Hindawi
Mobile Information Systems
Volume 2018, Article ID 2516972, 19 pages
https://doi.org/10.1155/2018/2516972

Research Article

IoT Services and Virtual Objects Management in Hyperconnected Things Network

Israr Ullah,[1] **Muhammad Sohail Khan**,[2] **and DoHyeun Kim**[1]

[1]*Computer Engineering Department, Jeju National University, Jeju City, Republic of Korea*
[2]*Department of Computer Software Engineering, University of Engineering and Technology, Mardan, Pakistan*

Correspondence should be addressed to DoHyeun Kim; kimdh@jejunu.ac.kr

Received 24 January 2018; Revised 15 April 2018; Accepted 14 May 2018; Published 13 June 2018

Academic Editor: Maristella Matera

In recent past, Internet of Things- (IoT-) based applications have experienced tremendous growth in various domains, and billions of devices are expected to be connected to the Internet in near future. The first step for development of IoT-based applications is to virtualize the physical devices by abstracting device properties in virtual objects. Later, these virtual objects can be combined to compose different services for diverse applications. Many existing systems provide virtualization service for physical devices and service composition. But, with the growth of the network, when too many devices and services are added in the IoT network, its management will become a cumbersome task. This paper presents an architecture of IoT services and virtual objects management in hyperconnected things network to facilitate the management tasks. We also have implemented a Service and Virtual Objects Management (SVOM) system prototype to effectively organize and monitor the physical devices through corresponding virtual objects and services composed in the IoT environment. The proposed system also provides interface for user interaction to perform supported control operations on selected device and check device operational and fault status. For scalability analysis of the proposed system, we have performed simulation in the OMNeT++ simulator to study impact of the IoT network size on key performance measures like response time, throughput, and packet delivery ratio. Simulation results reveal that with the growing network size, the gateway nodes become the performance bottleneck. We have also performed resources requirement analysis for virtual objects and control overhead analysis of the proposed management system. Simulation results reveal that control overhead is insignificant in normal scenarios; however, in extreme network conditions, we may have to sacrifice fewer bits which is, in fact, worth nothing when compared to the flexibility and control offered by the proposed management system.

1. Introduction

Recent developments in communication technologies have enabled the small computing devices to communicate and share information anytime, anywhere. Majority of the conducted and ongoing research is centered around the same objective, that is, to enable seamless usage of computing devices in daily life. This is made possible by integration of many underlying enabling technologies including Internet, tiny devices, sensors, and operating system, protocols, and interoperability standards. Still, there are many open challenges and opportunities, inviting research attention for investigation and exploitation; for example, the recently coined term Internet of Things (IoT) has opened a new avenue for research and development in many different fields [1]. IoT is mainly focused on connectivity of things (daily life objects with attached sensors or actuators) to the Internet so that users can remotely monitor and control a particular activity or device.

Internet of Things (IoT) is designed to attach a small communicating device with everything that we want to monitor or control using the Internet. The IoT device may be a sensing device that collects some desired information, or it can be an actuating device that can accept commands to perform some desired tasks. IoT devices have limited communication, computation, and battery power, so specialized lightweight protocols are designed for efficient resource utilization and communication over the Internet, for example, CoAP protocol [2]. IoT Systems have tremendous capabilities and applications [3]. In the recent past, many giant manufacturing and development organizations have

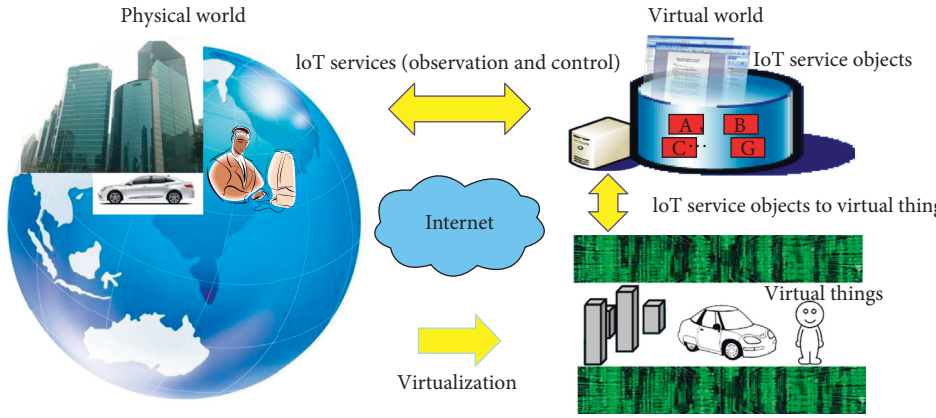

Figure 1: Physical world observation and control through virtual world in the IoT environment.

made investment in this technology to realize its potential. Many projects are initiated to address different aspects of IoT-based systems.

The development of IoT composition tools is one of the key areas of research to enable mass participation in realization of the IoT vision [4]. The objective is to enable a common user to utilize IoT-based system in order to achieve his/her desired task without any programming skills. These systems will also enable managers and designers to use the system on their own without bothering IoT experts to achieve some tasks. Many Do-It-Yourself (DIY) IoT projects are initiated in this connection to develop IoT composition toolboxes using standard business process modeling (BPM) notations. The user of this system is expected to be aware of the BPM notation and will only require slight training to utilize the system in his/her domain.

Many existing systems provide virtualization service for physical devices by creating its virtual object (VO). Later, VOs are combined to compose services which can be used to build various applications [5]. Figure 1 is a typical illustration of physical world observation and control through the virtual world in the IoT environment. Virtual world is also known as cyberworld, and the complete system is then referred as cyberphysical system (CPS). This strategy works fine when the system has limited number of registered devices and services, but when the system grows and too many virtual objects are added, devices management becomes a cumbersome task. Similarly, when too many services are composed, services management will be a challenging task. To resolve this issue, we propose "*Services and Virtual Objects Management (SVOM)*" system as a solution. The main objective of our proposed system is to effectively organize the service and virtual objects to facilitate the management task. Furthermore, we have performed simulation of the proposed system in OMNeT++ for scalability analysis to study the impact of the IoT network size on key performance measures like response time, throughput, and packet delivery ratio. We have also conducted experiments for the analysis of resources requirement and control overhead generated by the proposed system.

Rest of the paper is organized as follows. Section 2 presents an overview of some related works in the literature. Detailed description of the proposed system design and functionality is covered in Section 3. System implementation prototype is presented in Section 4. Scalability analysis of the proposed system is performed via simulation in OMNeT++, and simulation results are presented in Section 5. Towards the end, we conclude this paper in Section 6 with an outlook to our future work.

2. Related Work

Since inception, IoT has attracted too much research attention, and it is considered among the future disruptive technologies having the potential to transform every aspect of human life. Various IoT-based applications are developed for building smart environment in home, cities, health, education, security, entertainment, and industry. To enable rapid application development, several standard platforms are introduced. Philips company, known for its home appliances manufacturing, has introduced an IoT management platform for their products called Philips Hue which allows the users to easily connect home appliances to the Internet [6] for remote monitoring and control, thus realizing the concept of smart homes. The IoT.est project is an effort towards building intelligent IoT-based solution using semantic technologies to support interoperability among distributed and diverse sources [7]. They have developed a test bed for the IoT service creation and validation through several use cases. Both Philips Hue and IoT.est are focused on realization of IoT and do not consider management and scalability issues. Management functionality is given due consideration in the IoT architecture reference model (IoT-ARM) [8]. The functional model for IoT as proposed in the IoT-A project includes management functionality group having five components, that is, configuration, fault, reporting, member, and state. In this paper, we are focused on fault and configuration management in hyperconnected IoT systems with too many virtual objects and services.

Finding a suitable service required for a particular application development over Internet is a challenging task. Normally, a dynamic service discovery option does not satisfy the complex user requirements for exceeding number of requests. Reference [9] presents an idea of on-demand service creation by combining existing services to meet

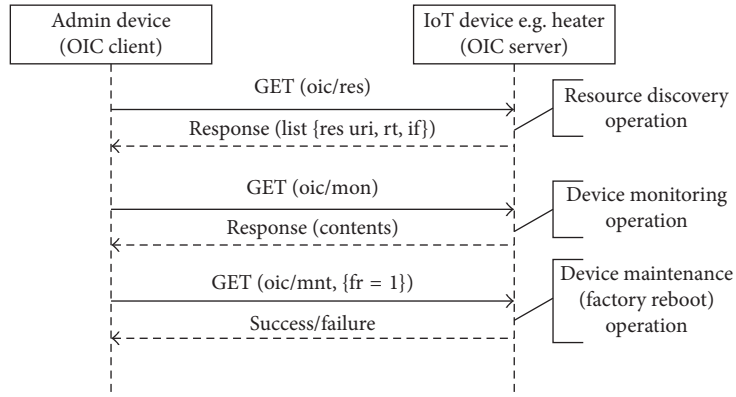

Figure 2: Typical interaction between OIC client and server for device management operation.

complex user needs using visual and interactive interface. User-desired objective is decomposed into subgoals, and the attempt is made to satisfy each subgoal in an iterative fashion. In [10], the authors present a semiautomatic mechanism for service filtering using semantic ontology. Their proposed system consists of two key components, composer and inference engine. The inference engine helps in the selection of best matching service by performing service filtering which is then connected by the composer component to produce desired service. A similar system is reported in [11] for automatic service composition. Intelligent solutions based on AI algorithms are also explored for automatic service composition in [12]. A framework for IoT objects naming and management is proposed in [13], based on the IoT-IMS platform. However, their main focus is on addressing the issue due to lack of proper rules or specification for naming things.

Device management, diagnostics, and fault recovery capabilities are considered among the core requirements in standards developed for IoT technologies. For instance, the most popular standard developed by Open Connectivity Foundation (OCF) defines guidelines for device management, diagnostic, and maintenance through two OIC core resources */oic/mon* and */oic/mnt* that shall be supported by all devices [14]. The monitoring resource (*/oic/mon*) is used for collection of device statistics, for example, packets sent, packets received, last operation time, and so on. This maintenance resource (*/oic/mnt*) has two important properties: (a) factory reset "*fr*" is used to update the device configuration to its original (default) state (factory state and equivalent to hard reboot), and (b) reboot "*rb*" triggers a soft reboot of a device while maintaining most of the configurations intact. Figure 2 shows typical interaction between OIC client and server for device management operation. The OCF is also sponsoring an open source project *IoTivity* to provide reference implementation for OCF technical specifications [15]. Likewise, oneM2M is a global organization that aims at the development of functional architecture, API specifications, security solution with interoperability for M2M communication, and IoT technologies. The oneM2M standard architecture defines three main entities, that is, common service entity, network service entity, and application entity [16]. The common service entity has two main components, that is, common service functions (CSFs) and enabler functions (EFs). CSF includes *Device Management* among core components to utilize the information for administrative purposes, that is, diagnostic and troubleshooting. The oneM2M specifies four key functions for device management, that is, device diagnostic and monitoring function (DDMF), device configuration function (DCF), device firmware management function (DFMF), and device topology management functions (DTMFs). Similarly, ITU-T also considers device management, diagnostic, and fault recovery capabilities among the common service and application support functions [17].

Recently, Do-It-Yourself- (DIY-) based projects are initiated in many fields to enable mass involvement and promote rapid development without requiring any special training. Arduino [18], Intel Edison [19], and Raspberry Pi [20] boards are very popular for customized hardware-based application development but users need to learn their respective programming languages in order to build and execute applications using these boards. Thus, only programmers can benefit from it. The SAM project by the Kickstarter [21] company is a useful initiative towards DIY-based development for normal users to build applications for fun and help students in learning basics of electronics through demonstration. Reference [22] presents an improved DIY-based service composition toolbox using the CoAP protocol. They have developed a visual interface for composition of services by combining registered virtual objects using drag and drop operations.

