ORIGINAL RESEARCH PAPER

A Global Generic Architecture for the future Internet of Things

Wei Wang, Kevin Lee and David Murray

Abstract The envisioned 6A Connectivity of the future IoT aims to allow people and objects to be connected anytime, anyplace, with anything and anyone, using any path/network and any service. Because of heterogeneous resources, incompatible standards and communication patterns, the current IoT is constrained to specific devices, platforms, networks and domains. As the standards have been accepted worldwide, most existing IoT platforms use Web Services to integrate heterogeneous devices. Human-readable protocols of Web Services cause non-negligible overhead for object-to-object communication. Other issues, such as: lack of applications and modularized services, high cost of devices and software development also hinder the common use of the IoT. In this paper, a global generic architecture for the future IoT (GGIoT) is proposed to meet the envisioned 6A Connectivity of the future IoT. GGIoT is independent of particular devices, platforms, networks, domains and applications, and it minimizes transmission message size to fit devices with minimal capabilities, such as passive RFID tags. Thus, lower physical size and cost are possible, and network overhead can be reduced. The proposed GGIoT is evaluated via performance analysis and proof-of-concept case studies.

Keywords Internet of Things • SOA • Generic architecture • Middleware • Ontology

1 Introduction

The Internet of Things connects physical objects on the Internet. A thing is a virtualized object in information systems [1]. The virtual object has identities, attributes and communicates via its interfaces. The IoT involves people-to-people, people-to-object (P2O) and object-to-object (O2O) communication. Traditional Internet applications, such as blogs and online games, are facilitated by people-to-people communication. On the Internet, People can interact with the physical world via P2O communication, such as monitoring and tracking physical objects. In O2O communication, objects communicate with other objects, such as Near-Field Communications (NFC) [2].

The development of the IoT depends on innovation across many domains and industries, such as hardware manufacture and information technologies [3]. To collect real-time status of physical objects, barcodes wireless sensor networks and RFID technologies are necessary. Web 2.0 allows third-party users to provide and discover virtual objects and services on the Internet. To process device messages, Cloud Computing can provide elastic resources to process excessive real-time events [4]. With mobile networks and Wi-Fi, smart phones can act as proxies to connect objects to the Internet [2]. To exchange large amounts of data in networks, bandwidth is required to be increased [1].

The development of the IoT can be classified into four phases: i) Intranet of Things, ii) Extranet of Things, iii) Internet of Things, iv) Future Internet of Things and People [5]. Most existing RFID applications in closed systems are considered as Intranet of Things and Extranet of things, and little demand is needed for exchanging data across domains [5]. The third phase points to the current IoT designed for specific devices, platforms, networks, domains, services and applications. The fourth phase, Future Internet of Things and People, aims to connect people and objects on a global scale and across domains and industries [5]. In the future IoT, objects will become context-aware and make spontaneous decisions to communicate with people and machines [6].

The future IoT will be an integral paradigm across many domains, in which objects can seamlessly communicate with each other [7]. The future IoT needs an open architecture to realize interoperability among heterogeneous resources [1]. This paper presents a global generic architecture for the future IoT (GGIoT), which has many advantages. First, as the IoT is crossing domains, a generic architecture improves interoperability across systems. Second, as technologies are evolving and user requirements are changing, developing applications on a generic architecture can reduce time and cost to suit to the changes. Third, a generic IoT architecture allows third-parties to offer and consume services on public platforms in a simple way.

Building a generic IoT architecture is challenging, due to many issues. In this paper, a generic architecture is presented for the future IoT, and aims to resolve the following issues.

- *Device integration*: The IoT involves diverse devices, such as sensors, barcode and RFID tags, to collect and transmit description of physical objects. To integrate heterogeneous devices, Web Services enable standard web protocols to request data from IP-enabled devices. Web Services cannot be embedded on some resource-constrained devices. The future IoT needs to connect devices with minimal capabilities and physical sizes. In the proposed architecture, diverse devices can be seamlessly integrated at middleware in the distributed proxies. It is independent of devices and networks.

- *Semantic integration:* The future IoT needs to interpret meaning of received messages from different people. It is challenging to standardize description of objects by defining mandatory rules, similar to enforcing people to speak the same language worldwide. Using adaptors can interpret meaning of exchanged messages between systems. However, the interpretation of an adaptor is only limited to recognize patterns appear. This paper presents a novel approach of data integration based on building ontologies. Users are not required to describe data schema, object properties and units of measure in device messages. Meanwhile, the meaning of object and service description is also globally consistent.
- *Personalization:* Personalization delivers customized services to meet user needs [7]. As billions of objects are expected to connect to the future IoT, it is difficult to predesign all virtual objects and services. In this paper, the architecture enables third-parties to create and customize content on a global public platform.
- *Unexpected Interaction:* The future IoT will reach a global scale. When objects move between spaces, it is difficult to handle the uncertainties and the unexpected interactions among objects [8]. If an object discovers many unknown objects nearby, it should only interact with certain objects. Otherwise, numerous unnecessary interactions would wastes resources. The proposed IoT architecture uses distributed proxies to coordinate the local interaction among objects and services.
- *Real-time Capability:* In the future IoT, a great number of objects can generate a massive of real-time events. Object property values change constantly. To update real-time status of physical objects, events need to be immediately transmitted and processed. Most existing IoT platforms utilize Web Servers to exchange data between systems. Apart from the high overhead of web protocols, indirectly accessing sensor data in remote web servers also increases latency. In the proposed IoT architecture, to reduce network traffic for local O2O communication, device messages are exchanged and processed in distributed proxies via binary protocols.
- *Service Modularization*: Due to diversity of objects, services and devices, most existing IoT platforms do not provide reusable and modularized services, which make providing and consuming services difficult for third-parties, and also increase development costs. In the proposed architecture, a physical object or service is virtualized as a primitive middleware component. Cost and time can be reduced by reusing many atomic components to combine a composed service via the globally consistent interface.

