the capability

The authors in [16] have highlighted that the previous constraints and principles bring in many advantages to the distributed applications: scalability, loose coupling, better security, simple addressability, connectedness, and performance. Further, they compare RPC with REST for the same qualitative measures, and argue that RESTful approaches are always better for each of the previous measures. One more advantage of RESTful components is that they can be composed to produce mashups, giving rise to new components which are also RESTful.

3. CoAP

Since HTTP/TCP stack is known to be resource demanding on constrained devices, protocols such as Embedded Binary HTTP (EBHTTP) and Compressed HTTP Over PAN (CHoPAN) have been proposed. However, the issue of reliable communications still remains a concern. The IETF work group, Constrained RESTful Environments (CoRE), has developed a new web-transfer protocol called Constrained Application Protocol (CoAP), which is optimized for constrained power and processing capabilities of IoT. Although the protocol is still under standardization, various implementations are in use. CoAP in simpler terms is a two-layered protocol: a messages layer, interacting with the UDP, and another layer for request/response

interactions using methods and response code, as done in HTTP. In contrast to HTTP, CoAP exchanges messages asynchronously and uses UDP. The CoAP has four types of messages: Acknowledgement, Reset, Confirmable (CON), and NonConfirmable (NON). The non-confirmable messages are used to allow sending requests that may not require reliability. Reliability is provided by the message layer and will be activated when Confirmable messages are used. The Request methods are: GET, POST, PUT, and DELETE of HTTP.

Coordination Languages

Carriero and Gelernter argue in that a complete programming model can be built by combining two orthogonal models—a computation model and a coordination model. The computation model provides the computational infrastructure and programmers can build computational activity using them, whereas the coordination model provides the support for binding all those computational activities together. They argue that a computational model supported by languages such as C, by themselves cannot provide genuine coordination support among various computing activities. This observation is more relevant in IoT–Cloud programming, wherein there are numerous distributed activities which have to be coordinated in a reliable and fault-tolerant manner. Coordination can be seen through two different perspectives: (1) based on centralized control, named as Orchestration and (2) based on distributed transparent control, named as Choreography. The W3C's Web services choreography working group defines Choreography as "the definition of the sequences and conditions under which multiple cooperating independent agents exchange messages in order to perform a task to achieve a goal state." Orchestration is seen as "the definition of sequence and conditions in which one single agent invokes other agents in order to realize some useful function." There are many languages that provide Choreography and Orchestration support. We briefly review some of the features in coordination languages such as Linda, eLinda, Orc, and Jolie.

Polyglot Programming

Polyglot programming is also called multilingual programming. It is an art of developing simpler solutions by combining the best possible solutions using different programming languages and paradigms. This is based on the observation that there is no single programming paradigm or a programming language which can suit all the facets of modern-day programming or software requirements. It is also called poly-paradigm programming (PPP), to appreciate the fact that many modern-day software combines a

subset of imperative, functional, logical, object-oriented, concurrent, parallel, and reactive programming paradigms.

One of the oldest examples of polyglot programming is Emacs, which is a combination of parts written in both C and eLisp (dialect of Lisp). Web applications are generally based on three-tier architecture to promote loose coupling and modularity, and they are also a representation of polyglot software systems. Polyglot programming has been observed to have increased programmer productivity and software maintainability in web development.

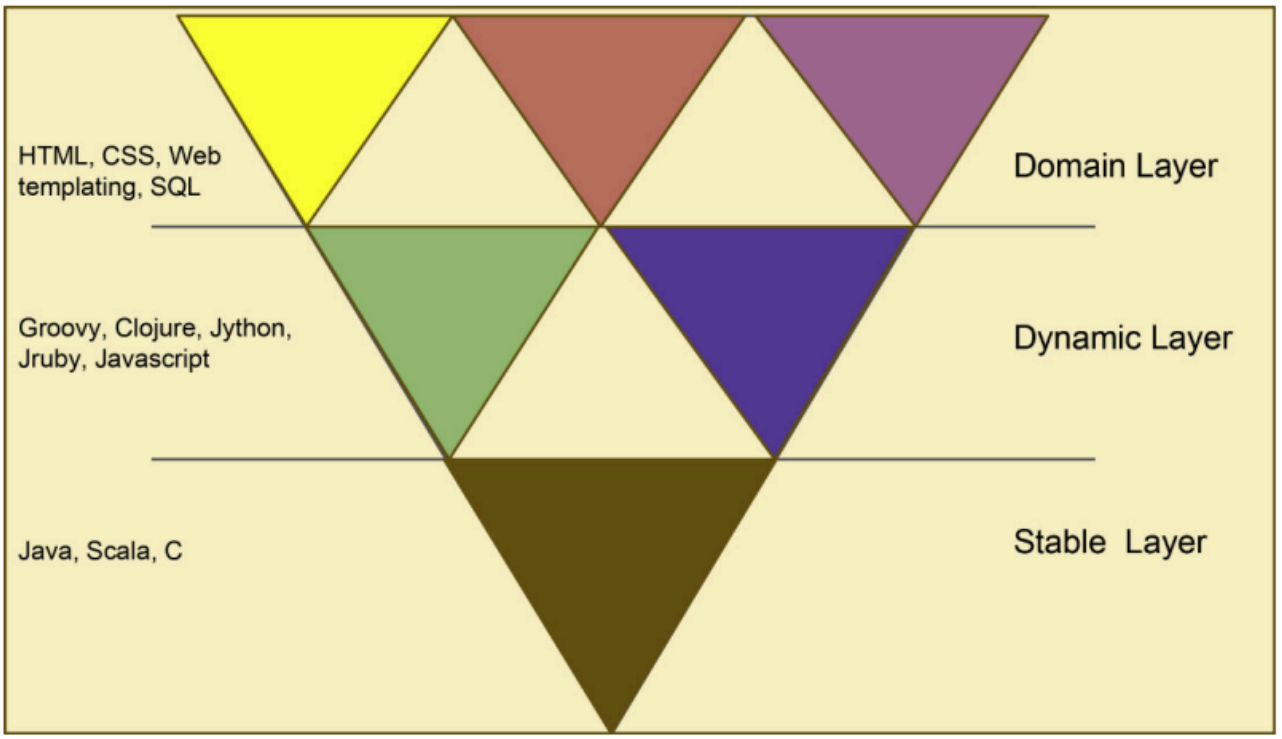

FIGURE 5.2 Inverse Pyramid for Polyglot Programming

In a Polyglot programming environment, the platform used for the integration, and the different programming languages supported by the given platform are the two essential aspects. An Inverse pyramid [29,30] can be used to categorize the programming languages in a polyglot software system. The Inverse pyramid has three layers: stable, dynamic, and domain, as shown in Fig. 5.2.

IoT Programming Approaches

1. Node-Centric Programming

   Here, every aspect of application development, communication between nodes, collection and analysis of sensor data, and issuing of commands to actuator nodes has to be programmed by the application developer. Although it has greater control over the way that programs work, it is too labor-intensive and does not promote portability.

2. Database Approach

In this model, every node is considered to be a part of a virtual database. Queries as part of an application can be issued on sensor nodes by the developer. This model does not support application logic at this level, rendering it to be of little use in IoT application development.

3. Macro Programming

In this methodology, application logic can be specified and also abstractions are provided to specify high-level communication, thereby hiding low-level details from developers, which will aid in modular and rapid development of applications.

4. Model-Driven Development

This takes note of both vertical and horizontal separation of concerns. Vertical separation increases the level of abstraction, thereby reducing applicationdevelopment complexity. Horizontal separation of concern reduces development complexity by describing the system, using different system views. Each perspective elaborates on a certain aspect of the system.

Existing IoT Frameworks

The IoT research communities from many academic and research organizations are constantly striving to simplify the efforts involved in IoT application development by developing new programming frameworks.

1. Mobile Fog

Cisco has proposed a new computing model called Fog computing [33]. Here, generic application logic is executed on resources throughout the network, including routers and dedicated computing nodes. In contrast to the pure Cloud paradigm, fog-computing resources perform low-latency processing near the edge, while latency-tolerant, large-scope aggregations are performed on powerful resources in the core of the network (Cloud).

2. ELIoT (Erlang Language for IoT)

Although the language Erlang was originally designed for embedded platforms, over a period of time it amassed a complex infrastructure, which is usually not required in devices and is a burden on resource constrained things. ELIoT [35], Erlang language for IoT, tries to address this for IoT application development.

3. Compose API

Compose API is an IoT service-provider platform through RESTful APIs, wherein things, users, and the Compose platform can interact with each other to provide

services based on IoT, called Internet of Services (IoS). Compose platform is based on Web of Things (WoT): all of the physical objects connected to the platform are web-enabled and can interact among themselves using the web protocols. Along with the APIs, the compose platform consists of GUI, semantic registry, cloud runtime, and communication libraries.

4. Distributed Dataflow Support for IoT

In this approach, existing IoT dataflow platforms such as WOTkit processor and Node-RED are extended to support distributed dataflow, which is one of the important characteristic features of IoT. Dataflow programs are generally called flows, which consist of nodes connected by “wires.” The dataflows are generated using JSON documents. During execution, nodes get instantiated in the memory, and the code is executed as and when the node receives data on the incoming “wire.”