Many IoT projects and research contributions are geared towards prototypes development, protocols, standards, and process automation. This work is an extension of [22], and our proposed system is an effort to address scalability and management issues in the future IoT world with anticipated growth and hyperconnectivity.

3. Proposed Management System for IoT Service and Virtual Objects

The conceptual layering structure of the proposed management system for IoT service and virtual objects is given in Figure 3. There are four main layers: physical layer, virtual object layer, service composition layer, and management layer. Brief description of each layer is given below.

Physical Layer is mainly composed of two types of devices: (a) sensors are electronic and/or electromechanical

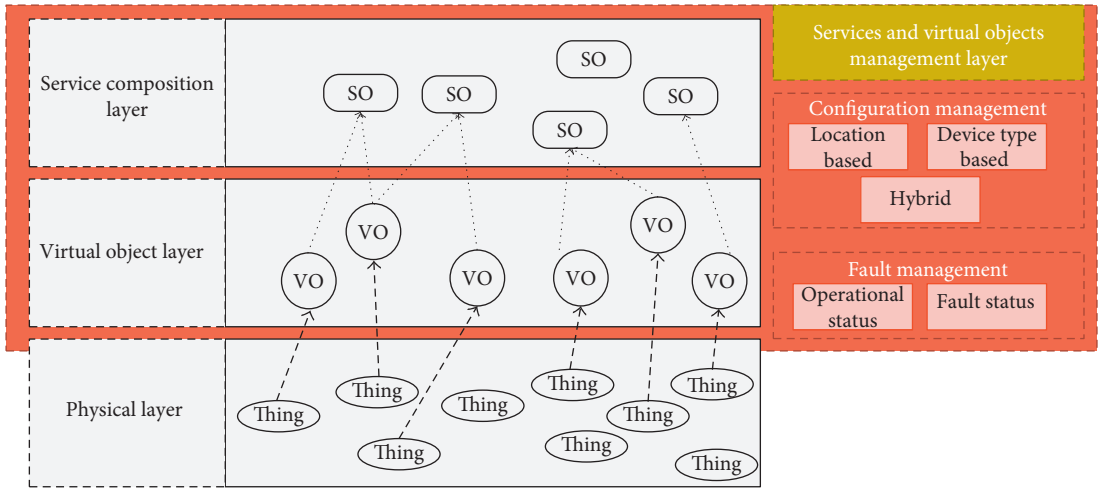

Figure 3: Conceptual layering structure of the proposed management system for IoT service and virtual objects.

devices which acquire data from their surroundings; (b) actuators are used to control the contextual parameters through their actuation. In the figure, things represent both set of sensing and actuating things.

The *Virtual Object Layer* represents the physical layer things in the form of virtual objects (VOs). VOs are the in-system representation of a thing at the physical layer which encapsulates the information associated with a physical thing. Through VO manipulation, users can interact with the system environment. We have developed a virtual device manager (VDM) application to create corresponding virtual objects for each physical device in our system. The virtual objects at the virtual object layer are utilized by the *Service Composition Layer* in order to compose service objects (SOs) by combining the functionality offered by two or more virtual objects. A simple SO may thus contain an input VO joined with an output stream to display the collected data. A more complex example of a service object would be to join a temperature sensor VO with a fan VO with the settings that when the temperature value exceeds x degrees Celsius, the fan shall be turned ON. The acquisition of the temperature value and the actuation of the fan depend on the functionality encapsulated in their corresponding VOs. The Specialized Service Composition Manger (SCM) application is used to perform tasks of this layer.

An IoT system having these three basic layers (i.e., physical, virtualization, and service composition) will work fine when the system has a limited number of registered devices and services. But, when the system grows and too many virtual objects are added, then devices management will become a cumbersome task. Similarly, when too many services are composed, then services management will also be a challenging task. To resolve this issue, we propose *Services and Virtual Objects Management* (SVOM) system as a solution. The main objective of our proposed SVOM system is to effectively organize the service and virtual objects to facilitate the management task, and it will work across the two layers (i.e., second and third layers) as shown in Figure 3.

The proposed system architecture is shown in Figure 4. Through device virtualization process, we capture properties and behavior of physical IoT devices (sensors and actuators) in form of virtual objects and stored them in a database. Information of virtual objects is then utilized by the service composition process in order to build service objects. The proposed SVOM reads stored VOs and SOs information to facilitate fault and configuration management. The SVOM system also provides interactive services for device access and control through virtual objects. Management functionality can be exposed through API to external customized applications and service composition module.

The SVOM system internal functionality regarding services and virtual objects management can be grouped into two broad perspectives as shown in Figure 5. Brief detail of each component is given in the following subsections.

3.1. Configuration Management. Under configuration management, the system provides an interface to explore specifications of registered virtual objects and services in three different ways as shown in Figure 6.

Location-based management will enable the users to display the virtual objects and services inside a selected region. Users can choose diameter of the circle and then click over the map to select the center of the region of interest.

All devices and services located within the circular region will be displayed. Users can interact with any device to see its current status and use the device in service composition. *Service/device type-based* management is useful when too many devices are registered in the system, and it becomes very difficult to find/choose a particular device needed by the user. This option simply provides a facility to the user to specify his/her desired type of device/service to visualize a particular type of devices or services. *Hybrid* is just a combination of aforementioned two approaches. Users can search a particular type of device or service within a selected region by choosing the radius of circular area and clicking over the map to define its center.

Configuration-Management algorithm pseudocode is given in Figure 7. It takes a list of registered virtual objects as an input along with optional parameters, that is, device type,

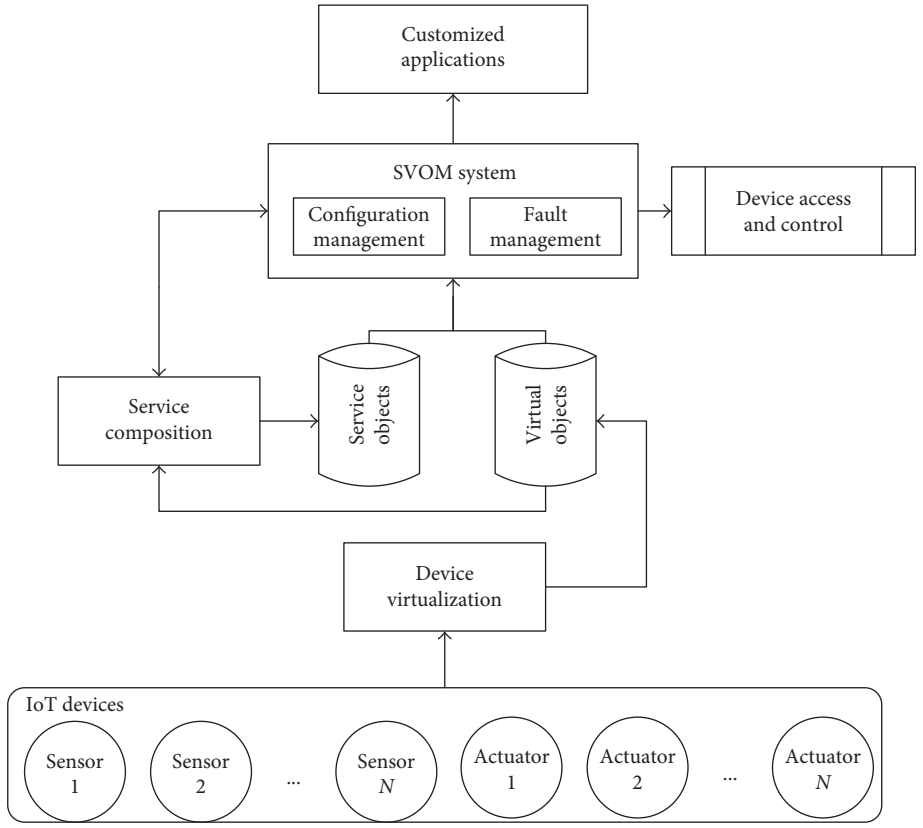

Figure 4: Proposed system architecture.

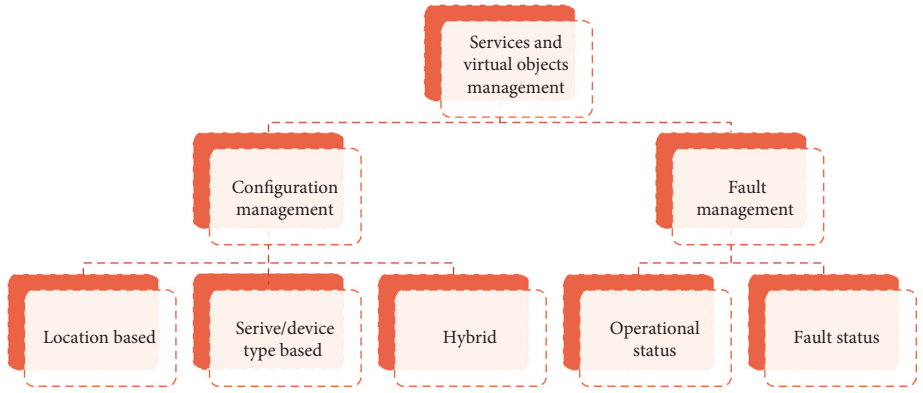

Figure 5: Structure of proposed system management functionalities.

location, and range. Every registered device type and location is checked if it meets user-desired parameter values, and then it is added into the *outputList* which is finally returned by this algorithm.

3.2. Fault Management. This component is focused on locating the working/not working and idle/busy devices and services. Through CoAP communication, we can easily get the current status of the registered devices and services. The flow diagram of the fault management is shown in Figure 8.

Operational status management can be used if the user is interested in finding the current idle/busy state of registered devices or services. If a device or service is currently being used by some application, then the status of that device or service is set to busy; otherwise, it becomes idle if not used by any application for a certain amount of time.

Fault status management will provide a list of nonfunctional devices to easily locate such devices for repair or replacement. A simple keep alive query (CoAP GET command) is sent to every device, and if no response is received within certain amount of time, then retry attempts are made for specified number of times. If all retry attempts are failed, then the corresponding device is labeled as nonfunctional. Any service using a nonfunctional device is also marked as notworking.