This paper is structured as follows: Section 2 discusses architectural requirements for the future IoT. Section 3 analyzes constraints of existing IoT platforms. Section 4 presents a global generic architecture (GGIoT) for the future IoT. In Section 5, the architecture is evaluated via performance analysis and proof-of-concept case studies. Finally, Section 6 concludes this paper.

2 Architectural requirements for the future IoT

Many obstacles, such as incompatible standards, different communication patterns and lack of scalable frameworks, may hinder the envisioned future IoT. There is no single set of standards for the IoT currently. Some organizations are standardizing the IoT in different domains, such as EPC-global [9], ISO/IEC [10] and ZigBee [11]. These standards are evolving and incompatible with each other. It is impractical to ignore existing standards, and create new standards for the future IoT. The proposed IoT architecture uses existing Internet infrastructure, and it can integrate emerging resources independent of specific standards.

2.1 The 6A Connectivity of the future IoT

The envisioned 6A Connectivity of the future IoT allows people and objects to be connected anytime, anyplace, with anything and anyone, using any path/network and any service [3], which is the goal of the architecture in this paper. In terms of anytime, the future IoT is required to process received data on demand. IoT applications have different demands for latency, which may vary from a few seconds to a few days [5]. Previous work shows that the performance of Web Services is acceptable for most IoT applications currently [12]. For some time-sensitive IoT applications, such as self-driving vehicles, latency needs to be further decreased to guarantee QoS.

In the future IoT, objects and devices access the Internet via different networks and paths. With respect to anyplace, it is required to integrate diverse networks. To enable the anything connectivity, size and cost of sensors and RFID tags need to be further reduced. As a result, most common objects can connect to the future IoT. In regards to anyone, people may use different languages to describe entities in device messages. The future IoT needs to integrate device messages worldwide, and provide an easy manner to allow third-parties to offer and consume services.

2.2 Interoperability

In the IoT, interoperability is the capability of integrating heterogeneous devices, networks, systems, services, APIs and data representation across domains and systems [13]. It can be classified into network, syntactic and semantic levels [14]. The network interoperability focuses on device connectivity across networks, but does not concern the shared content. Implementing Web Services on sensors can encapsulate sensor message via web protocols [15]. The future IoT will connect many resource-constrained devices that do not support Web Services. It is required to use a globally consistent method to connect lightweight devices independent of specific networks, platforms and systems, and to integrate devices appears in the future [7].

When device messages are exchanged across systems, the incompatible data structures may hinder data parsing. Syntactic interoperation defines common data format and structure, such as HTML, for exchanged messages [14]. In the IoT, HTML adds much overhead for sensors [16], and Resource Description Framework (RDF) is heavy for resource-constrained devices as well [17]. Considering a trade-off between description level and the produced overhead, XML, JSON and CSV are accepted as the most suitable formats to describe objects in the IoT [5]. The difference is that these formats use different delimiters to set boundaries between data elements.

As entities are diversely described in device messages, semantic integration can be achieved by converting the entity description into system-readable representation via customized adaptors, and then interpret the meanings [18]. This method adds high design complexity to the adaptors, as an adaptor is limited for pre-defined data conversion.

Building ontologies can regularize rules to represent entities [14]. For example, Sensor Network Ontology describes types of sensors and sensor networks [19]. To classify object and service in the IoT, a service ontology use three sub-ontologies to describe the hosted services of sensors, locations, and physical properties [20]. However, as entity properties cannot be customized by third-parties, description capability of the ontology is limited and the ontology lacks scalability. In the IoT-A project, the class information model is used to classify fine-grained entities and build complex relations between the entities [21]. This model does not specify how entities are represented in sensor messages and how context is abstracted and interpreted in IoT applications.

2.3 The SOA principle

To allow third-parties to offer and consume services, the service-oriented architecture (SOA) is a design style that is independent of specific technologies and products [22]. SOA is not limited to the WS-* standards. Component-based models can also be SOA-enabled, such as the EJB specification. Compared to using Web Services, binary protocols are allowed in communication in component-based middleware, which permits a lower data rate in networks and lower overhead for sensor devices [23]. Traditional Internet applications are mostly designed for people-to-people communication. Human-readable web protocols facilitate third-parties to provide and consume services easily. Most IoT services, such as Smart Home and Smart Transport, are based on O2O communication. Using machine-readable binary protocols is beneficial for improving the overall performance.

2.4 Service modularization and Loose-coupling

In the IoT, loosely-coupled systems enable the logical separation of virtual objects and services. Each atomic object and service can be individually added, removed, and reconfigured. Thus, multiple objects and services can be combined to create new services. As primitive services are reused, cost and time in development can be reduced. Web Services provide relatively coarse-grained services, and component-based middleware enables more fine-grained services. A composed service should not consist of many constituent Web Services, as the accumulated latency is intolerant for some time-sensitive applications [23]. For component-based models, services can also be designed as coarse-grained, such as the Façade Pattern, which allows multiple primitive services to be composed into a coarse-grained service to meet application needs [24].