5. PyoT

PyoT is a programming framework for WSNs, which have the capability to communicate with each other through the Internet, using 6LoWPAN and CoAP. PyoT abstracts WSNs as software objects, which can be manipulated and composed to perform complex tasks. PyoT uses CoAP’s RESTful interface to interact with nodes. Applications can consider sensing and actuating capabilities of nodes, shared with the external world through URIs. The users can discover available resources, monitor sensors and actuators, store data, define events and actions, and program to interact with resources using Python. PyoT supports “in-network processing,” in which a part of the application logic can be directly run on devices.

Module - 03

⇒ Stream processing in IoT

Introduction

The emergence of stream processing is driven by the incompetence of the traditional batch-processing paradigm, when it comes to processing fast data. Nowadays, building a modern information-technology system demands the ability of processing an unprecedented volume of data using possibly distributed resources, and exploring the concealed value of data within a tight time-constraint.

Having gained extensive attention from the research and industrial community, the batch-processing model derives a series of techniques to accomplish the first goal.

These batch-based techniques are relatively competent in terms of handling the ever-growing data volume and increasingly complex data format. However, they are all struggling to meet the second goal, in which the strict time-constraint has eliminated the luxury of storing the data somewhere before executing relevant operations against it. When it comes to the data- processing phase, these streams flow through a computation topology where continuous queries (ie, long-standing queries that usually operate over time and buffer windows) are installed to be processed in a record-by-record manner

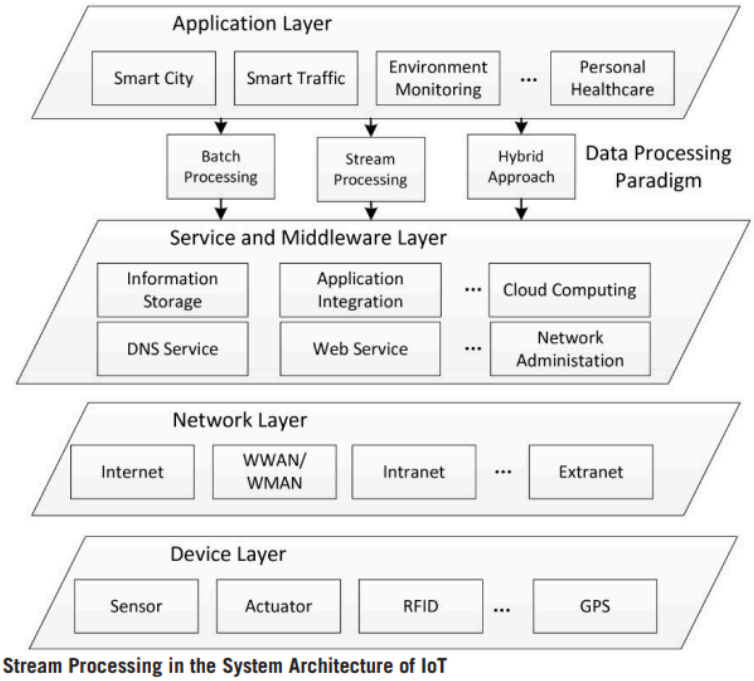

FIGURE 8.1 Stream Processing in the System Architecture of IoT

The applications from the IoT domain have always been the driving force that motivates the development and adoption of the stream-processing paradigm. The primary cause is that

the way of data generation has become increasingly active in the emerging IoT applications. Previously, data in the conventional scenarios resulted from passive reactions to real-world events or user queries, but nowadays IoT data are mostly automatically generated by large-scale sensor networks for monitoring and decision-making purposes. As a consequence, not only has the amount of data being generated soared, but also the places of data production have become much more geographically dispersed than before. In some cases, leveraging the stream-processing model to handle data in motion is the only viable option

⇒ The foundations of stream processing in IoT

There is a considerable ambiguity related to the terms “stream processing” and “stream analytics,” as they have been simultaneously used by a diverse range of research communities. For example, stream processing in the context of parallel processing refers to a computer programming paradigm that allows applications to better exploit computation parallelism using a combination of heterogeneous resources, such as CPU, GPU, or field-programmable gate arrays (FPGAs).

On the other hand, stream processing in the field of connection-oriented communications means to transmit and interpret digitally encoded coherent signals in order to convey data packets for the higher-level network abstraction.

Stream

A stream is a sequence of data elements ordered by time. The structure of a stream could consist of discrete signals, event logs, or any combination of time-series data, but the way of recovering data from one to another must be append-only, resembling a conveyor belt that continuously carries data elements through a processing pipeline. In terms of representation, a data stream has an explicit timestamp associated with each element, which serves as a measurement of data order. Based on this, we formally define the denotation of stream in the context of IoT, as a Data Element–Time pair (s, Δ), where

1. .s is a sequence of data elements that are made available to the processing system over time. A data element may consist of several attributes, but it is normally atomic, as these attributes are tightly coupled with one another for logical consistency. Typical element types include immutable data tuples of the same or similar category, as well as heterogeneous events that come from a variety of sources. Depending on the specific application scenario, data elements can be either regularly generated by sensor networks that have monitoring intervals, or randomly produced by real-world

events such as user clicks on a website, updates to a particular database table, and system logs produced by Internet services.

2. Δ is a sequence of a timestamp that denotes the sequence of data elements. Since heterogeneous elements could be aggregated into a single stream out-of-order due to the uncertainty of distributed data-collection and transmission procedures, the use of a timestamp is necessary to reconstruct the logic sequence for the following analytics. In addition, timestamps can be also used to evaluate the real-time property of a stream-processing system, by checking on whether the results have been presented on time.

Stream Processing

Stream processing is a one-pass data-processing paradigm that always keeps the data in motion to achieve low processing-latency. As a higher abstraction of messaging systems, stream processing supports not only the message aggregation and delivery, but also is capable of performing real-time asynchronous computation while passing along the information. The most important feature of the streaming paradigm is that it does not have access to all data. By contrast, it normally adopts the one-at-a-time processing model, which applies standing queries or established rules to data streams in order to get immediate results upon their arrival.

All of the computation in this paradigm is handled by the continuously dedicated logic-processing system, which is a scalable, highly available, and fault-tolerant architecture that provides systemlevel integration of a continuous data stream. As a consequence of the timeliness requirement, computations for analytics and pattern recognition should be relatively simple and generally independent, and it is common to utilize distributed-commodity machines to achieve high throughput with only sub-second latency.

However, there is another subclass of stream-processing systems that follows the microbatch model. Compared to the aforementioned one-at-a-time model, in which it is difficult to maintain the processing state and guarantee the high-level fault-tolerance efficiently, the microbatch model excels in controllability as a hybrid approach, combining a one-pass streaming pipeline with the data batches of very small size. It greatly eases the implementation of windowing and stateful computation, but at the cost of higher processing-latency. Although such a model is called microbatch, we still consider it to be a derived form of stream processing, as long as the target data remains constantly on the move while it is being processed.

Characteristics of stream data in IoT

As suggested by its name, stream data in IoT constitutes inherently dynamic, continuous, and unidirectional data flows that are normally processed in a one-pass manner. Such a dynamic paradigm has endowed it with several common properties, such as timeliness, randomness, endlessness, and volatility.

1. Timeliness and Instantaneity

   Ensuring the timeliness of processing requires the ability to collect, transfer, process, and present the stream data in real-time. As the value of data may vanish over time rather rapidly, the streaming architecture needs to perform all the calculation and communication on the fly with the data that has newly arrived.

2. Randomness and Imperfection

   Randomness and data imperfection are two direct consequences of the dynamic nature of stream data. There could be several unforeseeable factors that affect the processing chain. For example, the data generation process may induce randomness because the data sources are normally independently installed in different environments, which makes it nearly impossible to guarantee the sequence of data arrival across different streams.

3. Endlessness and Continuousness

   As long as the data sources are alive and the stream-processing system is properly functioning, newly generated data will be continuously appended to the data channel until the whole application is explicitly turned off. Therefore, processing stream data needs the support of high-level availability to avoid any possible interruption of data flow, which may lead to the accumulation of backlogs, and, finally, the breach of the real-time promise.

The general architecture of stream processing system in IoT

First of all, we argue that a stream-processing architecture should include an integral data-processing chain that covers the whole lifespan of data (from its generation up to its consumption). However, most of the previous research had used this term in a narrower sense, only referring to the organization of a logic-processing system where the relevant analytics are performed.

When it comes to building stream applications for real-world scenarios, such an assumption is poorly suited because collecting and aggregating data from geo-distributed data sources are inherently costly procedures. There are a series of development and deployment hurdles to be overcome by the use of dedicated streaming components.