Fault Management algorithm as given in Figure 9 takes a list of registered virtual objects as an input and tries to

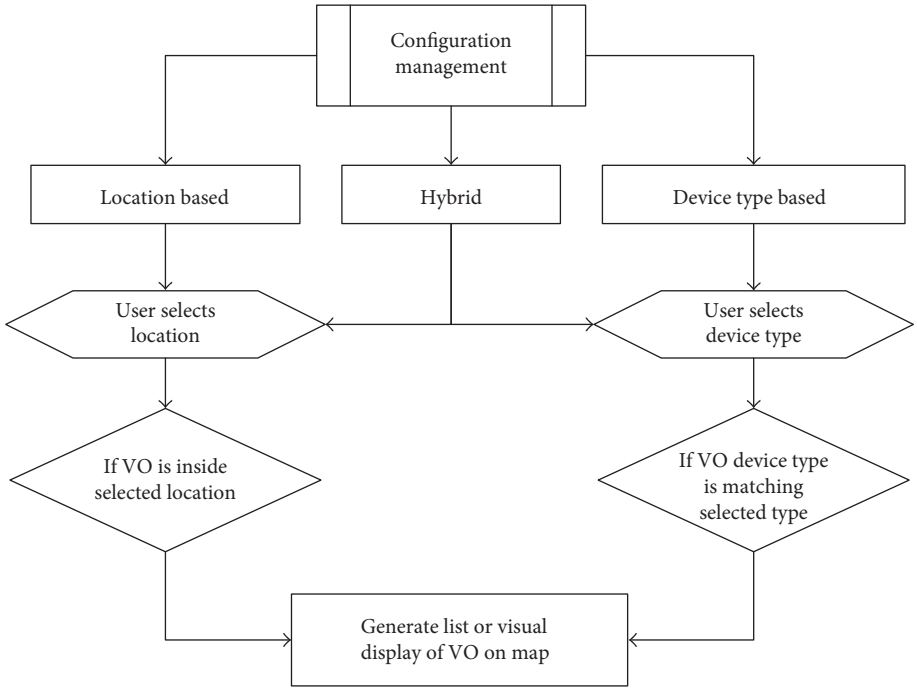

Figure 6: Configuration management flow diagram.

```
Algorithm Config-Management ()
      Input:
            InputList List of Registered Virtual Objects
            desiredType = arg[0] default (any)
            desiredLoc  = arg[1] default (center)
            searchRange = arg[2] default (max)
            OutputList  = new VOList[]
      Foreach vo in InputList
      {
            If (vo.devType = desiredType)
                  If(isDevinRange(vo.devLoc, circle(desiredLoc, searchRange))=TRUE)
                        OutputList.Add(vo)    %Make output List
      }
      return OutputList
```

Figure 7: Pseudocode for configuration management algorithm.

connect to each device using its URI to get its current status information. Threshold defines a maximum number of attempts for connection establishment. If no response is received, then the respective device is considered to be "*Not Working*." The output list holds fault status information with each virtual object and can be used by the application to identify defective devices for repair or replacement.

4. Development of Proposed Management System Prototype

We have developed a prototype for management of IoT services and virtual objects in Visual Studio 2015 using C#. CoAP.NET library [23] is used for accessing IoT resources attached with Intel Edison Board. CoAP.NET implementation is based on the Californium (Cf) framework implemented in Java. Microsoft SQL Server is used to store profile information of registered IoT services and virtual objects. Table 1 summarizes the key functions list of our proposed system with brief explanation.

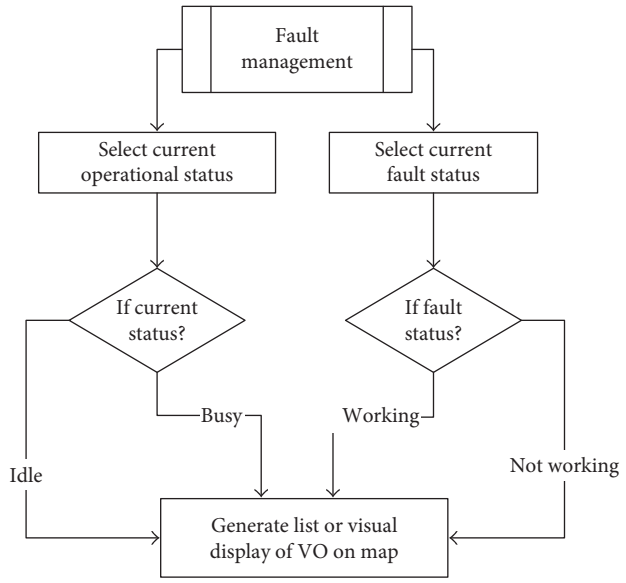

Figure 8: Fault management flow diagram.

In the following subsections, we present the working of three main modules of our proposed SVOM system, that is, virtual objects management, service objects management, and VO-based interactive access to physical devices.

4.1. Virtual Objects Management. Screenshot of the virtual objects management system interface is given in Figures 10 and 11. The interface has two main tabs. The first tab is used to display virtual objects information for registered IoT

```
Algorithm Fault-Management (InputList List of Registered Virtual Objects)
      //Create empty List for holding output
      OutputList= new VOStatusList[]
      Foreach vo in InputList
      {
            FaultStatus="Not Working"        %assume device is not working
            OpStatus="Null"                  %assume device is not working
            attemptNo=0
            While attemptNo < Threshold      % attempt to get device status
            {
                  If (getStatus(vo.DeviceUri) = TRUE)
                        FaultStatus="Working"      %TRUE when device is working
                        If(getLastAccessTime(vo.DeviceUri) < MaxActiveTime)
                              OpStatus="Busy"
                        Else
                              OpStatus="Idle"
                        Exit While Loop
                  Else
                        attemptNo++
            }
            OutputList.Add(vo, FaultStatus, OpStatus)   %Make output List working
      }
      return OutputList
```

Figure 9: Pseudocode for fault management algorithm.

Table 1: List of SVOM system APIs.

Category	API function	Explanations
Common	GetSensorsNum	Get total number of registered sensing devices in a system
	GetActuatorsNum	Get total number of registered actuating devices in a system
	GetMapServiceURI	Get map service information for visual display of virtual objects
	ConfigUI	Build user interface for a selected virtual object to provide interactive access and control
	ExecCommand	Execute supported operation on selected device using its virtual object profile
Virtual objects management	GetVOStatus	Get current status of virtual objects, for example, idle, busy, functional, and nonfunctional
	AddVObject	Add virtual object profile information to the database
	UpdateVObject	Modify existing virtual object profile information in the database
	DelVObject	Delete the selected virtual object from the database
	VOConfigManagement	Perform configuration management on the virtual object to return the selected type of virtual objects in the given area
	VOFaultManagement	Perform fault management on registered virtual objects to identify faulty and operational devices
Service objects management	GetSOStatus	Get current status of the service object by checking the status of its associated virtual objects
	AddSObject	Add service object profile information to the database
	UpdateSObject	Modify existing service object profile information in the database
	DelSObject	Delete the selected service object from the database
	SOConfigManagement	Perform configuration management on service objects to return the selected type of service objects in the given area
	SOFaultManagement	Perform fault management on registered service objects to identify faulty and operational services

devices in a tabular format as shown in Figure 10. The second tab is used to visually display IoT devices in the form of icons over the map at their respective location as shown in Figure 11. Icons are chosen as per the device type for making it easy to visually identify the desired type of device in a given region/area over map. Before displaying the VOs information, first we need to connect to the database to read virtual objects information. Virtual objects' view can be changed from list to map by clicking the respective tab. Once VOs information is loaded, then various types of management tasks can be performed by the user. Brief description of management tasks related with VOs is given below.

In the application windows given in Figure 10, the first group box (top left) holds controls to perform configuration management-related tasks. Users can display all VOs registered in the system. When devices are geographically spread over a large area (multiple cities), users can choose to display the device only in a particular region. This is done by selecting the radius of the desired area and clicking over map to specify center of the selected region, then only devices

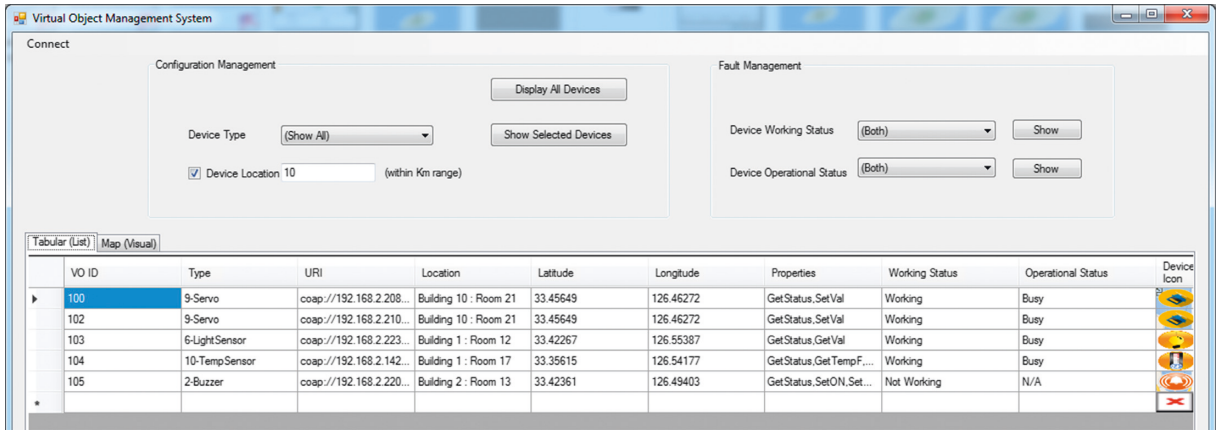

Figure 10: VOMS showing devices within a circular region of radius 10 km (tabular view).

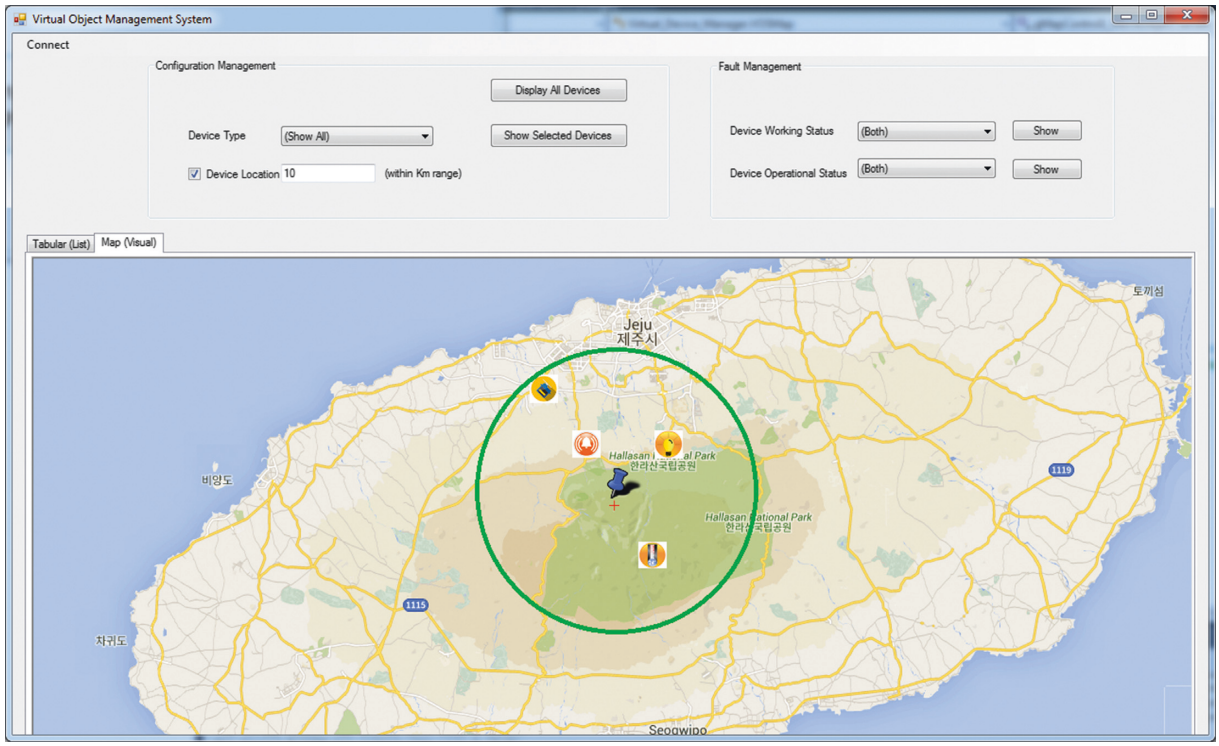

Figure 11: VOMS showing devices within a circular region of radius 10 km (view over map).

located within selected area will be displayed. During registration of devices in the VDM, we need to provide exact location coordinates (latitude and longitude) in order to facilitate management tasks. Furthermore, if there are too many devices in a particular area, then users can view only selected type of devices. Thus, users can view all or just selected type of VOs using this interface.