2.5 Multipoint communication

In the future IoT, an object needs to communicate with many objects at the same time. For example, a self-driving vehicle should simultaneously interact with the near cars, and traffic signals [25]. Multipoint communication can also be used to combine child-objects into a parent-object, and ensure that all the child-objects and parent-object can be individually addressed. For example, a machine consists of many parts. Multithreading is needed for object-to-service communication. One service needs to concurrently interact with multiple objects, or one object offers services to many entities. For example, an *Appliance Monitor* service allows a user to monitor all appliances at the same time [26]. In Smart Retail scenarios, many customers need to receive information from the same product simultaneously. Thus, multipoint communication is an architectural requirement when composing services in the future IoT.

2.6 Dynamicity and runtime reconfiguration

The dynamicity of the future IoT involves many aspects. Devices are added and removed dynamically in networks, due to many reasons, such as dispose of objects, shutting down devices, and moving devices between networks. The caused network topology is changed in real-time. It needs to dynamically allocate and release system resources for all connected objects and services [13], and create dynamic flows among objects, services and systems [27]. To add or remove connections between objects or services, existing connections with other objects and services should not be affected. To enable this, interaction among virtual objects and services should not be pre-configured and hard-coded, and connections among virtual objects and services needs to be reconfigured at runtime[13]. The WS-* standards cannot achieve it, as the dependency resolution mechanism are hard-coded in the Web Services. For component-based models, the components need to be fine-grained to enable runtime reconfiguration; the coupling between components needs to be managed from outside mechanisms [28].

2.7 Controlled interaction and Decentralization

When objects are moving between spaces, it is difficult to handle the uncertainties of interactive objects [8]. If an object discovers many unknown objects nearby, it should select certain types of objects for communication. Building a full connection between all objects in a network would cause many meaningless connections and waste resources.

In GGIoT, O2O communication is coordinated by a local proxy. By analyzing relations among objects and services in ontologies, the proxy can coordinate interaction among these objects to avoid unexpected interaction.

Most existing IoT platforms use remote web servers to request sensor data via HTTP, and to provide URIs to access sensor data in the RESTful style. The centralized IoT architecture has security, privacy, trust, responsibility and data ownership issues [1]. When data is transmitted through many networks, it is inefficient to use centralized web servers to retrieve sensor data. The future IoT should support distributed data accessing, processing, storage and ownership. Users can decide which parts of data can be shared in public or specific groups [13].

2.8 Simplified deployment

Simplified deployment can improve the common use of the IoT. To reduce hardware cost, lightweight sensors and RFID tags can be used in parts of IoT applications. To reduce development cost, third-parties can reuse existing services to compose services. A plug-and-play mode can connect diverse devices transparently and seamlessly[29], and no-programming is required to deploy devices and services. The future IoT should use consistent methods to model sensor data [30]. Moreover, service composition and service optimization need be automated in context-aware environments in the future IoT.

3 Existing IoT platforms

Most existing IoT platforms are designed for particular devices, platforms, networks, domains and applications. These IoT platforms use RESTful APIs to access sensor devices, and to retrieve, store, update and delete data via the standard HTTP operations such as Get, Post, Put and Delete. The exchanged data formats are XML, JSON or CSV. Sensor data is stored in cloud-based databases for processing and accessing. IoT platforms, such as Xively [31], Axeda [32], ARMmbed [33], Arrayent [34] Carriots [35], Bugswarm [36], DIGI Device Cloud [37], Evrythng [38], Thingspeak [39], Nimbits [40], and GroveStreams [41] all follow this style. They have differences in some non-core functions, such as business model, data storage policy, data management, visualization, data analysis, event notification and access permission control.

The existing IoT platforms share many constraints that may hinder the deployment of the future IoT. To send data to the platforms, sensors must support Web Services, which is not suitable for resource-constrained devices. Another method is to use proxies to post sensor data to the RESTful API via manual programming. This method adds complexity for non-expert users. Most commercial platforms collaborate with their hardware manufacturers, and have specific requirements on devices and networks. For example, a device used in the AXEDA platform need to support mobile network, and the ARMmbed platform uses 6LoWPAN networks. The Xively platform requires users to write key pairs into firmware of the designated devices. The envisioned future IoT should be device and network-independent. The diversities and constraints may hinder the ubiquity of devices and networks in the IoT.

The average cost of a single sensor device is too high if connecting many ordinary objects for common use. Some IoT platforms, such as Kaa [42], offer open source codes to third-parties for building applications. However, cost of the required devices is still a barrier for connecting ordinary objects to the IoT; the size of the sensors may also be too big to attach them on small objects. To reduce device size and development cost, barcode and RFID tags can replace sensors in parts of applications. For example, the Evrythng IoT platform enables connection of barcode and RFID tags by posting static object descriptions via the RESTful APIs.

These IoT platforms do not provide ontologies, which hinder the interpretation of device messages from different parties. Third-parties describe entities in device messages. This constraint increases the message size and consumes more energy of devices. Due to diverse representation, it is also difficult to integrate sensor data at the syntactic and semantic level. Some IoT platforms, such as Arrayent and Kaa allow the users to create and share user-defined data models. The Arrayent platform enables describing device properties using the shared vocabularies, but it is limited to specified device properties. Other features, such context-awareness, service composition, service optimization, and automation also cannot be realized. Therefore, the existing IoT platforms are far from meeting the envisioned 6A Connectivity of the future IoT.