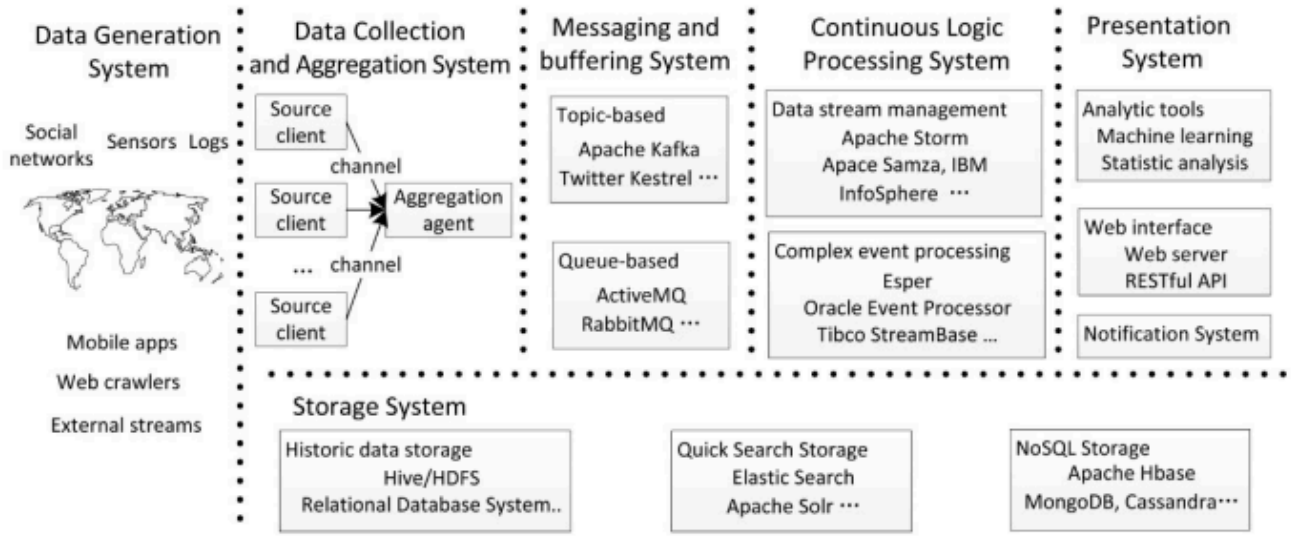

FIGURE 8.2 General Architecture of a Stream Processing System in IoT

The data-generation system denotes the spectrum of data sources that continuously produce raw information for the data-processing chain. There are a lot of entities that can fulfill this definition, which makes a full enumeration nearly impossible. However, we can still categorize the generated data into three types, in accordance with their modalities.

The first type, static data, refers to the long-term information that has already been stored in on-premise infrastructures or remote locations. As these data are mostly derived from the validated knowledge and are not frequently updated, they are usually fetched by the stream-processing system on a regular basis, serving as reference information during the analytic procedure. The second type, centralized stream data, is a special type of stream that only comes from a single centralized data source. Data of this type sometimes even demands to be processed right in the same place where it is generated, so there would be no need for aggregating data to achieve a unified data-view.

The Data Collection and Aggregation System combining with the Messaging and Buffering System plays the role of a message broker in the whole data-processing chain. To collect and aggregate different types of data, various forms of source clients are independently installed to drive the newly generated data in motion, while several aggregation channels are provided to gather these stream data into a centralized buffer, using hierarchical aggregation agents. There are two types of message buffers in terms of implementation: some are topic-based, which support a higher-level programmability, whereas the others are queue-based, and thus mainly optimized for performance concerns.

⇒ Challenges and future directions

The current stream-processing systems have been greatly improved to cater to the emerging needs of IoT applications. A state-of-the-art stream-processing system now should satisfy the following criteria: (1) horizontal scalability to accommodate different sizes

of processing needs, (2) easy to program and manage while concealing the tedious low-level implantation from its users, and (3) capable of dealing with possible hardware faults with graceful performance degradation rather than sudden termination.

Scalability

Scalability does not just refer to the ability to expand the system to catch up to the ever-increasing data streams, so that the promise of the Quality of Service (QoS) or Service Level Agreement (SLA) could be honored. Elasticity, the ability to dynamically scale to the right size on demand, is the future and advanced form of scalability. An efficient resource-allocation strategy should be adopted, by which the stream-processing system can start running with only limited resource usages, especially when the data sources are temporarily idle during the application-deployment phase. Afterward, as the workload of IoT may fluctuate and the user requirement may change over time, the system should dynamically provision new resources by taking into account the characteristics of the available hardware infrastructure, and free up some of them when they are no longer needed.

Robustness

Fault-tolerance is a commonplace topic when it comes to the design and implantation of streamprocessing systems, especially when considering that its availability is one of the most crucial prerequisites to guarantee the correctness and significance of real-time processing. The previous research and practice on fault-tolerance mostly rely on either system replication or state checkpointing, which are both not flexible enough to tailor to the robustness for operations in accordance with the trending faulttypes. Designing a hybrid and configurable fault-tolerance mechanism that is capable of recovering the system from unforeseeable failures is an open research-question left to be answered.

SLA Compliance

How to negotiate the SLA for stream-processing systems has been rarely discussed in the previous research. It also depends on which platform the system is running on, and how stakeholders are involved. But an inherent requirement is to achieve cost-efficiency, which translates to minimizing the monetary cost for the users, as well as reducing the operational cost for the provider (possibly data centers). Achieving SLA-compliance requires the stream-processing system to be equipped with the ability to trade-off between the justifiable

metrics, such as performance and robustness, with the running cost, the balance of which should be left for the user to decide when signing up for SLA.

Load Balancing

Currently, the applied load-balance schemes are very simplistic, the major target of which is to normally improve the performance of the system, especially by maximizing the throughput. However, the importance of load balance goes far beyond performance optimization. A wrong balancing decision may lead to unnecessary load-shedding, dropping arrived messages when the system is deemed to be overloaded, which ultimately impairs the veracity of the processing result. It is challenging to take the low-level metrics such as task capacity or lengths of thread-message-queues into consideration during the load-balancing process, but the perspective is very promising, as currently the system utilization rate is still moderate; even the stream-processing system is already saturated, where the inefficient load-balance mechanism is the culprit to blame.

⇒ A framework for distributed data analysis in IoT

Introduction

The main idea of this framework is to support efficient local processing and summarization of the data at the nodes, followed by global processing of the local summaries. This framework follows a similar idea to Fog computing [1] in the sense that it reduces the amount of data routed over the network backbone using the computational capabilities of the devices collecting the data.

The proposed framework covers three aspects: local data modeling, communication, and global data modeling. The local data modeling requires an efficient algorithm with low computational cost. The algorithm calculates local summaries of the data and identifies anomalies in the local data. Only the summaries of the data are communicated to a central location which we refer to as sink. Limiting the communication to only data summaries alleviates the overhead of communicating all the data over the network. The global data modeling component, which is located in a central location, is responsible for finding global summaries and anomalies.

Recent advancements in sensing technologies provide a cost-effective platform for monitoring applications to gather detailed observations from different environments

including factories, mines, agriculture, and urban areas. An important aspect of a monitoring system is the ability to detect significant events or unusual behavior in the environment. These unusual patterns are also called outliers, surprises, novelties, or events in different applications. Therefore, the challenge is to build a model of the multidimensional data distributed in the network in a robust and efficient manner—robust in the sense that the model accurately captures the characteristics of the data, and efficient in the sense that the model satisfies the resource constraints of the network. There is a wide range of anomaly detection techniques applied to monitoring applications. However, only few techniques consider both communication and computational constraints of the nodes in such network.

Anomaly Detection

Anomaly detection is an important unsupervised data processing task which enables us to detect abnormal behavior without having a priori knowledge of possible abnormalities. An anomaly can be defined as a pattern in the data that does not conform to a well-defined notion of normal behavior.

This definition is very general and is based on how patterns deviate from normal behavior. On this basis we can categorize anomalies in the data into three categories:

- Outliers—Short anomalous patterns that appear in a nonsystematic way in the collected data, usually arising due to noise or faults, for example, due to communication errors.
- Events/Change—These patterns appear with a systematic and sudden change from previously known normal behavior. The duration of these patterns is usually longer than outliers. Ine nvironmental monitoring, extreme weather conditions are examples of events. The start of an event is usually called a change point.
- Drifts—Slow, unidirectional, long-term changes in data [15]. This usually happens due to the onset of a fault in a sensor

Problem statement and definitions

Our aim is to apply an anomaly detection algorithm to the data distributed in a network. We consider a sensor network topology in which a set of sensors N = {Nj :J = 1,…,l} are connected in a hierarchical tree topology to a sink B. Over a fixed monitoring period, each node Nj observes a set of nj measurements X x = = { : k n 1,, } j k j j where each measurement xk ∈ℜ j d is a vector of values observed at the node, for example, in a monitoring application these values can be temperature, humidity, and light intensity.

Distributed Anomaly Detection

We first model the normal data of each node using the ellipsoidal boundary. We then communicate the parameters of the hyperellipsoid from each node to the sink, where we cluster these l hyperellipsoids to c clusters that reflect the global distribution of measurements in the network. The c merged hyperellipsoids corresponding to these clusters are then reported back to the nodes where anomaly detection can be performed. The final step is based on the idea that at some point in time all the nodes will observe all the modes in the environment. In some applications, the network may have multiple subsections which are expected to have different characteristics. In these cases, this algorithm should be run independently within each subsection.

Clustering Ellipsoids: we introduce a clustering approach to group similar ellipsoids together instead of simply merging all the ellipsoids into one ellipsoid. The main purpose of clustering ellipsoids is to remove redundancy between the ellipsoids reported by the nodes that have the same underlying distributions.