In Figure 11, through the fault management panel (top-right group box), users can choose to view VOs based on their working status (working, not working, or both). The virtual object status is set not working if no response is received from the corresponding physical device after specified number of attempts. A list of not working devices can easily be generated and handed over to the system maintenance department to quickly locate and resolve the issue. Common reasons for not working status of the device include network connectivity issues and device power depletion. Furthermore, users can also choose to view VOs having selected operational status (busy, idle, or both). The device status is set to busy if it is currently being used by a running application. If no query is received from any application within a specified amount of time, then the device status is set to idle. A list of idle state devices can easily be generated to facilitate users while creating a new service in order to use idle devices to have better response time. This also helps in load balancing while restricting the shared access to a particular device.

4.2. Service Objects Management. Figures 12 and 13 present screenshots of the Service Objects Management System

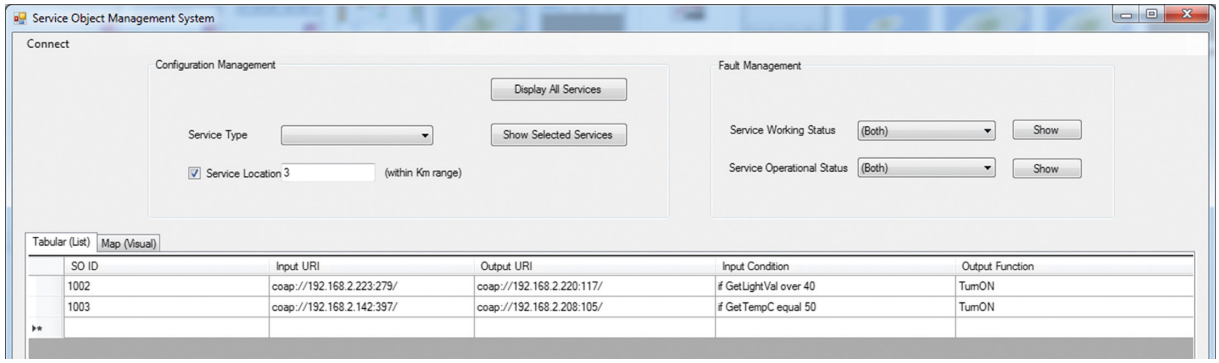

Figure 12: SOMS showing services within a circular region of radius 3 km (tabular view).

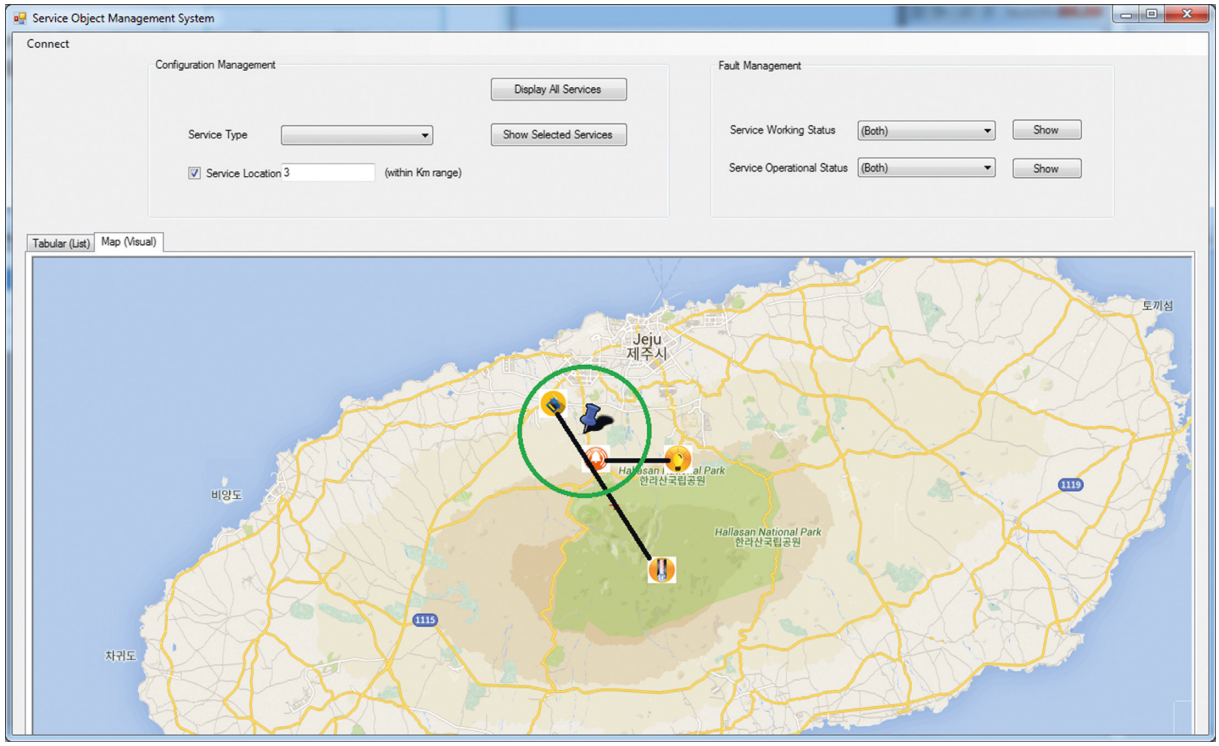

Figure 13: SOMS showing services within a circular region of radius 3 km (view over map).

interface. Like VOMS, it also has two main tabs. The first tab is used to display service objects information of composed service in a tabular format as shown in Figure 12. The second tab is used to visually display registered services in the form of connected icons by joining the constituent VOs over map at their respective location as shown in Figure 13. A service is composed of one or more VOs and icons of those VOs are connected to express a service over map. Before displaying the SOs information, first we need to connect to the database to read service objects information. Service objects view can be changed from list to map by clicking on the respective tab. Once SOs information is loaded, then various kinds of management tasks can be performed by the user. Brief description of management tasks related with SOs is given below.

The first group box (top left) in the application interface given in Figure 12 holds control to perform configuration management related tasks. Users can display all SOs registered in the system. When system devices are geographically spread over a large area (across multiple cities), users can choose to display the device only in a particular region. This is done by selecting the radius of the desired area and clicking over the map to specify the center of the selected region; then only services located within the selected area will be displayed. During composition of service in SCM, we need to connect appropriate VOs in order to facilitate management tasks. Furthermore, if there are too many services in a particular area, then users can choose to view service having the selected type of VOs as its input or output. Thus, users can view all or selected type of SOs using this interface. We can see in Figure 13 that only 2 services are available within the selected region of radius 3 km. A service is listed if at least one of its constituent devices is located within the selected region. Brief description of the two services is given in Table 2.

Table 2: Details of the selected service.

Service ID	Input device	Output device	Execution condition
1002	Light sensor	Buzzer	If GetLightVal over 40, then TurnON buzzer
1003	Temperature sensor	Serve motor	If GetTempC over 50, then TurnON serve motor

Service ID 1002 is an example of a typical alarm system for lighting control in a certain environment. It has a light sensor as its input and a buzzer as its output device. This service simply gets illumination data from the light sensor and will turn on the buzzer if its value is over 40. Service ID 1003 is an example of a typical temperature control system for monitoring overheating of a certain environment. It has a temperature sensor as its input and a serve motor (an actuator) as its output device. This service will get temperature data from the temperature sensor and will turn on the serve motor if its value is over 50. The serve motor can be any actuating device, for example, an air conditioner or a fan that needs to be turned ON in order to control overheating.

In Figure 12, using the fault management panel (top-right group box), users can choose to view SOs based on their working status (working, not working, or both). A SO status is set not working if one of its constituent VO devices is not responding. List of not functional services can easily be generated and handed over to system developers and maintenance department to quickly locate and resolve the issue. Common reasons for service failure include network connectivity issues and device power depletion. Furthermore, users can also choose to view SOs having selected operational status (busy, idle, or both). A SO status is set to busy if it is currently being used by a running application. If no query is received from any application within a specified amount of time, then corresponding service status is set to idle. A list of idle state service can easily be generated to facilitate users while creating a new application in order to utilize existing service and reduce development time. This also helps in avoiding creation of duplicate services.

4.3. Virtual Object-Based Interactive Access to Physical Devices. In this section, we show how users can interact with IoT devices using virtual objects in the proposed management system. Users can select any device to perform supported interactive operations through the configuration management. As shown in Figure 14, users will select the desired device. Afterwards, the current operation status of the device is checked, and if the device is not working, then appropriate error message is displayed. If the device is working properly, then the application uses the device profile information and builds a user interface to allow users to perform supported interactive operations. User commands are sent to the respective device using device URI.

We have tested temperature, humidity, and wind sensors along with the fan control using Intel Edison board and CoAP protocol. For the sake of brevity, we only show interaction with an IP camera used for experimental purposes as demonstration. In order to create a virtual object for the IP camera, we build its profile in *Virtual Device Manager* application. A typical virtual object profile for the camera has information as shown in the XML format in Figure 15.

The virtual object profile for IoT devices is a generic description which is shared by the individual devices. We have developed a specialized tool *Virtual Device Manager* application for creating the virtual object profile for any IoT device. End users can use this application to create and publish virtual object profile information of their respective devices. Moreover, it is worth mentioning here that the virtual object profile shall contain all those properties and operations which are actually supported by the corresponding IoT device. For instance, the virtual object profile given in Figure 15 consists of six properties which are the six different operations supported by the respective IP camera. Depending upon the requirement, users can expose all or partial functionality of their respective devices. Thus, two exactly same devices can provide different functionality, but in that case, the corresponding virtual profile of each device will be different. Finally, if the device functionality is changed, then users (owners) can simply update the virtual object profile for the corresponding device on management server using the designated application. All services depending upon or using the old virtual object profile may also require modifications.

A camera profile holds information about camera properties including its URI and location information. Properties sections hold supported operations for the designated camera. We can get stream from camera for live view and can save the stream in a video file for later retrieval and analysis. Furthermore, various control operations are also included to move the camera in four directions to get a desired view. For IP camera access and control, we need to double click on its icon in the SVOM system, and then a pop-up window is opened for the camera view. After establishing connection with the camera, a frame queue is created to hold frames from streaming object. Streaming object runs in a separate loop to get continuous frames from camera as per frame rate which are then stored in the frame queue. If we want to save the stream, then camera thread is initiated to get the frames from the queue and store it. This sequence of operations is illustrated in Figure 16. Once streaming is initiated, then users can execute various supported commands to move the camera through its virtual object.