As the existing IoT platforms do not offer off-the-shelf applications, users need to build applications from scratch, or to write code by using the provided tools and manuals. It requires programming skills for non-expert users, reduces service reusability, and also increases cost. The future IoT should provide simplicity to non-experts users. Some IoT platforms provide customized services to their clients, such as system integration, software development and hardware design. It would further increase the cost of using the IoT, because of the lack of standardization and interoperability across heterogeneous platforms.

4 Building a global generic architecture for the IoT

4.1 Design principle

GGIoT aims to integrate heterogeneous resources into the IoT, and to meet the 6A Connectivity of the future IoT. As some devices cannot run node-level middleware, device messages are integrated in gateway-level middleware to shield hardware details of the devices. The local proxy uses component-based middleware to receive, exchange and process device messages. O2O communication is enabled by binary protocols, as it can reduce message size to fit devices with minimal capabilities, and also reduce network overhead [43]. Each sensor message consists of a system-allocated object ID, and collected dynamic object property values. Object properties, static property values, measure of units, and data models are described in object templates in ontologies. Thus, message size can be reduced to a minimum to fit energy-constrained devices. In ontologies, third-parties are able to customize description of entities, and share the description worldwide. The ontologies can be easily expanded, and description of entities is also globally consistent. Each entity is described in a template. By mapping a sensor message with the related template in the ontologies, the message meaning can be interpreted.

In middleware, an object component is a virtualized object, which is allowed to interface with multiple service components via the consistent APIs. Many atomic object and service components can be combined to provide a composed service. The reuse of primitive services can improve the common use of the IoT, reduce development cost, and facilitate easy-deployment. The middleware tier provides consistent APIs of modularized services to IoT applications. Between the middleware and applications, communication patterns can be designed as pull, push or publish-subscribe, to fulfill different application needs without concerning the underlying devices and networks. This paper focuses on the virtualization and integration of heterogeneous resources at the middleware tier.

4.2 Object description

In the IoT, object-attached devices have diverse physical size, memory size, energy consumption, processing and sensing ability. It is necessary to use a globally consistent method for describing entities in device messages, and enable the method to fit devices with minimal capabilities. Passive RFID tags do not have the ability to sense and to store too complex data. Static properties of the connected objects can be described in detail in the backend systems, and an object identifier is described in device messages to link the description in the backend systems. Barcode tags also do not have sensing ability, and similar methods can be used to index object description in systems.

In GGIoT, dynamic property values of objects, such as temperature and location, are collected by sensors, active RFID tags and GPS devices, to represent real-time states of the labeled objects. A sensor message consists of two data fields: object identifier and dynamic property values. Other elements, such as message schema, static property and the values, and unit of measure, are considered as the description overhead, as they are constantly static in object description. By moving the description overhead to the back-end system, the size of sensor messages can be significantly reduced to fit lightweight devices. Stripping the overhead from sensor messages can also maintain the description consistency and shield complexity from users.

Figure 1 illustrates an example of the stripping process. A sensor message only contains a system-allocated object ID and the collected dynamic property value “*16*”. Other data elements are filled into in a generic milk template by a third-party user, which results in creating a customized object template. The customized template is stored in the ontologies for interpreting messages from the attached object. The gateway-level middleware abstracts each data elements of received messages. By matching them to the registered object template in the ontologies, meaning of received messages can be interpreted. The update of the ontologies is synchronized worldwide. Meanings of entity descriptions are globally consistent. Virtual objects and services across systems can be composed in middleware without using data conversion.

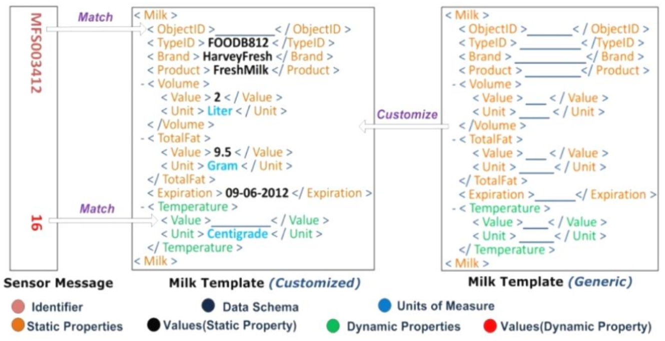


Fig. 1 Stripping the description overhead from a sensor message

Some devices do not have data encryption ability; the messages can be intercepted maliciously between devices and gateways. In GGIoT, the message description method can provide a potential solution to alleviate the "last mile privacy issue". As a sensor message does not contain any description about ownership, property, unit of measure and data schema, the message needs the associated template to interpret it. The device message is only system-readable. For barcodes or passive RFID tags do not have sensing ability, a device message or an image only contains an object ID used to index a static object template in systems.