Experimental Results: We now provide three examples (one real-life data set and two synthetic datasets where the modes or partitions in the data can be controlled) to illustrate how the distributed anomaly detection approach described earlier works. We compare the single global ellipsoid approach in Ref. [8] and the ellipsoidal clustering approach discussed here. We also use a centralized approach where all the data are transferred to the sink as a baseline. In the baseline approach, we first cluster all the data at the sink using the k-means clustering algorithm. Then, ellipsoidal decision boundaries are calculated using the data in each cluster and anomalies are flagged using all decision boundaries. Note that in the baseline, the cost of transferring all the data to the sink can be large compared to the first two methods

Efficient Incremental local modeling

One of the drawbacks of the proposed approach is that its local modeling is performed in batch mode. Each node has to buffer a window of measurements, then calculates the local ellipsoidal boundary, and sends it to the sink. The anomaly detection can be done according to both local and global ellipsoids at the node. However, selecting an appropriate window is known to be a difficult task, as a small window may result in an inaccurate

estimate of the local ellipsoid, and the data in a larger window size might not come from a unimodal distribution, which contradicts the assumption of the model.

Module - 04

⇒ Security and privacy in the Internet of Things

Introduction

The Internet of Things leads to a new computing paradigm. It is the result of shifting computing to our real-time environment. The IoT devices, besides connecting to the Internet, also need to talk to each other based on the deployment context. More precisely, IoT is not only about bringing smart objects to the Internet, but also enabling them to talk to each other. This will have direct implications to our life, and change the way we live, learn, and work.

Today, we have reasonably secure and safe online financial transactions, e-commerce, and other services over the Internet. Core to these systems is the use of advanced cryptographic algorithms that require substantial computing power. Smart objects have limited capabilities in terms of computational power and memory, and might be battery-powered devices, thus raising the need to adopt energyefficient technologies. Among the notable challenges that building interconnected smart objects introduces are security, privacy, and trust. The use of Internet Protocol (IP) has been foreseen as the standard for interoperability for smart objects. As billions of smart objects are expected to come to life and IPv4 addresses have eventually reached depletion, the IPv6 protocol has been identified as a candidate for smart-object communication. The challenges that must be overcome to resolve IoT security and privacy issues are immense.

IoT Reference Model

Today, there is no standardized conceptual model that characterizes and standardizes the various functions of an IoT system. Cisco Systems Inc. has proposed an IoT reference model that comprises seven levels. The IoT reference model allows the processing occurring at each level to range from trivial to complex, depending on the situation. The model also describes how tasks at each level should be handled to maintain simplicity, allow high scalability, and ensure supportability. Finally, the model defines the functions required for an IoT system to be complete

Table 10.1 IoT World Forum Reference Model	
IoT Reference Model	
Levels	**Characteristics**
Physical devices and controllers	End point devices, exponential growth, diverse
Connectivity	Reliable, timely transmission, switching, and routing
Edge computing	Transform data into information, actionable data
Data accumulation	Data storage, persistent and transient data
Data abstraction	Semantics of data, data integrity to application, data standardization
Application	Meaningful interpretations and actions of data
Collaboration and processes	People, process, empowerment, and collaboration

The important design factor is that IoT should leverage existing Internet communication infrastructure and protocols. Level 3 is famously referred to as Edge Computing or Fog Computing. The primary function is to transform data into information, and perform limited data-level analytics. Contextspecific information processing is done at this level so that we obtain actionable data. An important feature of fog computing is its capability of real time processing and computing. More precisely, levels 1, 2, and 3 are concerned with data in motion, and the higher levels are concerned with information derived from the data items. It leads to an unprecedented value zone wherein people and the processes are empowered to take meaningful action from the underneath world of IoT. The core objective is to automate most of the manual processes, and empower people to do their work better and smarter. At each level of the reference model, the increasing number of entities, heterogeneity, interoperability, complexity, mobility, and distribution of entities represent an expanding attack surface, measurable by additional channels, methods, actors, and data items. Further, this expansion will necessarily increase the field of security stakeholders and introduce new manageability challenges that are unique to the IoT.

IoT Security Threats

There are three broad categories of threats: Capture, Disrupt, and Manipulate. Capture threats are related to capturing the system or information. Disrupt threats are related to denying, destroying, and disrupting the system. Manipulate threats are related to manipulating the data, identity, time-series data, etc. The simplest type of passive threats in the IoT is that of eavesdropping or monitoring of transmissions with a goal to obtain information that is being transmitted. It is also referred to as capture attacks. Capture attacks are designed to gain control of physical or logical systems or to gain access to information or data items from these systems. The ubiquity and physical distribution of the IoT objects and systems provide attackers with great opportunity to gain control of these

systems. The distribution of smart objects, sensors, and systems results in self-advertisements, beacons, and mesh communications, providing attackers greater opportunity to intercept or intercede in information transmission within the environment.

Some of the well-known active threats are as follows: Masquerading: an entity pretends to be a different entity. This includes masquerading other objects, sensors, and users. Man-in-the-middle: when the attacker secretly relays and possibly alters the communication between two entities that believe that they are directly communicating with each other. Replay attacks: when an intruder sends some old (authentic) messages to the receiver. In the case of a broadcast link or beacon, access to previous transmitted data is easy. Denial-of-Service (DoS) attacks: when an entity fails to perform its proper function or acts in a way that prevents other entities from performing their proper functions.

IoT Security Requirements

The basic security properties that need to be implemented in IoT are listed next. Confidentiality: transmitted data can be read only by the communication endpoints; availability: the communication endpoints can always be reached and cannot be made inaccessible; integrity: received data are not tampered with during transmission, and assured of the accuracy and completeness over its entire lifecycle; authenticity: data sender can always be verified and data receivers cannot be spoofed and authorization: data can be accessed only by those allowed to do so and should be made unavailable to others. The requirements for securing the IoT are complex, involving a blend of approaches from mobile and cloud architectures, combined with industrial control, automation, and physical security

1. Lightweight Security

   The unprecedented value of IoT is realized only when smart objects of different characteristics interact with each other and also with back-end or cloud services. IPv6 and web services become the fundamental building blocks of IoT systems and applications. In constrained networked scenarios, smart objects may require additional protocols and some protocol adaptations in order to optimize Internet communications and lower memory, computational, and power requirements. The use of IP technologies in IoT brings a number of basic advantages such as a seamless and homogeneous protocol suite, and proven security architecture.

2. Scale

   The important requirement is the scale in which an IoT environment is expected to grow. The population of entities is expected to grow exponentially as users embrace more smart and connected objects and devices, more sensors are deployed, and

more objects are embedded with intelligence and information. Each entity, depending on its nature, characteristics, carries with it an associated set of protocols, channels, methods, data models, and data items, each of which is subject to potential threat.

3. IP Protocol Based IoT

   The use of IP technologies in IoT brings a number of basic advantages such as a seamless and homogeneous protocol suite, and proven security architecture. It also simplifies the mechanisms to develop and deploy innovative services by extending the tested IP-based frameworks. It leads to a phenomenon called “expansion of attack surface.”

4. Heterogeneous IoT

   Another important design consideration in the IoT is how the connected things can work together to create value and deliver innovative solutions and services. IoT can be a double-edged sword. Although it provides a potential solution to the innovation imperative, it can also significantly boost operational complexity if not properly integrated with key organizational processes. Security processes should also be properly designed to align with the organization processes. The complex operational technologies make it difficult for designing a robust security architecture in IoT. It is a common opinion that in the near future IP will be the base common network protocol for IoT.

⇒ IoT Security Overview

IoT Protocols

It starts with the description of general security architecture along with its basic procedures, and then discusses how its elements interact with the constrained communication stack and explores pros and cons of popular security approaches at various layers of the ISO/OSI model.

It gives an overview of the deployment model and general security needs. It presents the challenges and requirements for IP-based security solutions and highlights specific technical limitations of standard IP security protocols (IPSec). There are currently IETF working groups focusing on extending existing protocols for resource-constrained networked environments.

Network and transport layer challenges

The IPSec uses the concept of a Security Association (SA), defined as the set of algorithms and parameters (such as keys) used to encrypt and authenticate a particular flow in one direction. To establish a SA, IPSec can be preconfigured (specifying a preshared key, hash function, and encryption algorithm) or can be dynamically negotiated by the IPSec Internet Key Exchange (IKE) protocol. The IKE protocol uses asymmetric cryptography, which is computationally heavy for resource-constrained devices. To address this issue, IKE extensions using lighter algorithms should be used. Data overhead is another problem for IPSec implementations in IoT environments. This is introduced by the extra header encapsulation of IPSec AH and/or Encapsulating Security Payload (ESP), and can be mitigated by using header compression

IoT Gateways and security

Connectivity is one of the important challenges in designing the IoT network. The diversity of end points makes it very difficult to provide IP connectivity. It is important that non-IP devices too have a mechanism to connect with IoT. The IoT gateways can simplify IoT device design by supporting the different ways nodes natively connect, whether this is a varying voltage from a raw sensor, a stream of data over an inner integrated circuit (I2C) from an encoder, or periodic updates from an appliance via Bluetooth. Gateways effectively mitigate the great variety and diversity of devices by consolidating data from disparate sources and interfaces and bridging them to the Internet. The result is that individual nodes do not need to bear the complexity or cost of a high-speed Internet interface in order to be connected.