In the SVOM system, when a list of registered devices is displayed, then by hovering mouse over the device icon, a tool tip is appeared showing the name of virtual object. Double click on IP camera icon to open a new pop-up window for interaction as shown in Figure 17. Windows form controls in the pop-up window is dynamically linked with corresponding properties of selected VO. We use Afroge.NET library for getting live camera stream. Cisco WVC200 Wireless-G PTZ Internet Video IP camera is used in this experiment, and this camera model is Onvif complaint and supported by Ozeki Camera SDK to perform

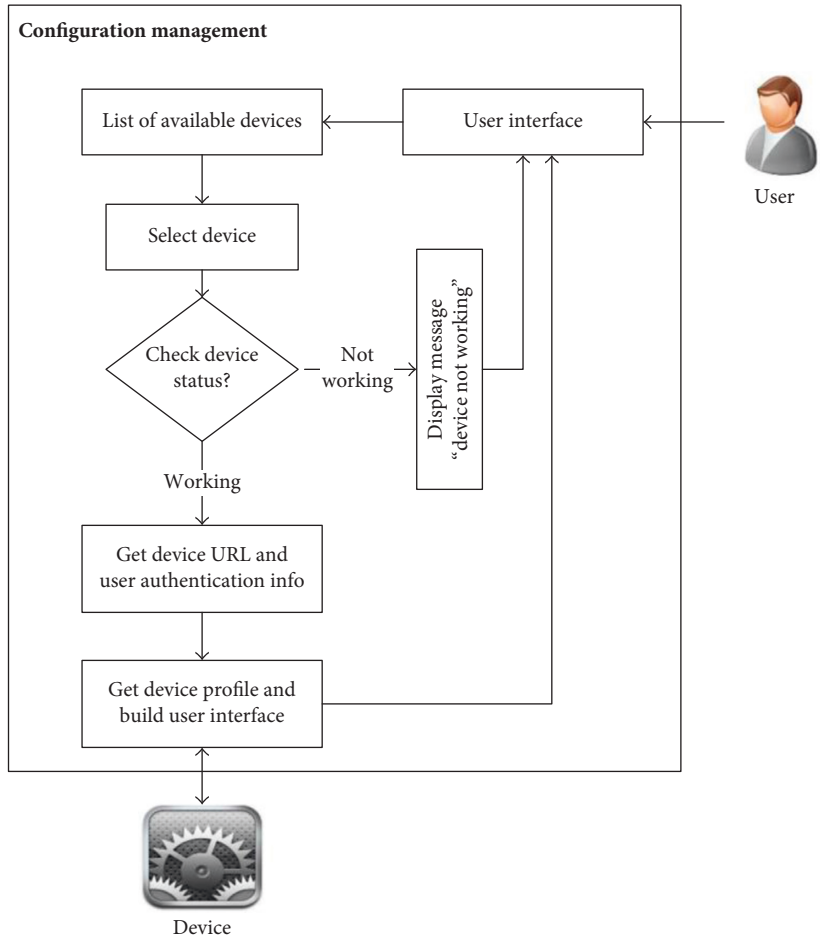

Figure 14: Mechanism for virtual object-based interaction with physical devices using configuration management.

```
<?xml version="1.0" encoding="UTF-8"?>
- <Devices>
  - <Device>
      <VO_ID>16</VO_ID>
      <Uri>coap://10.102.42.125</Uri>
      <Type>IPCamera</Type>
      <Location>Room123</Location>
      <LocLat>33.455</LocLat>
      <LocLong>126.74</LocLong>
    - <Properties>
        <P>GetStream</P>
        <P>SaveStream</P>
        <P>MoveUp</P>
        <P>MoveDown</P>
        <P>MoveLeft</P>
        <P>MoveRight</P>
      </Properties>
    </Device>
</Devices>
```

Figure 15: Typical IP camera profile (xml view).

interactive operation on the camera. For sake of demonstration, only camera movement operations are implemented, and zooming buttons are disabled as this camera VO profile (given in Figure 15) does not have zooming operations. Users can record the stream on some online storage for later retrieval and analysis. Furthermore, we have developed customized interfaces for known types of registered IoT devices that share same profile information. For new and unknown type of IoT devices, the system support dynamic creation of interactive interface where elements on the interface is created during run time. For every property in the registered device virtual profile, we create a button on the interface with a clickable event. When a button is clicked, the event will be triggered to execute corresponding operation on the IoT device. This will just provide end users to test properties included in the virtual profile of registered IoT devices.

5. Scalability Analysis of Proposed System via Simulation

We have performed simulation of proposed system in OMNeT++ for scalability analysis to study the impact of IoT network size on key performance measures like response time, throughput, and packet delivery ratio. In addition, we also tried to analyze resource requirement for proposed management system along with analysis of control overhead introduced due to centralized management. OMNeT++

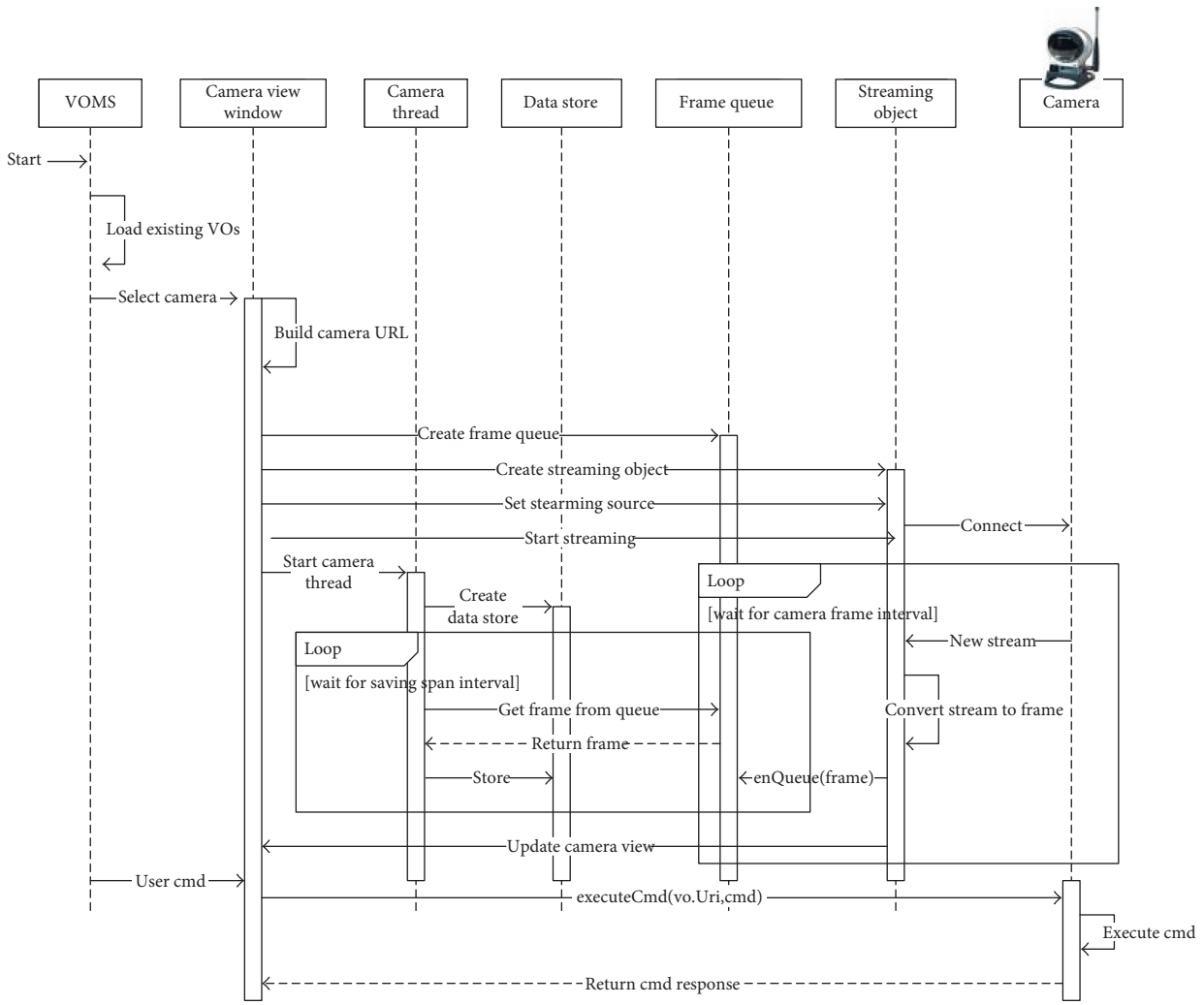

Figure 16: Sequence diagram for IP camera access and control.

(Objective Modular Network Testbed in C++) is a well-designed discrete event simulation environment [24]. To perform network simulation in OMNeT++, an open-source framework INET is developed which includes various protocols for wired, wireless, and mobile networks [25]. These simulations are performed in OMNeT++ 5.2 with INET framework version 3.6.

5.1. Protocol Implementation in OMNeT++. To achieve proposed management system functionality for scalability analysis, we have implemented two application layer protocols in OMNeT++; one for management server and another for IoT devices. Figure 18 shows typical simulation setup for evaluation of the proposed management system. IoT devices are connected to the local gateway node through wireless link with data rate of 54 Mbps and gateway nodes are connected to management server over a 100 Mbps wired link through intermediate routers. For sake of illustration, graphical view of nodes' internal layered architecture is highlighted (red color rectangle) in Figure 18 for management server and an IoT node. At management server, an instance of management protocol *mgtServerApp[0]* is used at the application layer (highlighted in yellow rectangle). Likewise, an instance of IoT device protocol *ioTDeviceApp[0]* is used at application layer of IoT nodes. At simulation start-up, all IoT nodes send registration request to management server containing its profile information. Upon registration, management server creates a virtual object (VO) for each IoT device to hold its profile information which is also used for onwards communication and control with corresponding IoT devices. Screenshot given in Figure 18 is taken when registration process of all IoT devices was completed as indicated by the tag on application layer of management server *Registered VOs* = 10 (means 10 IoT devices are registered). Similarly, the tag on application layer of IoT device is changed to *Registered* = *yes* indicating its registration with server is successfully completed. After registration, management servers can send various commands to registered IoT devices using its VO, for example, to get its operational status, to initiate data transmission, and so on.

5.2. Performance Measures and Simulation Parameters. We have selected three key performance measures, that is,. response time, throughput, and packet delivery ratio to study the impact of IoT network size on these metrics.

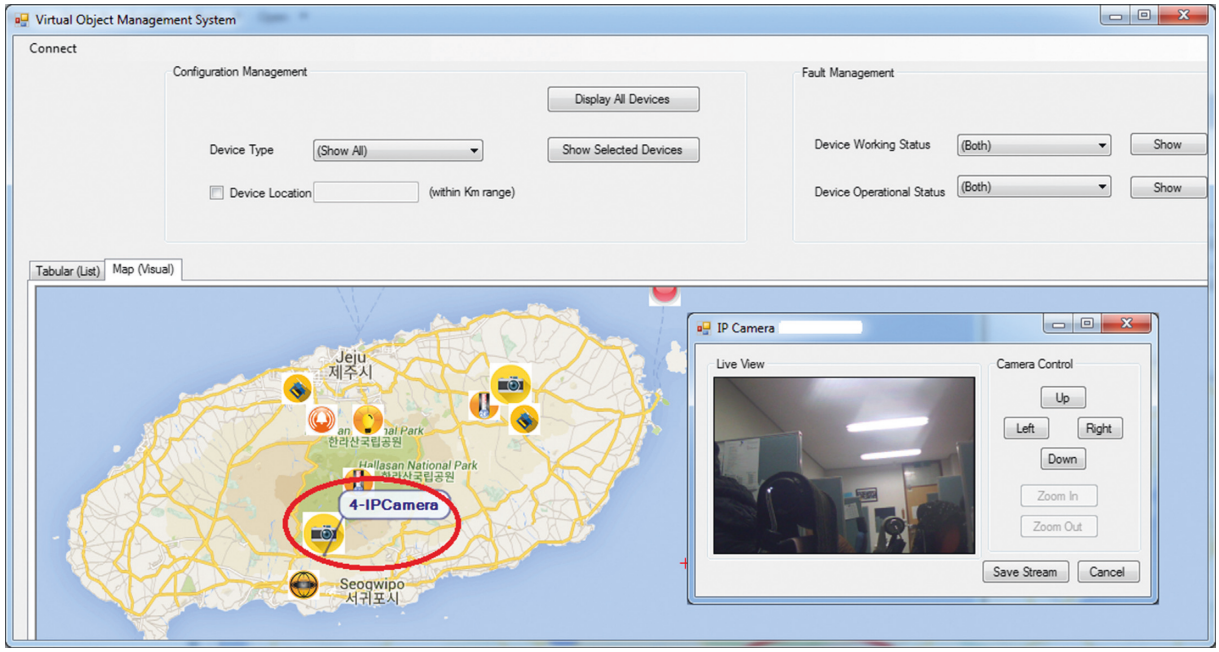

Figure 17: Interactive window for IP camera with live streaming, and corresponding virtual object is shown in red circle.