4.3 The overall architecture of GGIoT

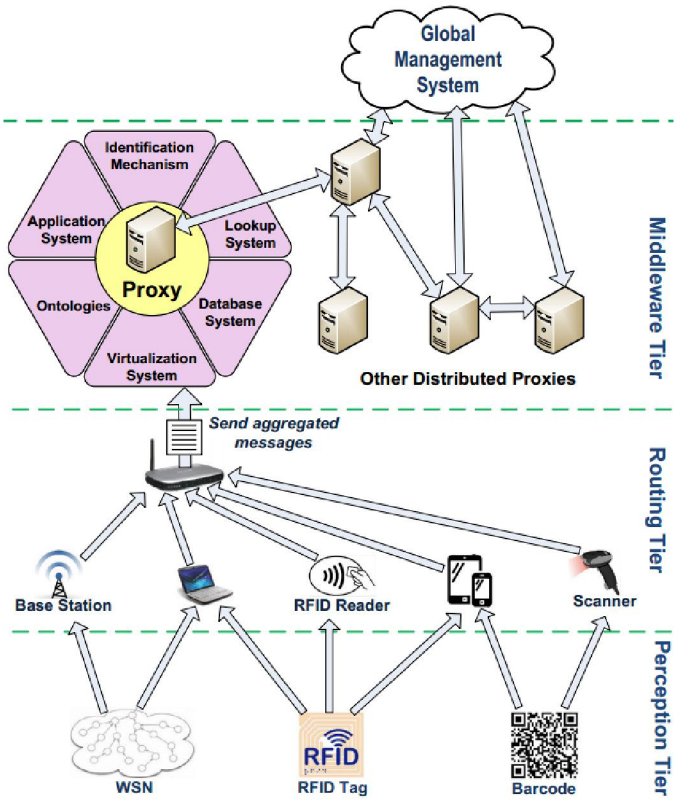


Fig. 2 The overall framework of GGIoT

GGIoT is component-based, proxy-integrated and binary-protocol-enabled. It is independent of underlying device, network, system and service. The consistent description of entities enables GGIoT to integrate resources across domains and parties. Figure 2 demonstrates the overall framework which includes: perception tier, routing tier, middleware tier and global management system (GMS).

The perception tier collects raw data from the physical world using different devices, and to represent the data as dynamic property values of virtual objects in middleware. The routing tier builds communication channels between various devices and the middleware in the proxies. A variety of intermediate devices, such as mobile phones, tablets, and laptops can be used to relay messages from end-point devices to nearby gateways. The gateways then route aggregated messages to the middleware tier via the Internet. As mobile networks cover most places of the world, mobile devices can route messages in areas other networks are unavailable. Considering security or privacy issues, the public routing devices should be managed by trusted parties, and relevant legislations need to be issued.

At the middleware tier, a distributed proxy consists of ontologies, identification mechanism, lookup, database, virtualization and application system. The virtualization system virtualizes a device message of a physical object into an object component, and modularizes a service as a service component in middleware. Many primitive object and service components can be combined as a composed service to reduce development cost. Each component runs temporarily in middleware. If a component is removed, the unused resource is released.

The identification mechanism assigns a global unique identifier (GUID) for a virtual object. In device messages, an object ID is pre-allocated by Identifier Manager before initializing the object component in middleware. The assigned object component ID is identical to the object ID in device messages. The aim of this design is to discover the virtual objects using system-allocated component IDs. The IDs are temporarily allocated to object and service components, and can be recycled and reused. In GGIoT, the lookup system provides a discovery mechanism to enable the discovery of virtual objects and services, or classification of objects and services in the ontologies.

The application system can offer various development tools for third-parties. For example, template editing tools can be plugged into browsers, and to allow third-party users to describe objects and services in templates rather than in device messages. Third-parties can also use their own words to find suitable existing templates to match connected objects and services. Objects and services can be customized by modifying existing generic templates. Thus, third-party developers focus on application design without concern for hardware details of sensory devices. GGIoT also allows using multiple off-the-shelf services to compose a service on demand.

The object and service ontology describe the relation between virtual objects and services. The unit ontology describes units of measure. Moreover, location, device, time and other types of entities can also be described in the ontologies. The ontologies can be updated to adapt to new emerging entities. To ensure global consistency, all ontology data is managed by the GMS. Other distributed proxies periodically download updates. The GMS allocates a range of identifiers to all distributed proxies. Then each distributed proxy further assigns the allocated identifiers to the local virtual objects and services.

As most O2O communication occurs within specific areas, using a distributed proxy to handle communication of near objects and services can reduce network traffic and access latency. The distributed proxies can run on a local network, metropolitan area network, private and public cloud. Location and specification of a proxy is determined by the hosted objects and services, required resources and application needs. All distributed proxies form a global network. If O2O communication is beyond the range of a proxy, the GMS can coordinate the communication of the involved virtual objects and services in different proxies. The GSM can be used for discovering objects and services across proxies. Images of the GSM can be backed up and synchronized in different locations worldwide. The design of the GSM is beyond the scope of this paper.

4.4 Object Virtualization

GGIoT enables a globally consistent description of entities before virtualizing objects into middleware. Third-parties, such as end-users and manufactures, use their own words to look for off-the-shelf templates in ontologies to map connected objects and services. The user-defined keywords are analyzed in search engines, and then the most suited templates are discovered to represent static status of the connected objects. GGIoT provides flexibility to allow third-parties to use their own language to represent entities in the IoT, and all participants need to commit the rules for data representation in device messages. In middleware, an object component relays messages from the object-attached device to the wired service components. By mapping the messages into the associated object templates, meaning of the object state can be interpreted. Figure 3 illustrates the object virtualization process in GGIoT.