IoT Routing Attacks

Threats arising due to the physical nature of IoT devices can be mitigated by appropriate physical security safeguards, whereas secure communication protocols and cryptographic algorithms are the only way of coping with the fact that they arise due to IoT devices communicating with each other and the external world. For the later, IoT devices can either run the standard TCP/IP protocol stack, if their computational and power resources allow, or can run adaptions which are optimized for lower computational and power consumption. There are some well-known routing attacks that can be exploited by attackers. The 6LoWPAN networks or an IP-connected sensor networks are connected to the conventional Internet using 6LoWPAN Border Routers (6LBR).

Bootstrapping and Authentication

Bootstrapping and authentication controls the network entry of nodes. Authentication is highly relevant to IoT and is likely to be the first operation carried out by a node when it joins a new network, for instance, after mobility. It is performed with a (generally remote) authentication server using a network access protocol such as the PANA.For greater interoperability, the use of the EAP is envisioned. Upon successful authentication, higher layer security associations could also be established (such as IKE followed by IPSec) and launched between the newly authenticated endpoint and the access control agent in the associated network.

Authorization Mechanisms

The present day services that run over the Internet, such as popular social media applications, have faced and handled privacy-related problems when dealing with personal and protected data that might be made accessible to third parties. In the future, the IoT applications will face similar issues, and others that may be unique to the domain. The OAuth (Open Authorization) protocol has been defined to solve the problem of allowing authorized third parties to access personal user data.

⇒ Robustness and Reliability

Introduction

Building a reliable computing system has always been an important requirement for the business and the scientific community. By the term reliability, we mean how long a system can operate without any failure. Along with reliability, there is another closely related quality attribute, called availability. Informally, availability is the percentage of time that a system is operational to the user. An internet of things (IoT) system deploys a massive number of network aware devices in a dynamic, error-prone, and unpredictable environment, and is expected to run for a long time without failure.

Failure Scenarios

Like any mission critical system, we say that an IoT-based system becomes unreliable or unavailable when the system either fails to respond to a request or provides an unexpected, incorrect service [28]. A service failure happens when faults are not handled properly.

Researchers and practitioners have extensively studied various faults and remedial actions to keep software operational in a business critical scenario.

1. Infrastructure fault

   The cluster of network-enabled devices in an IoT-based system are expected to operate in unanticipated scenarios. Some of these scenarios can lead to infrastructure failures. For instance

   - In a given IoT application scenario, the network-enabled devices ought to be embedded in a specific environment to gather and process data stream. The devices can fail due to the physical condition and interference with the environment in which they operate. Such an operating condition can reduce the life of such devices drastically due to the physical deterioration.
   - The external environment may provide unexpected inputs to IoT entities resulting in a computation failure in the device.

2. Interaction Fault

   Network-enabled devices and appliances have widely varying compute capabilities. When these devices are made to communicate with each other and share data, there can be operational failures due to several reasons:

   - The entire network or the communication components can fail.
   - Due to the heterogeneity of the devices, there could be an “impedance mismatch” of the data being exchanged. Such a scenario can occur when the IoT system allows a device to join the network in real time.
   - Interaction faults are also caused by unexpected workload coming from various IoT components.

3. Fault In Service Platforms

   It is unlikely that the platform will be built from the scratch. It will integrate many third party products and will be integrated with external partner systems. Even if we assume that this middleware has been thoroughly tested for its own functionality, many transient faults can be due to off-the-shelf components. The reliability of these third party components may be questionable and can often be a cause of failure of the main system. Additionally, the external partner systems of the IoT application middleware can fail or provide incorrect data, which can result in a failure of the middleware.

Reliability Challenges

1. Making Service available to user

The aim of an IoT application is to provide an immersive service experience through a tightly coupled human–device interaction in real time. Therefore, it is highly important that the availability of the system be judged from the user perspective. This is known as “user perceived availability” [21] where the perceived availability is about delivering the service to the user, not just surviving through a failure. A relatively old study on Windows server [22] shows that though the server was available for 99% of the time (obtained from the server log), but the user perceived it to be just 92%.

2. Serviceability of IoT systems

We discussed earlier that it is quite likely for an IoT system to have a set of devices that can dynamically come and join the system. In such a case, ensuring serviceability without disrupting the ongoing activities is more difficult in the case of IoT devices. In a traditional high-available system, nodes as well as the software running on them go through a scheduled maintenance when the software and patches are upgraded to prevent any upcoming failure. Such a traditional maintenance may not be feasible for an IoT system.

3. Reliability at network level

Most Internet of Things applications for buildings, factories, hospitals, or the power grid are long-term investments that must also be operable for a long time. The networks can also be unmanaged (eg, home automation, transport applications). This implies that the network must be able to configure itself as environmental conditions or components in the network itself change so that the information can always be transmitted from one application to the other reliably. There can be further complexities in using sensor devices.

4. Device level reliability

From the network level, let us now focus on the embedded devices that are connected via network. Even when the network is reliable, there are scenarios when the applications running on these devices may generate poor quality data, which makes the entire computation unreliable. Consider a scenario when the device needs to gather image from the physical world where it is embedded. Due to the environmental condition, the quality of the images captured can be below the acceptable level; as a result the associated inferences drawn from the captured images become unreliable. For sensor devices, the environmental condition can result in a bit error

Privacy And Reliability

Data privacy is an important part of IoT, specifically when an IoT system allows a machine-to-machine interaction, where machines can join the network dynamically. In this context, identity management, and proving identity on-demand has been considered an important mechanism to ensure the authenticity of the communicating parties. For instance consider an IoT-based vehicle management system, which expects vehicles to reveal their identity in a vehicular network. The system can create an alarm and can trigger actions if the deployed sensors on a street sense that a car has not revealed the identity. However, this alarm can be a false one if the car is a police car, which can reveal its identity to another police car and to the designated staff at the police station, but keep its identity hidden during undercover work otherwise. Under such a scenario, the real-time surveillance system's reliability of detecting intrusion becomes questionable due to the inaccuracy. In this chapter we refrain from discussing detailed issues related to identity preservation, anonymization, and use of pseudonyms (using alternate identity) since they strictly belong to the security.

Interoperability of Devices

Since IoT allows heterogeneity of devices interacting with each other, there is always a possibility that the participating devices cannot exchange information due to the lack of standardization. As of today, the standardization of the communication among the devices has not been enforced in IoT. In the article from Telenor group the interoperability issues of communicating devices in the context of IoT have been discussed in detail. Traditionally, system reliability is often associated with various other quality attributes like performance, availability, and security. Interoperability, an important quality attribute by itself, has not traditionally been considered in conjunction with reliability.

⇒ TinyTO: Two-Way Authentication for constrained Devices In The Internet of Things

Introduction

Throughout the years, especially WSN consisting of constrained devices with limited resources in memory, energy, and computational capacity, rapidly gained popularity. Thus, the questions raised of how to integrate them into the IoT and what challenges occur when looking at their constrained resources. The number of possible deployments of such

networks rises, and more applications have a need for confidential and authenticated communication within the network. This security issue must be addressed, due to the fact that sensitive information (e.g., identity (ID), names, or Global Positioning System (GPS) information) is linked almost everywhere to all kinds of collected data, such as temperature, sound, and brightness. Hence, collected data is no longer anonymous and is often desired to be kept confidential. If room information can be retrieved by eavesdropping due to missing security in the communication, then an attacker would be aware of sensitive information and could plan, for example, a burglary. Therefore, collected data must be transmitted in a secure manner and/or over a secure channel providing end-to-end security, giving only authorized entities (e.g., gateway, security system, or company members) access to this confidential information.

Security Aspects and Solutions

The necessity of providing an end-to-end security solution in WSNs is not entirely new. Over the years, different approaches have emerged that address various security issues. Thus, an often-quoted solution is to predistribute symmetric keys. However, flexibility of the deployment, connectivity between nodes, and resilience against attackers is limited significantly.

The basic idea is to allow for individual nodes to verify that a transmitted public key matches the claimed identity, without relying on a trusted third party (e.g., Certificate Authority (CA)). For an exhausting mapping among all keys and identities, a large number of keys and certificates must be stored on every node, which is not feasible. Hence, a hash function, mapping from identity to the hash value of the corresponding public key, is preshared. Thus, only hash values and identities must be compared, which requires only a fraction of the memory and computational power. This can be optimized further by using Merkle Trees, in which nonleaf nodes are labeled with the hash of the labels of its children.

Design Decision

An ideal solution for the two-way authentication should work generically on WSN nodes of all classes, especially because the trend goes toward heterogeneous WSNs. However, because WSN nodes are primarily designed to collect data, they prioritize frugality and longevity over processing-power and memory size.

One approach to adapt the traditional Public-Key Cryptography (PKC) to WSNs is the integration of extra hardware into nodes, for performing security operations and operations

that are separate from the main application and the node processor. At first glance, Class 2 devices have more resources and can be used for this purpose.

TinyTO Protocol

Due to TinyTO's main goal of supporting an end-to-end security with two-way authentication on Class 1 devices, the authentication protocol has to always include a key exchange, such that several possible handshake candidates can be considered in practice, leading to the final design and implementation of TinyTO. First, handshake candidates for TinyTO and their drawbacks are introduced. Second, the resulting TinyTO handshake, including two-way authentication purposes and aggregation support, are described. Finally, key information on the respective implementation is presented.

***Possible Handshake protocol candidates*:** Handshake protocol candidates considered in this section support a two-way authentication of two independent entities without prior information exchange, which make them highly appropriate for TinyTO. From this stage on, the traditional naming pattern of cryptography is applied to protocol descriptions, assuming two communication parties—Alice and Bob—which are instantiated as sensor nodes.