Response time is computed by measuring the time taken to receive a response from an IoT node to a query generated by management server. Throughput is the total data received at server in per unit time from all sources, that is, active IoT devices. Packet delivery ratio is computed by considering the ratio between total number of data packets received at server versus the total number of data packets generated in the network. Table 3 presents the configuration for various parameters used in simulation.

5.3. Performance Analysis. We have conducted simulations in two phases: (a) with single gateway node and (b) with multiple gateway nodes. Following subsection presents results for each phase with brief discussion.

5.3.1. Results with Single Gateway Node. First, we performed a set of experiments by varying the number of IoT devices in the network with varying packet sending rates. In this phase, all IoT devices are connected with management server through a single gateway node. The gateway node is connected with management server via a 100 Mbps wired link through a router as shown in Figure 18. To compute IoT device response time, management server sends an operational status query to all IoT devices through a broadcast message. In response to the status query message, every IoT device sends its status information (idle or busy) to management server. First, experiments are performed with 30 IoT devices in the network. The average response time was about 5 msec with network size of 30 nodes. Similar experiments were performed with network size of 60, 90, 120, and 150. Linear growth in average response time was observed with growing network size as shown in Figure 19(a). There were two choices to evaluate the impact of network size on throughput: (a) increase the number of sources, that is, active IoT device to generate more data in the network and (b) increase the data rate, that is, packets/sec while keeping number of sources fixed. The latter approach was used here to evaluate network performance in terms of throughput. For throughput calculation, we have used fixed network size of 60 nodes with 6 sources, that is, active IoT devices. For these simulations, the data rate for each active IoT device was varied from 200 to 1000 packets/sec (with step of 200). With six sources, a date rate of 200 packets/sec, and a packet size of 1000 bytes, data generated in the network become $6 \times 200 \times 1000 \times 8/10^6 = 9.6$Mbps. Data throughput computed at management server node was also around 9.6 Mbps as shown in Figure 19(b). It means that with a data rate of 200 packets/sec, packet delivery ratio in the network was 100% which is confirmed from packet delivery ratio graph given in Figure 19(c). Throughput results are given in the box-plot graph format, which also captures the variation in throughput during simulation time. When packet sending rate was doubled, that is, 400 packets/sec, we get data throughput around 15 Mbps, not doubled as expected. Packets delivery ratio graph in Figure 19(c) shows that only 76.40% packets are delivered with a data rate of 400 packets/sec; therefore, throughput was not doubled. Further increase in data rate to 600 packets/sec results slight improvement in throughput results, that is, 16.5 Mbps but no improvement in throughput was observed for date rate of 800 and 1000 packets/sec as shown in Figure 19(b). On the other hand, the results given in Figure 19(c) show significant degradation in packet delivery ratio with an increase in data sending rate. Upon investigation, it was revealed from simulation trace files that with increase in data sending rate, most of the packets gets dropped at MAC layer of IoT device nodes. Packets are dropped because nodes are unable to get access to the shared wireless channel due to contention. This is due to the fact that all IoT devices are using single gateway

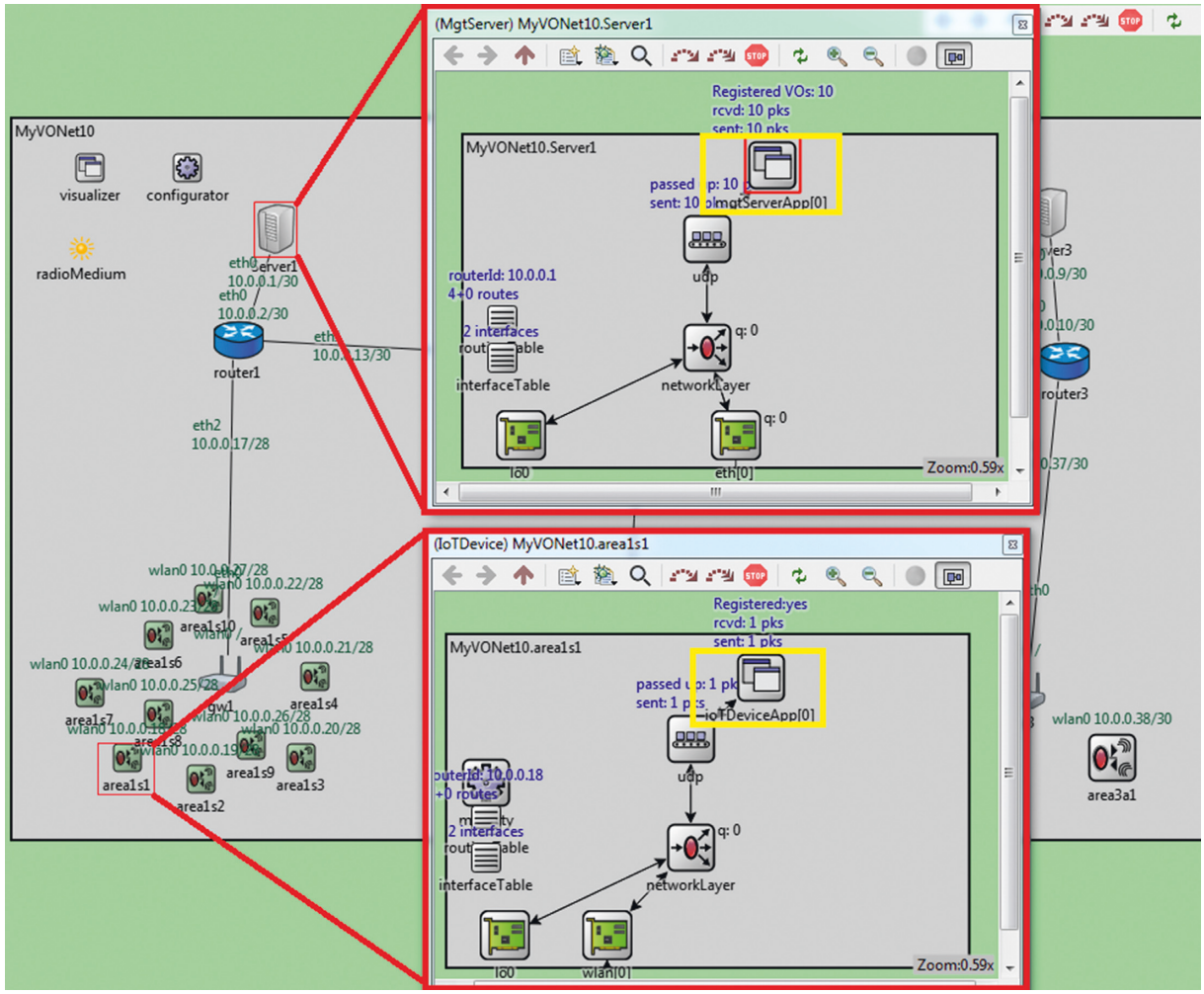

Figure 18: Simulation setup in OMNeT++.

Table 3: Simulation parameters.

Parameter	Value/range	
	IoT device(s)	Management server
Nodes count	30, 60, 90, 120, 150	1
Gateway nodes	1, 2, 3	
Application layer	IoTDeviceApp	MgtServerApp
Packet size	1000 bytes	NA
Packet sending rate (per node)	200, 400, 600, 800, 1000 packets/sec	NA
Transport layer	UDP protocol	
Routing layer	Fixed routing IP protocol	
MAC layer	IEEE802.11	Ethernet
Bit rate	54 Mbps	NA
Communication range	100 m	NA
Area size	1000 m × 1000 m	
Mobility type	No mobility	
Simulation time	20 sec	

node which is overloaded. Simulations results reveal that with growing network size and data load, the gateway node becomes the performance bottleneck. Hence, we performed another set of experiments with multiple gateway nodes to verify this assertion.

5.3.2. Results with Multiple Gateway Nodes. In the second phase, the same set of experiments is repeated with multiple gateway nodes, that is, 2 and 3. Gateway nodes are placed in the network at suitable locations to have uniform distribution of IoT nodes.

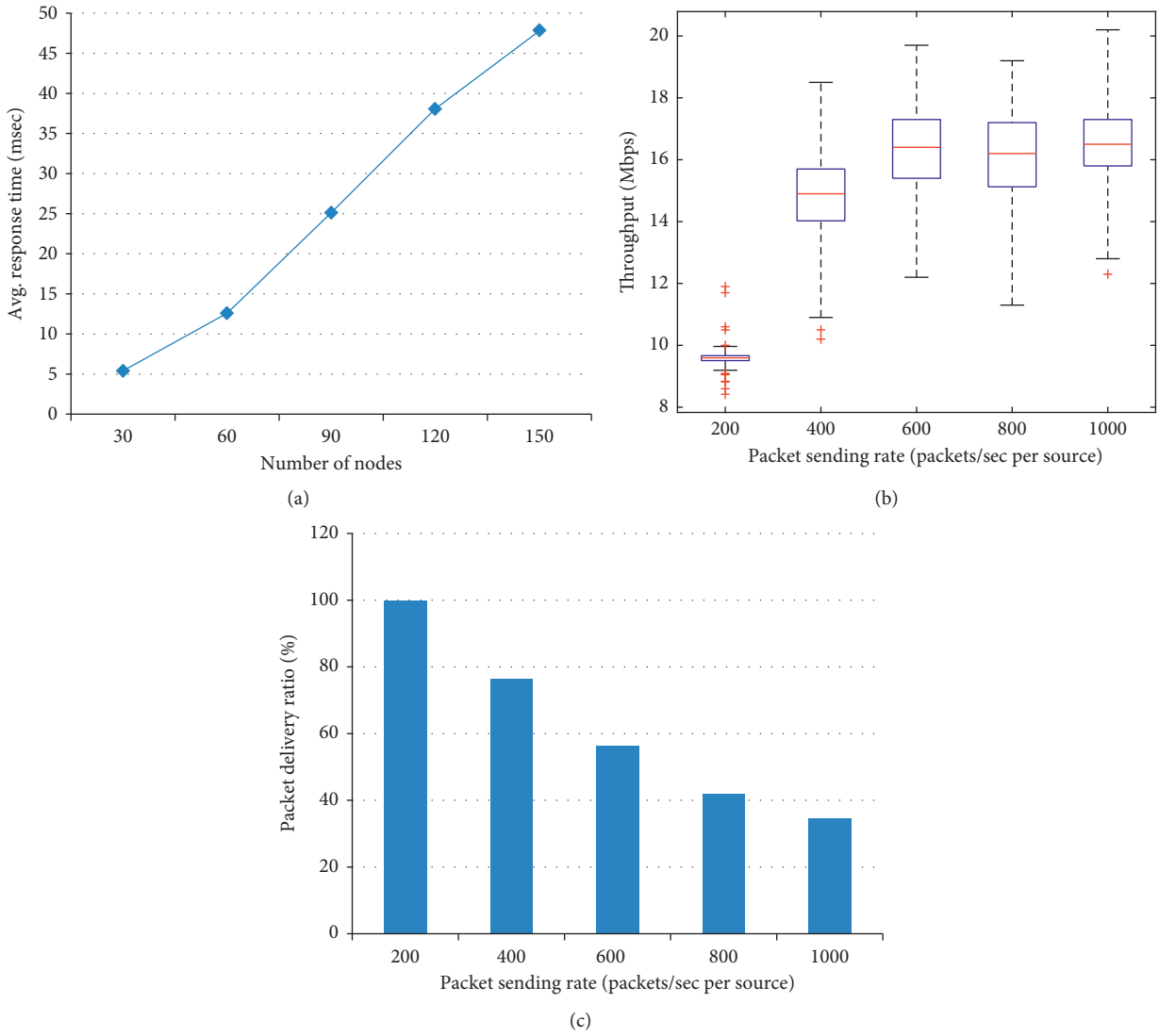

FIGURE 19: Simulation results with single gateway node. (a) Average response time of IoT device with respect to varying number of nodes in the network. (b) Network throughput with respect to varying packet sending rate. (c) Packet delivery ratio in the network with respect to varying packet sending rate.