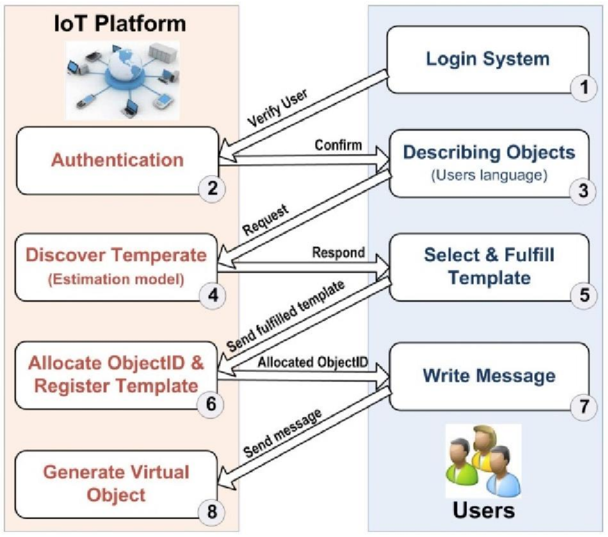


Fig. 3 The process of object virtualization in GGIoT

In Step 1, third-parties login to a user management system. It allows users to store user profiles and manage virtual objects via web browsers. Step 2 is used to verify user identities by authentication methods, such as security devices and password. In step 3, third-parties discover an existing template to describe a connected object. Users can specify a template ID to access an object template, or use their own language as keywords to find the most suitable template. It is difficult to standardize languages to describe entities worldwide. GGIoT offers flexibility of allowing third-party users to represent entities using their own language in the IoT.

In step 4, the ontologies analyze the search keywords, and find some suitable template candidates to match the connected object. The discovered object templates can be translated into different languages depending on users' preferences. Then URIs of the translated templates are sent to the users. In step 5, when the users receive the system-recommended object templates, they can select the most suitable template to describe the new connected object. The URI of the selected template is returned to the systems for registration. If the users cannot find a ready-made object template, they can fill blanks of a generic object template. The filled object template is registered in ontologies, and a template ID is assigned for the template.

In step 6 and step 7, a GUID is allocated to the object. The object ID is formatted into device messages by the users, and is identical to the component ID of the virtual object in middleware. Thus, the object can be discovered via the system-allocated object ID. A mapping between the object ID and the related template ID is registered in a distributed proxy. As a result, it is unnecessary to present the template ID in the device messages. It is beneficial to remove the description overhead from the messages, and enhance privacy strength of the transmitted data. In a device message, another data field *Values of Dynamic Properties* is collected from the physical object in real-time. For example, in a sensor message "*MFS003412 16*", "*MFS003412*" is the system-allocated object ID and "*16*" is the dynamic property value of *Temperature*, which is collected by a sensor. As a template ID "*FOODB812*" is pre-associated with the object ID in a proxy, the template can be used to interpret meaning of the sensor message.

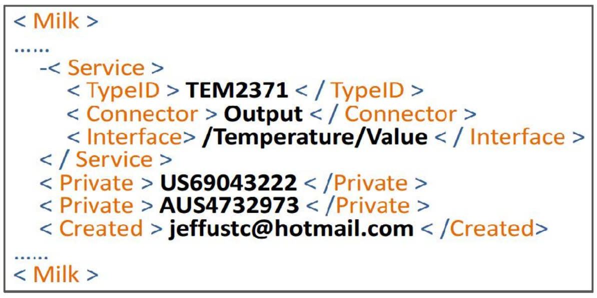
In step 8, the middleware receives the sensor messages, and map them into the related object template to interpret the message meaning. The middleware can also generate an object component to relay the device message to other service components in real-time.

4.5 Building ontologies

In GGIoT, ontologies describe entities and the relations between entities. The object ontology, service ontology and unit ontology are the three basic ontologies that describe the relations among objects, services and units of measure. Other ontologies, such as the location, time, and device ontology, can also be used to describe other types of entities. Each template describes one type of entities and the relations with other entity types. All templates form the ontologies in GGIoT. The templates are published, customized, discovered and accessed on public platforms, to achieve the consistency worldwide. The ontologies are formed in a hierarchical structure, and can be extended by adding new templates to describe new type of entities.

An object template describes static states of one type of objects. An object component outputs the received device messages to update dynamic property values of the object instance. A clear separation of object type and instance can offer a loosely-coupled communication pattern in GGIoT. The ontologies verify types of objects and services when a proxy coordinates O2O communication among the objects. An object is limited to communicate with pre-defined types of objects and services to avoid unexpected interaction. All object templates constitute the object ontology. Reusing existing templates can simplify deployment, save time and cost, and enable consistent entity description worldwide. Third-parties can also customize objects by editing existing object templates. By adding new object properties into an existing template, a new object template is generated. One type of objects is constrained to consume limited types of services, which can also be defined in the object template.

Figure 4 illustrates an object template of a bottle of milk. If adding a <*Service*> field into the object template, the object component is entitled to interface with a service of *Temperature Monitor*. "*TM2371*" is the URI of the service template in the service ontology. <*Connector*> declares the interface type, and <*Interface*> locates the interface. In the ontologies, shared templates are public templates, and private templates can only be accessed by authorized users. By adding the <*Private*> field into the object template, the associated object component can only be accessed by two communities "*US69043222*" and "*AUS4732973*". The <*Created*> field indicates the creator of the customized template. The creator can also grant different access rights, such as reading, deleting and modifying, to other users.