At first glance the Station-to-Station protocol (STS) seems to be an ideal candidate for TinyTO because STS is based on a Diffie–Hellman's key exchange, followed by an exchange of authentication signatures [41]. Both parties, Alice (A) and Bob (B), compute their private key x and a public key X in the beginning. Next, Alice sends her public key XA to Bob. Once Bob receives XA, he can compute a shared secret KAB with XA and xB, according to the Diffie–Hellman's key-exchange algorithm [38]. Bob can now encrypt any message to Alice using KAB. For decryption purposes Bob sends XB back to Alice, so that she can compute the same shared secret KAB. Additionally, Bob sends a token consisting of both public keys, signed with his own private key to authenticate himself. Alice can use XB to verify that Bob was indeed the same person who had signed the message and computed the shared secret. Bob is now authenticated to Alice. As the last step of the two-way authentication, Alice constructs an authentication message and sends it to Bob to authenticate herself to Bob. To avoid unnecessary communication overhead, the second key-exchange message is combined with the first authentication message.

***BCK With Preshared keys for TinyTO*:** PSK is selected for TinyTO to verify the identity of an entity. The distribution of PSKs is simple in the context of WSN devices: Adding a unique PSK to the programming procedure introduces practically no overhead because nodes

need to be programmed before deployment in any case, and the key generation and management can be moved to the software programming the nodes. Compared to approaches where every node is equipped with a set of keys for encryption between peers before deployment, TinyTO assumes that every node has only one PSK, solely for authentication toward the gateway. The developed handshake for TinyTO compares to BCK,with preshared keys that form master keys for an initial authentication between the node and the gateway.

Handshake Implementation: From now on, it is assumed that TinyTO is implemented in TinyOS, where different components are “wired” to each other and use the offered set of functionality. Thus, the TinyTO handshake is implemented in the component HandshakeHandler, which is exclusively responsible for handshake-message handling, including message composition and reply handling. HandshakeHandler is wired to components called in for cryptographic functions. The three TinyTO handshake messages HS1 to HS3 are implemented in a similar manner. The msgType field is used to distinguish between handshake messages and other types of control messages that are sent via the same port. hsType identifies different handshake messages HS1, HS2, and HS3. Furthermore, public ECC keys are broken down into x- and y-coordinates for easy handling on the node’s side.

⇒ Obfuscation and diversification for securing the Internet of Things

Introduction

Information sharing in Internet of Things (IoT) is the element that makes the cooperation of the devices feasible, but on the other hand, it raises concerns about the security of the collected data and the privacy of the users. Some of the collected data contain (sensitive) personal and business information about the users. Therefore, it is highly significant to appropriately protect this data while it is stored and transmitted.

Nonetheless, the IoT service providers principally are tackling the availability and interoperability of the IoT services, so the security of the devices and services has never been the main focus. A report by claims that nearly 70% of the devices participating in IoT are vulnerable to security exploits, which make the network prone to security attacks. Therefore, there should be effective techniques to protect these devices and the shared

information over the network, in order to ensure the liability of the system in addition to the availability of it.

Distinguishing characteristics of IoT

1. Operating Systems and software in IoT

   IoT is made up of a wide variety of heterogeneous components, including sensors, devices, and actuators. Some of these components are supplied by more powerful 32-bit processors (eg, PCs and smartphones), whereas some others are equipped with only lightweight 8-bit micro-controllers. On that account, the chosen software should be compatible with all ranges of devices, including the low-powered ones. Moreover, the software should not only be able to support the functionality of the devices, but also should be compatible with the limitation of the participating nodes of this network, in terms of computational power, memory, and energy capacity.

   The following items are the generic prerequisites of software running on IoT devices:

   - Heterogeneous hardware constraints: The chosen software for IoT should require a fairly low amount of memory, and operate with low complexity, so that the IoT devices with limited memory and computational power would be able to support the operations.
   - Programmability: From the development point of view, the chosen software should provide a standard application-program interface (API), and should also support the standard programming languages.
   - Autonomy: For energy efficiency means, (1) the software should allow sleep cycles for saving energy when the hardware is in the idle mode; (2) the network stack should be adaptive to the constrained characteristic of the devices in IoT, and also should allow the protocols to be replaced at each layer; and (3) the chosen software should be sufficiently robust and reliable.

2. IoT Network stack and access protocols

   IoT is built up of a large number of objects, including sensors, devices, and applications, connected to one another. There are three different types of links for connecting these objects to each other. (1) device-to-device (D2D), which is the connection between the devices; (2) device-to-server (D2S), which is the connection between the devices and the servers, and (3) server-to-server (S2S), which is the connection between different servers to share collected data. Considering the TCP/IP as the de facto standard for the communication networks, some are in the

opinion that it could be used also for IoT in the future, to provide flexible IP based architecture. Currently, the low capacity of the resource-constrained devices makes it challenging to deploy IPv6 in IoT and in Low power and Lossy Networks (LLNs). LLNs are types of networks in which both routers and nodes are constrained in terms of memory, energy, and processing power.

3. Security and privacy in IoT

   Considering the fact that IoT is founded on the Internet, it is susceptible to the traditional security attacks threatening the Internet. Furthermore, the key characteristics of IoT not only make the traditional security challenges more severe, but they also introduce new challenges.

   - Heterogeneity: IoT embraces a diverse set of devices with different capabilities that communicate with each other. An extremely constrained device should open up a secure communication channel with a more powerful device, for example, a smartphone.
   - Low capacity: The devices in IoT are fairly limited in terms of computational power, memory, and battery capacity, which make them unable to handle complex security schemes.
   - Wireless connection: Devices connect to the Internet through various wireless links (eg, ZigBee, Bluetooth). The wireless connection increases the chance of eavesdropping.
   - Embedded use: most of the IoT devices are designed for a single purpose with different communication patterns.
   - Mobility: The devices in IoT are mobile and are connected to the Internet through various providers.

Obfuscation and diversification techniques

Code obfuscation is to transform the program's code into another version, which is syntactically different but semantically the same. That is, the program still produces the same output although its implementation differs from the original one. The purpose of this transformation is to make the code more complex and difficult to understand, in order to make the malicious reverse-engineering of the code harder and costlier for the attacker. Certainly, with given time and resources, the attacker may succeed in comprehending and breaking the obfuscated code; however, it requires more time and energy compared to

breaking the original version. There have been several different obfuscation mechanisms proposed. Each of these mechanisms applies the obfuscation transformations at various parts of the code, and also at various phases of the program development. For instance, obfuscation through opaque predicates is a technique used to alter the control flow of the program.

Opaque predicates are Boolean expressions that are always executed in the same way, and the outcome is always known for the obfuscator, but not for the attacker in priori.

Diversification aims to make diverse unique instances of a program that are syntactically different but functionally equivalent. This idea was pioneered in 1993, to protect the operating systems through generating diversified versions of them.

Currently, the software and operating systems are developed and distributed in a monocultural manner. Their identical design makes them suffer from the same types of vulnerabilities, and they are prone to the same security attacks. An intruder, by exploiting these vulnerabilities, can simply undertake a vast number of systems. Diversification, by introducing multiculturalism to the software design, aims at impeding the risk of massive-scale attacks. The way that a program is diversified is unique, and it is kept secret. Assuming that the attacker discovers the diversification secret of one instance of the program, it can possibly attack against that specific version, whereas the others would still be safe. That is to say, a single-attack model does not work on several systems, and the attacker needs to design system-specific attack models. For this reason, diversification is considered to be a promising technique to defend against massive-scale attacks and protect the largely distributed systems.

Enhancing the security in IoT using obfuscation and diversification techniques

The majority of the security threats in IoT base their exploits on the vulnerabilities that exist at the application layer and the network layer. As we discussed earlier, the vulnerabilities caused at the development phase of the applications and software are unavoidable, and some of them remain unknown until an attack occurs (ie, zero-day attacks). Therefore, there should be security measures considered to prevent intruders from taking advantage of these vulnerabilities. To this end, we propose two novel techniques that make it difficult for an attacker to exploit the system’s vulnerabilities to conduct a successful attack.

We propose (1) obfuscating/diversifying the operating systems and APIs used in the IoT devices, and (2) obfuscating/diversifying some of the access protocols among nodes of this network. The earlier approach secures the IoT at the application layer, whereas the later introduces security at the network layer.

The sensors and devices participating in IoT function with the help of the operating systems and software on them, and, like any other software system, they have vulnerabilities that make them prone to security attacks. An intruder seeks to exploit existing vulnerabilities on the system by finding his or her entry to the system. For instance, a piece of malware can capitalize on these vulnerabilities to inject its malicious code to spy on or manipulate the targeted system. In our proposed approach, we do not aim at removing these vulnerabilities, but we aim at preventing or making it difficult for the attacker to learn about these vulnerabilities and to utilize them. We achieve this goal by applying obfuscation and diversification techniques on the operating systems and APIs of the devices in IoT. Obfuscation of the operating systems and APIs make them complicated to comprehend, thus an attacker needs to spend more time and effort to understand the program in order to design an attack model.