With 30 IoT devices in the network, the average response time was almost the same (around 5 msec) with 1, 2, and 3 gateway nodes. However, with increase in network size from 60 to 150 nodes, significant difference between response time was observed as shown in Figure 20(a). With 2 gateway nodes and 150 nodes in the network, the average response time recorded was around 20 msec which is almost 60% reduction in average response time as compared to the case of single gateway node. Average response time was further reduced to 10 msec when we used 3 gateway nodes with network size of 150 nodes (almost 77% reduction in average response time as compared to the case of single gateway node). For throughput calculation, we have used fixed network size of 60 nodes with 6 sources, that is, active IoT devices. With 2 gateway nodes and 6 active IoT nodes, the recorded throughput was (9.6, 19.2, 28.8, 37.6, and 38.1 Mbps) with packet sending rate (200, 400, 600, 800 and 1000 packets/sec), respectively, as shown in Figure 20(b). This shows linear increase in throughput until packet sending rate of 800 packets/sec. However, there is no significant difference between throughput for 800 and 1000 packets/sec. Packet delivery ratio results for two gateway nodes in Figure 20(d) indicate that 100% packets were delivered for packet sending rate 200–600 packets/sec. However, for 800 and 1000 packets/sec, packets delivery ratio was 97.88% and 79.30%, respectively. In other words, increase in packet sending rate from 800 to 1000 packets/sec only results in packet loss in the network, as there was no improvement in the throughput. Finally, with 3 gateway nodes, linear increase in throughput results was observed with increase in packet sending rates as shown in Figure 20(c). Packet delivery ratio results with three gateway nodes are given in Figure 20(d) indicating that 100% packets were delivered for packet sending rate 200–1000 packets/sec. Thus, with given network conditions and data load, three gateway nodes seem to be good choice. Significant improvement in results is observed by using multiple gateway nodes.

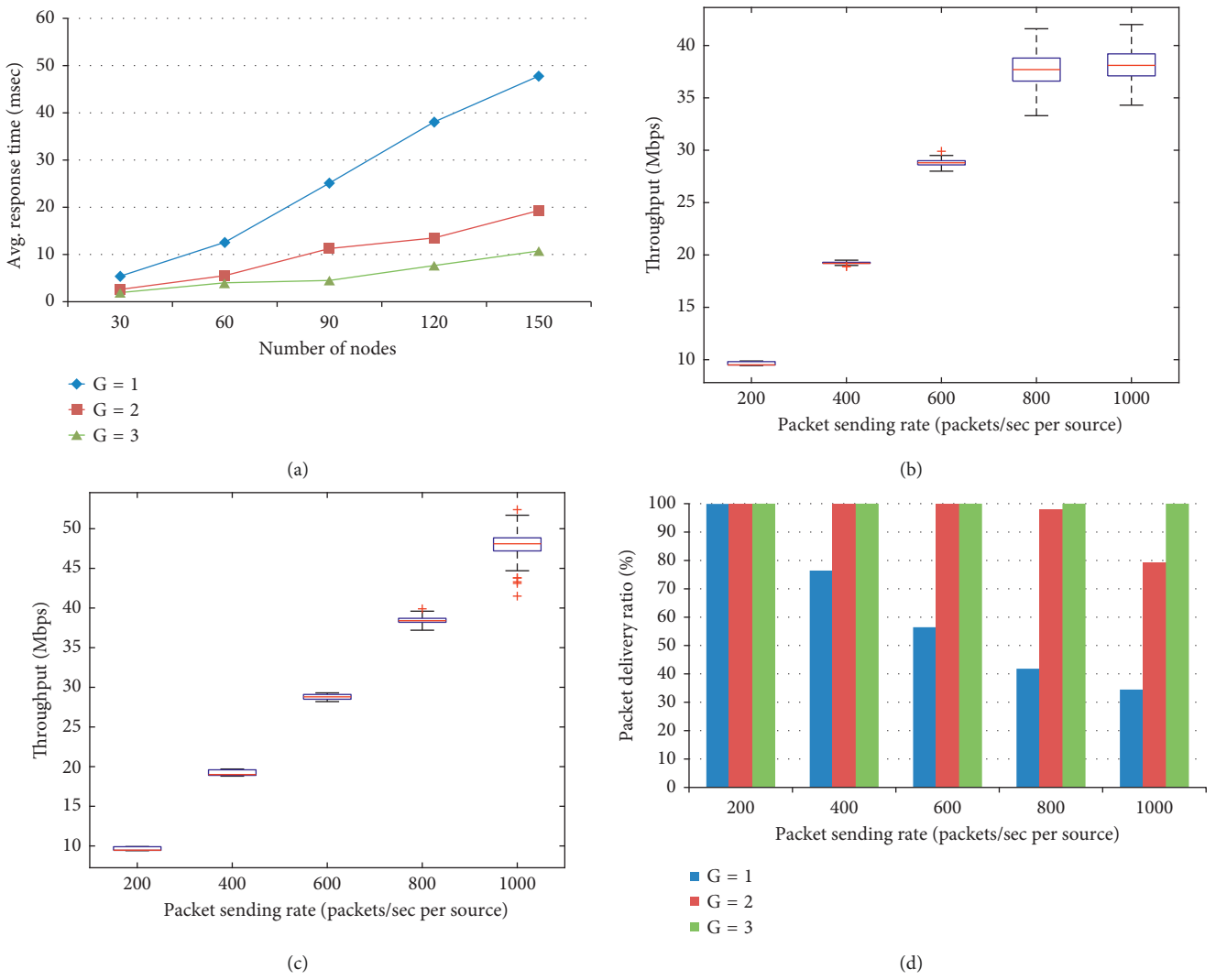

FIGURE 20: Simulation results with multiple gateway nodes. (a) Average response time of IoT device with respect to varying number of nodes in the network. (b) Network throughput with two gateway nodes. (c) Network throughput with three gateway nodes. (d) Packet delivery ratio in the network with respect to varying packet sending rate.

5.3.3. Analysis of Virtual Objects Resources Requirement. We have conducted another set of experiments through simulations to analyze resources requirement (memory) at management server with growing network size. As stated earlier, virtual object for any IoT device is created using a specialized tool *Virtual Device Manager* application. Figure 15 shows typical representation of virtual object profile (IP camera in this example) in XML format. It is worth mentioning here that structure of virtual object profile will remain the same; however, memory requirement may vary depending upon the recorded properties of registered IoT devices. In the simulation setup, we consider seven different types of IoT devices, that is, camera, serve motor, temperature sensor, green led, axis sensor, light sensor, and buzzer where memory requirement for virtual object profile of each type device is 170, 154, 122, 122, 122, 106, and 122 bytes, respectively. Multiple simulation runs were tested with growing number of IoT devices from 200 to 1000 (step size of 200). In each simulation run, IoT devices of aforementioned types were created using uniform distribution. Memory requirement for holding virtual objects profile information was recorded at management server after completion of registration of all IoT devices. Figure 21 shows the total memory required for holding virtual objects profile information at management server with growing number of IoT devices.

Memory requirement at management server exhibits linear growth with growing number of IoT devices in the network as depicted in Figure 21. Computation requirement may vary depending upon the type of operation that we want to perform. Computation complexity of the current supported operations in existing system is between $O(n)$ and $O(n^2)$. However, complex operations will require more computation and will certainly demand intelligent and distributed solutions as conventional approaches will not be scalable when performed at large scale with millions of registered IoT devices.

5.3.4. Analysis of Control Overhead. To analyze overhead traffic due to centralized management server, we have conducted several set of experiments. Experiments are performed with

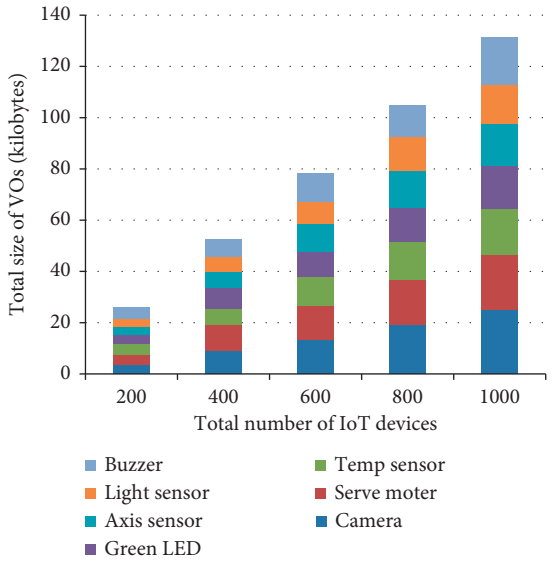

FIGURE 21: Analysis of memory requirements with growing number of IoT devices.

varying network size and data sending rates. Proposed management system generate various control packets in the network which includes initial registration packets, periodic status enquiry messages for registered IoT devices, and command message to initiate and terminate various active flows in the network. To quantify network overhead of proposed system, normalized control overhead (NCO) metric is used which is computed as below:

$$\text{normalized control overhead}\,(\%) = \frac{\text{control packets}}{\text{data packets}} \times 100. \tag{1}$$

Normalized control overhead (NCO) is the ratio between total control packets and total number of data packets generated in the network. As given in (1), this metric measures the control overhead in terms of control packets required for generating 100 data packets. Control packets generated in network mainly depends upon the network size and number of active flows, while NCO depends on two factors, that is, control packets and data packets. In other words, if the network size is growing without increase in active flows and data sending rate, then NCO will increase. The same behavior is exhibited in the experimental results presented in Figure 22(a). As the number of active source nodes and data sending rate were kept fixed, data generated in the all network sizes were the same. However, increase in network size causes increase in network control packets. Therefore, we can observe that NCO increases linearly with growing network size. In these experiments, the highest value of NCO was recorded as 16 (approximately) as shown in Figure 22(a) which means that with a network of size 180 IoT devices, the proposed management system generates around 16 control packets for every 100 data packets. In real world scenarios, increase in data rate is also expected with growing network size which will then compensate for increasing control overhead. To further validate this assertion, we have conducted another set of experiments with growing data sending rate, but this time network size was kept fixed, that is, 60 IoT devices. With the increase in data sending rate, more data packets are injected into the network with same number of control packets which is results in decrease in NCO as shown in Figure 22(b).

Furthermore, we have also performed experiments to analyze the impact of overhead injected by proposed management system on network performance measures, that is, end-to-end delay and throughput. For this purpose, we conducted simulations with and without management system while keeping all other parameters the same. We consider a network of 60 nodes divided into two areas, namely, area-1 and area-2. Nodes in each area are wirelessly connected to a local gateway node, and the gateway nodes are connected via routers using wired network. Management server is also connected to an intermediate router. Five nodes in area-1 are selected to send data to five other nodes in area-2. Nodes are randomly selected and activated for specific period of time (e.g., 5 sec). Another set of nodes are selected when activation time is over. In the first round of experiments, communication is initiated by management server, and all data are routed through the proposed management system. While in the second round, same experiments are repeated by establishing a direct (fixed) communication route between randomly selected source and destination nodes, thus by-passing the management system. With increasing data rate, gradual increase in end-to-end delay can be observed for both cases as shown in Figure 23(a) which is due increase in contention between active source nodes while access shared wireless channel. Comparatively more increase in end-to-end delay statistics can be observed for management system as shown in Figure 23(a). This is due to the fact that the data packets have to traverse a longer route including protocol stack at management server which induces slight increase in end-to-end delay. However, throughput results for both cases remain almost the same as shown in Figure 23(b). The difference between throughput results only become slightly visible when data sending rate was set to 120 packets/sec per source, that is, $120 \times 5 = 600$ packets/sec are generated in the network, and the size of each packets is 1000 bytes. In highly congested network conditions, the probabilities of packets collision get increased for experiment with management system as the scenarios has relatively more packets competing for channel access thus resulting in slight degrading in overall network throughput.