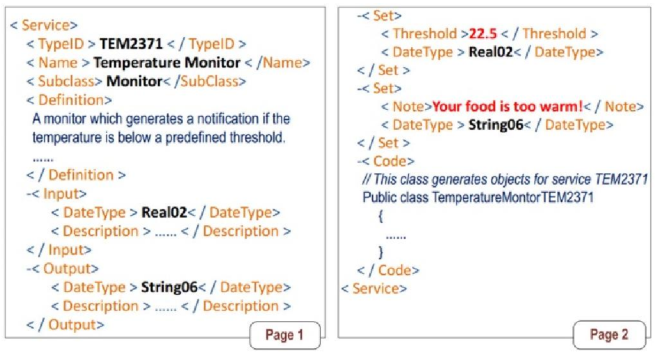
```
< Milk >
……
   -< Service >
       < TypeID > TEM2371 < / TypeID >
       < Connector > Output < / Connector >
       < Interface> /Temperature/Value < / Interface >
   < / Service >
   < Private > US69043222 < /Private >
   < Private > AUS4732973 < /Private >
   < Created > jeffustc@hotmail.com < /Created>
……
< Milk >
```

Fig. 4 Adding service and ownership into object template

A physical object consists of a set of physical properties, and uses different units to measure the values. In GGIoT, the unit ontology classifies the units of measure; describes the relations between units; maintains semantic consistency of units. In process of service composition, a unit can be converted into other units to adapt to interface of the wired components. For example, one meter can be converted to 10 decimeters, or 100 centimeters. In a unit template, a field <*UnitID*> contains an identifier used to access the unit template; a field <*Subclass*> indicates classification of the unit; <*definition*> defines the semantic meaning; and

<*Conversion*> describes the relations with other units.

In middleware, a service component offers a primitive service. It receives messages from the wired components, and then output the processed data. A service template describes one type of service, and is the primitive unit to form the service ontology. A service template contains a class used to generate service components. To customize new types of services, existing service templates can be edited by third-parties, which results in registering new templates in the service ontology. Figure 5 illustrates a service template for a *Temperature Monitor* service. The service can send a notification message to subscribers if the temperature value of an object falls below a threshold.



```
< Service>
  < TypeID > TEM2371 < / TypeID >
  < Name > Temperature Monitor < /Name>
  < Subclass> Monitor< /SubClass>
  < Definition>
   A monitor which generates a notification if the
   temperature is below a predefined threshold.
   ......
  < / Definition >
  -< Input>
      < DateType > Real02< / DateType>
      < Description > ...... < / Description >
  < / Input>
  -< Output>
      < DateType > String06< / DateType>
      < Description > ...... < / Description >
  < / Output>
                                          Page 1

  -< Set>
      < Threshold >22.5 < / Threshold >
      < DateType > Real02< / DateType>
  < / Set >
  -< Set>
      < Note>Your food is too warm!< / Note>
      < DateType > String06< / DateType>
  < / Set >
  -< Code>
   // This class generates objects for service TEM2371
   Public class TemperatureMontorTEM2371
      {
        ......
      }
  < / Code>
< Service>
                                          Page 2
```

Fig. 5 The service template of Temperature Monitor

The data fields <*TypeID*>, <*Name*> and <*Definition*> can be used for identification and discovery. <*Subclass*> indicates the service classification. The fields <*Input*> and <*Output*> declare receptacle and interface of the service component. For a virtual service, the receptacle receives messages and the interface output processed data. The field <*DateType*> is referred to data-type ontology that describes data types and the relations among them. As a result, output data of a service component can be converted into a compatible data type to connect another service. For example, integer data can be converted into real data to adapt to the input of a service component. The temperature threshold and notification message are adjustable variables. In the data field <*Set*>, third-parties are allowed to edit the threshold values, which resulting in generating a customized service template and a new ID is allocated for the new template. The field <*Code*> can provide the class source code used to generate instances of service components.

Due to the global scale of the future IoT, if an object moves to a new network, the object would be unware of the existence of objects nearby. The object needs to select particular types of objects for interaction. In GGIoT, the limitations are defined in the object template. The relation among virtual objects and services is many to many. For instance, a refrigerator can use a Voltage Monitor service to monitor its voltage, and concurrently subscribes the *Temperature Monitor* service. One service can also be used by multiple virtual objects. For example, a *Speed Monitor* service can be used to measure speed of multiple vehicles at the same time.

The device ontology describes properties of object-connected devices. By adding a <*device*> field into an object template, the description of the connected device is linked to the device template. Other ontologies, such as time and location, can be created to describe other domain-specific entities. All the ontologies use a template as the primitive unit, and can be customized to generate new templates to meet application needs. The IoT is evolving; it is difficult to pre-design all ontologies once and for all. In GGIoT, the ontologies are expanded to adapt to emerging entities without losing the relations with existing entities.

4.6 Virtualization System

The virtualization system is used to virtualize physical objects and services into components in middleware, and to combine primitive object and service components to provide composed services. In GGIoT, the loose-coupling feature enables efficient service composition via globally consistent interfaces of virtual objects and services. Figure 6 illustrates the virtualization system.