The second part of our idea is to apply obfuscation and diversification on the access protocol of the communication links among the nodes of the network. The application level protocol of a network identifies the interfaces and the shared protocols that the communicating nodes use in the network. Protocol identification refers to identifying the protocol used in a communication session by the communicating parties. Static analysis could be used to capture the protocol used in the communication and compare it to the common existing protocols. The knowledge gained about the protocol used can be misused by an intruder to break into the communication, to eavesdrop, or to manipulate the data being sent over the network.

Our idea is to make it hard for an attacker to gain this knowledge and identify the protocol used. We propose to obfuscate the protocols, in order to make the protocol unintelligible and difficult to identify. Protocol obfuscation removes/scrambles the characteristics that make the protocol identifiable, such as byte sequence and packet size (eg, by making them look random). Cryptography is a common way to obfuscate the protocol. Upon the security need and the capacity of the network, different levels of encryption could be employed.

Module - 05

(The content of this module differs from what is in the syllabus; the headings are referred from the notes uploaded on AES)

Scenario

Imagine you would like to know the weather outside in your garden. In an ideal IoT world, you would have a range of sensors, for example, a temperature sensor that can talk directly to the Internet. This would be achieved by sending the temperature data from the sensor directly to a backend service via the Internet. The backend service would probably display the data on a web page and would ideally offer the ability to display and analyze the data. There are a few complexities associated with such a simple example. The most obvious is that most sensors cannot communicate via the Internet to backend services. In part this is due to the cost of having Wi-Fi or wired Ethernet connections to every sensor, and in part also due to the electrical power required. Supporting a full TCP/IP networking stack on every sensor is not required if a low-power wireless network is available. It is for these reasons that local sensors often communicate via a gateway, which can broker or relay messages across the Internet. Adding a single Wi-Fi or wired Ethernet Internet connection to a gateway which communicates to a group of sensors or actuators is more cost-effective. The Internet connection can then be shared by using either local wired or wireless connections between the gateway and local devices. The aim of this chapter is to design, build, and test an environmental-sensing IoT architecture for weather monitoring. The architecture must use commodity hardware and software to produce a system that is secure, reliable, and low cost.

It must be generic and applicable to alternative application areas by swapping the sensors/actuators. The weather station will reside outdoors in a suitably exposed location and will need to record values from various sensors at regular time-intervals, mainly temperature, humidity, wind speed and direction. This data needs to be reliably stored and transmitted for use in analysis, graphs, and for consumption by external services.

Architecture Overview

There are three key components to an IoT architecture: the sensors and actuators, the gateway, and the backend services. These three components communicate via a wide variety of interfaces and protocols, and, as a whole, enable the functionality of an IoT device, as shown in Fig. 15.1.

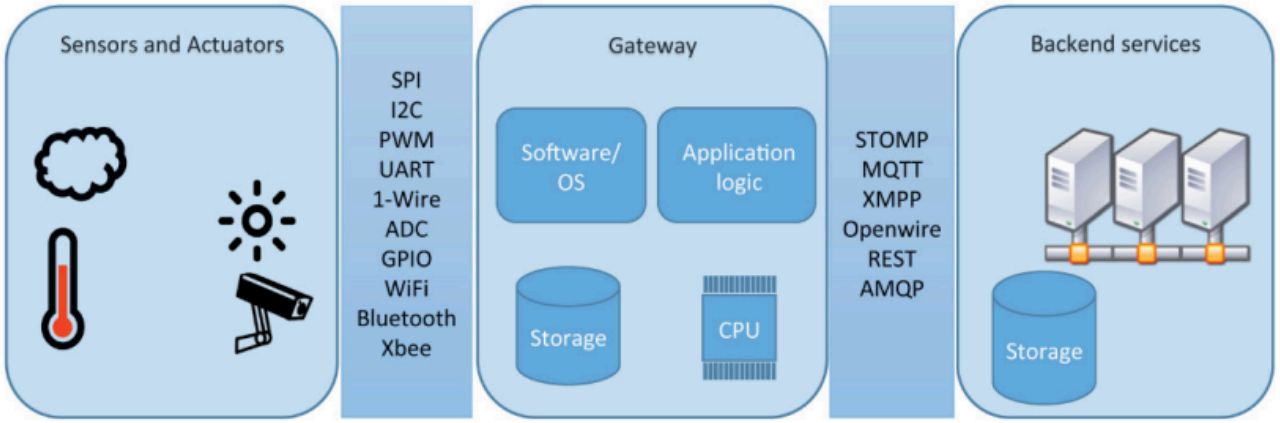

FIGURE 15.1 IoT Architecture Key Components and Example Protocols

The sensors and actuators are application-specific and will require some thought to ensure that the data collected meets the accuracy and sampling frequency required for analysis.

The gateway is responsible for communicating with the sensors and actuators as well as the backend services; it is a translator between localized interfaces, such as hard-wired sensors and remote backend systems. It basically adds TCP/IP capability to sensors, and, although it could be part of a sensor, we separate the gateway to allow a many-to-one mapping with sensors, as well as additional functionality, such as data persistence for unreliable Internet connectivity.

The backend services are predominantly used to store the IoT device data but can include additional functionality such as individual device configuration and analysis algorithms. It is beneficial to also consider data analysis as part of the backend, since the data is only useful if it can be analyzed.

Sensor to Gateway Communication

The sensors themselves usually have low-level electronic interfaces for communication. For example, I 2 C and SPI are common serial-communication buses capable of linking multiple electronic components. Connecting one or more sensors to the serial bus of a gateway device is a simple way of creating an IoT device. The transmission distance of many hard-wired interfaces can be an issue. For example, a few meters for I2 C and up to 1 km for RS-485. In its simplest form, the gateway is a converter from these lower-level electronic interfaces to Internet-compatible interfaces. Many sensors come with multiple electronic

interfaces or are available in different flavors, each with their own advantages and disadvantages. There are two key categories of interfacing, those that are physically close to the gateway, perhaps on the same Printed Circuit Board (PCB), and those that are slightly farther from the gateway, for example, on a length of cable. The chosen interface dictates the physical characteristics of the device and is therefore application-specific.

Sensors

We will now discuss the sensors required to build the environmental-sensing IoT gateway device for weather monitoring, as introduced in Section 15.2. The IoT application area will drive the sensor selection and specification. In the current section, we select a range of different sensors with a variety of capabilities as examples. The sensors have a reasonable accuracy for an external weather station, and will all be connected simultaneously, although many IoT requirements will have fewer sensors. Table 15.1 provides a summary of the sensors, their electronic interface, and hardware specifics. The ultraviolet (UV) intensity sensor uses an ML8511 [5] detector, which is most effective between 280 and 390 nm, and will detect both UVA and UVB. The output is a linear voltage relating to the UV intensity (mW/cm2), which can be read using an analog pin and then converted to a UV index. The wind vane has eight magnetic reed switches arranged in a dial; each reed switch is connected to a different-sized resistor. As the vane changes direction a magnet moves, making and breaking the switches, and therefore changing the resistance of the circuit. The magnet is sized such that when it is halfway between two reed switches both will be connected.

The three-cup hemispherical anemometer is used to calculate the wind speed. As the anemometer cups rotate, a single magnet on the device closes a reed switch, completing the circuit. This particular device has two reed switches, resulting in the circuit closing twice per revolution.

The Gateway

Next we consider the IoT gateway device, which can be split into the hardware and the software. Selecting the hardware to sit between sensors and an Internet connection requires careful thought. In its simplest form, the gateway reads data from the sensor's electronic interface and transmits it to a destination across the Internet. For this to take place, the gateway must have the appropriate electronic interfaces, a processor with memory, and either wired or wireless Internet capability. Often, using a small microprocessor such as those used by Arduino would be perfect. However, such a simple

design omits some of the critical IoT features such as Secure Sockets Layer (SSL) or Transport Layer Security (TLS), a cryptographic protocol that provides communication security over TCP.

The specific IoT application area will drive the hardware selection, for example, if it has to operate in a low-power, off-grid location, or if electromagnetic interference or radiation are a concern. If we were to focus on application-specific functionality or power consumption, the result would be a heavily customized device (eg, with custom firmware). This would provide little option for extensibility or adaptability to other applications, since all noncritical functionality would be removed to save power. If an application area requires specific hardware, such as Digital Signal Processing (DSP) or hardware video-encoding, or has a real-time requirement, this needs to be taken into consideration

Gateway Hardware

As new hardware is constantly appearing, we divide the available hardware into groups and select the preferable hardware for the gateway, based on the previously outlined requirements. It is worth revisiting the hardware market frequently to identify better-suited hardware. A microprocessor usually only offers processing power and requires RAM and storage to be added as separate chips on a PCB. This results in a lot of additional work and power, whereas a microcontroller has most of the basics embedded on a single chip; it is for this reason that we consider microcontrollers for the gateway hardware. Table 15.2 shows a selection of common microcontrollers. They tend to have a low clock-speed and are suitable for real-time applications. Programming requires low-level languages such as C or assembly. More powerful microcontrollers exist, and are often referred to as a System on a Chip (SoC), although the distinction between them is blurred. The term microcontroller often refers to low-memory, low-performance, single-chip devices, whereas a SoC usually supports an operating system such as Linux or Windows, although this is not a definition. Table 15.3 shows a selection of common SoC processors. Both a microcontroller and a SoC are of little use on their own; they need to be assembled on a PCB with other components, for example, power regulators, pin headers, and peripheral interfaces. The easiest way to build a prototype is to use a development board. The development board, often called an evaluation board, will expose all the electronic interfaces, and provide a way to program and power the chosen chip.