Finally, these simulation results reveal that the impact of communication overhead due to centralized management system is insignificant on network performance measures. However, in extreme network conditions, we may have to sacrifice fewer bits which is in fact worth nothing when compared to the flexibility and control offered by proposed management system over registered IoT devices.

6. Conclusions

This paper presents a management architecture for effective organization of services and virtual objects in

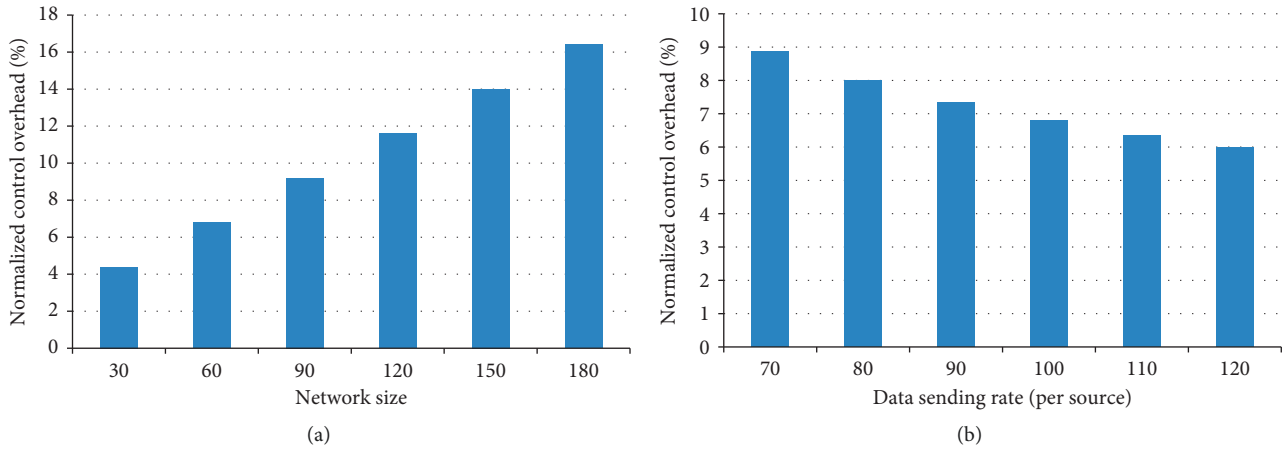

Figure 22: Simulation results with five active sources sending data @100 packets/sec. (a) Normalized control overhead with growing network size. (b) Normalized control overhead with increasing data rate.

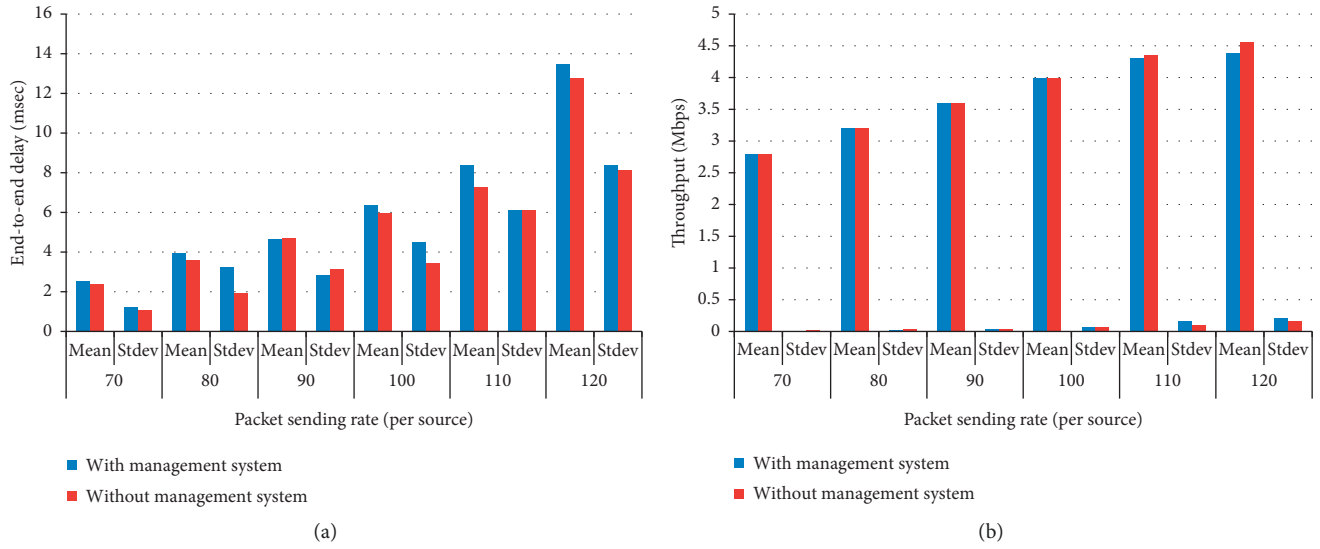

Figure 23: Simulation results with fixed network size of 60 nodes and five active sources. (a) End-to-end delay with increasing data rate. (b) Network throughput with increasing data rate.

hyperconnected IoT environment to facilitate management tasks, for example, locating devices in particular region, identifying faulty devices and services. For better visualization, services and devices are displayed over the map at their corresponding locations. The system also allows live interaction with device through their corresponding virtual objects to perform supported operations. We have performed the analysis of the proposed system performance with growing network size by implementing two application layer protocols in OMNeT++ simulator. Simulation results indicate that gateway nodes will become the performance bottleneck when the network size grows beyond certain extent. One way to improve the performance of hyper connected networks is by utilizing multiple gateway nodes as demonstrated through simulations results in this paper. Other solution includes improvement in wireless connectivity protocols for IoT devices. We have also performed resources requirement analysis for virtual objects along with control overhead of the proposed management system. Simulation results reveal that control overhead is insignificant in normal scenarios; however, in extreme network conditions, we may have to sacrifice fewer bits, which is in fact worth nothing when compared to the flexibility and control offered by the proposed management system over registered IoT devices. Furthermore, the current setup only supports those IoT devices with CoAP protocol. We are also working on the development of an interworking proxy to enable communication with IoT devices that do not support CoAP protocol. In future, we are also looking forward to integrate this system with an IoT-based application development tool and incorporate security module. This will enable common users to rapidly build their customized IoT-based applications by quick selection of desired devices and services. Furthermore, we will try to integrate IoTivity framework in our current system for sake of interoperability.

Data Availability

No data were used to support this study.

Conflicts of Interest

The authors declare no conflicts of interest.

Acknowledgments

This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (2014-1-00743) supervised by the IITP (Institute for Information & Communications Technology Promotion), and this work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (no. 2017-0-00756, Development of interoperability and management technology of IoT system with heterogeneous ID mechanism).

References

[1] L. Coetzee and J. Eksteen, "The internet of things—promise for the future? An introduction," in *IST-Africa Conference Proceedings, 2011*, pp. 1–9, Gaborone, Botswana, May 2011.

[2] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (coAP)," Internet Engineering Task Force, Fremont, CA, USA, 2014.

[3] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, "Internet of things: vision, applications and research challenges," *Ad Hoc Networks*, vol. 10, no. 7, pp. 1497–1516, 2012.

[4] P. P. Jayaraman, D. Palmer, A. Zaslavsky, and D. Georgakopoulos, "Do-it-yourself digital agriculture applications with semantically enhanced iot platform," in *Proceedings of the 2015 IEEE Tenth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, pp. 1–6, Singapore, April 2015.

[5] D. Mazzei, G. Fantoni, G. Montelisciani, and G. Baldi, "Internet of things for designing smart objects," in *Proceedings of the 2014 IEEE World Forum on Internet of Things (WF-IoT)*, pp. 293–297, Seoul, Korea, March 2014.

[6] Philipse Hue, 2016.

[7] S. De, F. Carrez, E. Reetz, R. Tonjes, and W. Wang, "Test-enabled architecture for IoT service creation and provisioning," in *The Future Internet Assembly*, pp. 233–245, Springer, Berlin, Germany, 2013.

[8] A. Bassi, M. Bauer, M. Fiedler et al., Enabling Things to Talk, Springer, Berlin, Germany, 2016.

[9] M. Naeem, R. Heckel, and F. Orejas, "Semi-automated service composition using visual contracts," in *Proceedings of the 7th International Conference on Frontiers of Information Technology*, ACM, Abbottabad, Pakistan, December 2009.

[10] S. Evren, H. James, and P. Bijan, "Semi-automatic composition of web services using semantic descriptions," in *Proceedings of the Web Services: Modelling, Architecture and Infrastructure Workshop in ICEIS 2003*, pp. 17–24, Angers, France, April 2002.

[11] G. Li, S. Deng, H. Xia, and C. Lin, "Automatic service composition based on process ontology," in *Proceedings of the Third International Conference on Next Generation Web Services Practices (NWeSP 2007)*, pp. 3–6, Seoul, Republic of Korea, October 2007.

[12] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau, "Automating DAML-S web services composition using SHOP2," in *Proceedings of the Second International Conference on Semantic Web Conference*, pp. 195–210, Springer-Verlag, Sanibel, FL, USA, October 2003.

[13] K.-D. Chang, C.-Y. Chang, H.-M. Liao, J.-L. Chen, and H.-C. Chao, "A framework for IoT objects management based on future internet IoT-IMS communication platform," in *Proceedings of the 2013 Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)*, pp. 558–562, Taichung, Taiwan, July 2013.

[14] Open Connectivity Foundation (OCF), *OIC Core Specifications*, 2017, https://openconnectivity.org/specs/OCF_Core_Specification_v1.3.0.pdf.

[15] Linux Foundation, *IoTivity–Open Source Project*, Linux Foundation, San Francisco, CA, USA, 2016, https://www.iotivity.org/.

[16] oneM2M, *Technical Specifications—Functional Architecture*, Version 2.10.0 Release 2, October 2016, http://www.etsi.org/deliver/etsi_ts/118100_118199/118101/02.10.00_60/ts_118101v021000p.pdf.

[17] ITU-T, *Requirements and Reference Architecture of the Machine-to-Machine Service Layer*, ITU-T, Geneva, Switzerland, 2015.

[18] Arduino, October 2016, https://www.arduino.cc/.

[19] Intel Edison Board, October 2016, https://software.intel.com/en-us/iot/hardware/edison.

[20] Raspberry Pi, October 2016, https://www.raspberrypi.org/.

[21] SAM: The Ultimate Internet Connected Electronics Kit, October 2016, https://www.kickstarter.com.

[22] M. S. Khan and D. Kim, "DIY interface for enhanced service customization of remote IoT devices: a CoAP based prototype," *International Journal of Distributed Sensor Networks*, vol. 2015, p. 8, 2015.

[23] A C Implementation of the CoAP Protocol, September 2016, https://github.com/smeshlink/CoAP.NET.

[24] A. Varga, "OMNeT++," in *Modeling and Tools for Network Simulation*, pp. 35–59, Springer, Berlin, Germany, 2010.

[25] A. Varga, *INET Framework for the OMNeT++ Discrete Event Simulator*, December 2012, https://inet.omnetpp.org/.