Gateway Software

The more powerful a microcontroller or SoC, the more complicated the software becomes, for example, there will be more interrupts and electronic interfaces. To manage the processing power efficiently, supporting libraries are required, for example, for threading.

The Arduino platform offers a great deal of libraries for all sorts of hardware, but ultimately the CPU and processing capabilities, combined with the lack of a full TCP/IP stack, make it unsuitable for a gateway. The .NetMF platform is open source and supports high-level programming languages such as C# and Visual Basic, and provides libraries for SSL, but it only runs on a limited set of hardware. Overall, .NetMF would be a good contender for the weather-station gateway. The Mbed platform is excellent for IoT devices, it supports SSL, and there is a good selection of supported hardware. Programming is done with C, which may be difficult for beginners. Commercial licensing must be considered, depending on the libraries and hardware utilized. The Mbed operating system is designed specifically for IoT, making it an excellent choice if you are comfortable programming in C.

Data Transmission

The sensor data needs to be transmitted to the backend services via the Internet. To do this we have an IoT gateway with either wired or wireless Internet access. The data is packaged for transmission and received by an online service. Traditionally the online service would use Remote Procedure Calls (RPC) [6] or Simple Object Access Protocol (SOAP) [8], but these are rather heavyweight and have been superseded by protocols such as Representational State Transfer (REST) and frameworks such as Apache Thrif.

SOAP transmits messages over different application protocols such as HTTP [9]. It is designed to allow interoperability between different services by exchanging messages, but is considered rather verbose. The default message transmission is XML but this can be substituted by binary encoding to reduce the message size. REST uses HTTP to transmit messages at the application layer. This way, any device which can communicate via HTTP can interact with a REST backend service. HTTP is well understood, and widely supported in many languages on many different hardware platforms, providing a greater level of flexibility for IoT devices. It is lightweight and supports different message payloads, for example, JSON and MIME.

Advanced Message Queuing Protocol

We have chosen to use the Advanced Message Queuing Protocol (AMQP) as our messaging protocol, as it is supported by many different server implementations, and the

client is available across platforms and programming languages. The main advantage is that AMQP is a wire-level protocol, not requiring a verbose HTTP packet. This means that data is sent more efficiently in smaller, specifically encoded packets. Our IoT device sends AMQP messages to a broker (server), which then passes them on to the readers (weather-station application code). There are many AMQP-supported architectures, ranging from publish/subscribe to content-based routing, and they are worth investigating for more complex scenarios (we are using a simple publish and subscribe architecture). We could run any AMQP-compatible server, such as Apache Qpid, RabbitMQ, or Apache ActiveMQ, but we have chosen to use the Windows Server AMQP-compatible service, called Windows Service Bus v1.0, which can run on any Windows server on-site installation.

Backend Processing

Sending data to a backend service is not the end of the IoT data story: some sources predict that the number of IoT devices will exceed 26 billion by the year 2020, excluding PCs, smartphones, and tablets [16]. This means that any IoT backend service will need to be able to scale and maintain a high level of availability, just to receive and store the data produced by IoT devices. This can cause issues. Let's assume you run a custom backend server to receive the IoT data streams. During application or operating-system updates the service may be offline, wasting valuable IoT data bandwidth. It is possible to run a failover service on enterprise-grade software and hardware to improve reliability, but this increases the complexity of the setup.

The objective of the backend services varies according to the application; generally there will be both compute and storage components. Fig. 15.4 shows the backend requirements for just the SHT15 sensor (temperature and humidity).

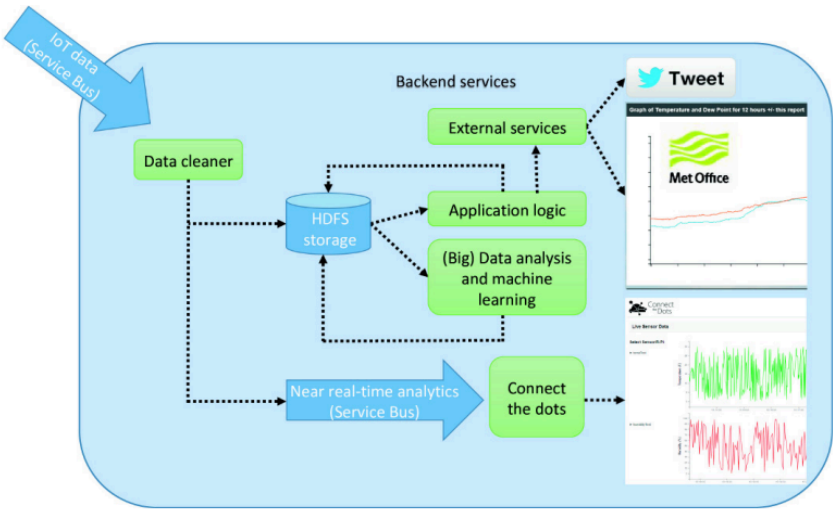

FIGURE 15.4 Backend Processing for the Weather Station SHT15 Sensor, Showing a Variety of Services Supported

Cloud or not to cloud

For the messaging infrastructure, we have chosen the Microsoft Service Bus, although we are not locked to any particular vendor since any AMQP-compatible service will work, including a cloudhosted solution. This ensures that there is a simple progression to and from a cloud-based solution, and that we can harness the benefits of a cloud-managed service. Comparing these benefits to an on-site solution depends on your expertise and current infrastructure; the cloud services are commercial and have a monetary cost. Collecting data from IoT devices will potentially result in a large volume of data being produced; although not officially Big Data, it still takes compute resources to process.

A cloud-based infrastructure provides the ability to purchase compute resources as, and when, they are required, even for short durations. This is particularly cost-efficient for cyclic or peak-demand data analytics. Cloud providers have fast interconnects between the data stores and compute resources, making them ideal for data processing/analytics. The Cloud-provider market is a fast-developing space and clearly offers benefits, especially to an IoT architecture. We are keen to harness these benefits, but fear vendor lock-in; redesigning an IoT solution to migrate a Cloud provider is unacceptable.

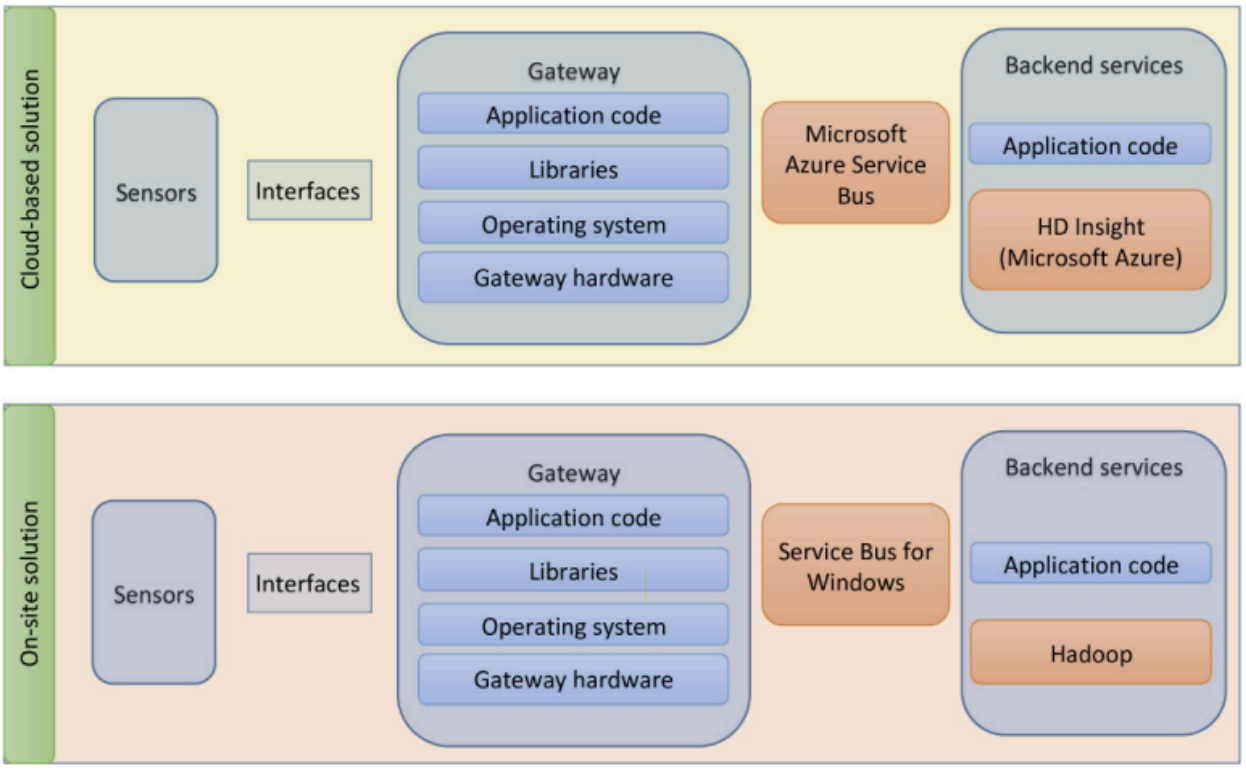

FIGURE 15.5 The Chosen Architecture is Designed to Operate Equally Well On-Site and With a Cloud Provider