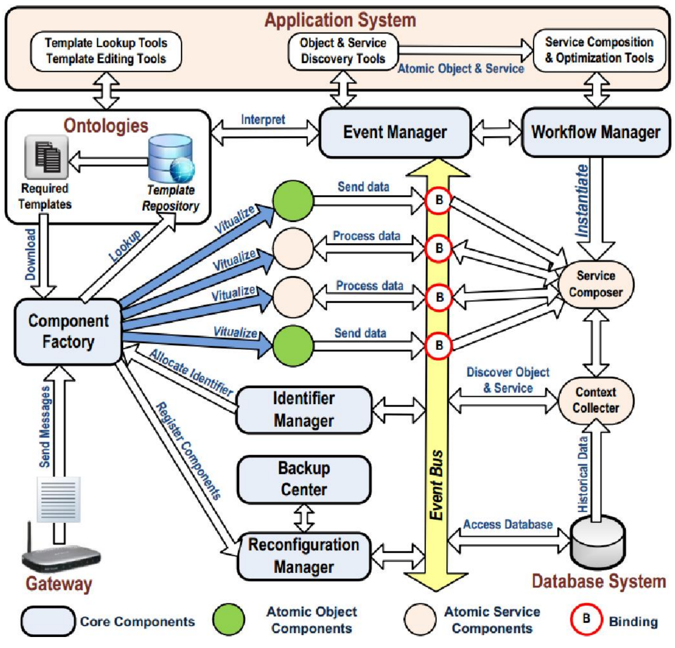


Fig. 6 The virtualization system in GGIoT

In a proxy, the gathered messages consist of multiple lines of text. Each message is received from an object-connected device. A message consists of two data fields: i) an object ID allocated by the Identifier Manager, and ii) collected data to represent dynamic object property values. The Component Factory can relay the device messages to the associated object components. By mapping the device messages to the object templates in ontologies, message meaning can be interpreted. As the object ID in a message is pre-allocated by the Identifier Manager, the proxy can judge if the virtual object exists. If a virtual object does not exist, the Component Factory initializes a new middleware component for receiving messages from the object, and the Identifier Manager allocates a GUID for the new generated component. The component ID is identical to the object ID in the device message.

If a message is from an already-virtualized object, the message is routed to the related object component. Other components wired to the object component can currently receive the message. The message is used to update the dynamic property values of the object in real-time. As interfaces and receptacles of distributed components are globally consistent, virtual objects and services can be seamlessly integrated in middleware, and heterogeneity of the underlying devices is hidden. As barcode tags and RFID tags do not have sensing ability, the labeled objects are virtualized as static virtual objects that do not have incoming messages.

Each object component has an expiration period. If the virtualization system has not received message from an object above a predefined period, the object component is altered to inactive state. If the object reconnects to a proxy, the component can be reactivated. The temporary disconnections can be caused by many reasons, such as turning off sensors, moving objects between networks, or rebooting a router.

If the object component fails to receive message from an object, it may be caused by low battery, abandoning objects, hardware error of devices and other issues. The object component is removed from middleware to release the unused system resource. The assigned component ID and object ID are recycled for reusing as well. Users can set a period threshold for the temporary and permanent disconnection. The Backup Centre is used to provide a recovery mechanism to recover the removed component, and to rewire previous bindings with other components.

To provide virtual services, the Component Factory can generate service components on demand. A service component has an interface to output data, and more than one receptacle to receive inputted data. The interfaces and receptacles are predefined in the service template in the ontologies. Many object and service components can be combined to offer a composed service. The composition workflow can be coordinated by a component of service composer, which is discussed in Section 4.7.

4.7 Service Coordination

A service composer is a component generated by the Component Factory. It contains rules and algorithms used to control workflow in service coordination. To fulfill different application needs, third-parties customize the workflow and rules in template of the service composer. A service composer is entitled to couple and uncouple a connection between two components from the outside. The decision-making depends on collected context and pre-defined rules in a workflow. In service composition, the required dynamic context is collected from the related object and service components, and the static context can be looked up in the object and service templates in the ontologies. A service composer needs to be aware of states of all involved components. To update component states, events are sent to the Event Manager when a component is activated, suspended or removed.

A composed service may suffer context change in the dynamic environments. To adapt to change, components are dynamically wired or unwired to other components by following pre-defined rules. With runtime reconfiguration, components and the wiring between components can be reconfigured without rebooting systems. Reconfiguration Manager can register, inspect and reconfigure components on runtime. By invoking control commands, components can be activated, suspended, removed, wired or unwired to other components. All components are loosely-coupled; unwring two components does not affect the coupling with other components. The loose-coupling feature can enable multipoint communication of virtual objects and services.

A GUID is assigned for an object or service independent of location and networks. An object ID is static during the lifetime of the object. If an object is moved from one place to another where uses another proxy, the object component in the previous proxy is terminated, and the setting of the object component is stored in the Backup Centre. Then the new proxy downloads the setting from the Backup Centre, and generates a new object component with the previous setting. The wiring with other components in the previous proxy can be rewired in the new proxy. The aim of the decentralized design is to reduce network traffic and access latency when objects move between proxies.

5 Evaluation

This section presents a proof of concept evaluation of the virtual system in GGIoT. As diverse devices are integrated at the gateway-level middleware, the use of devices is not a consideration in the test. As device messages are routed to a proxy from different networks and paths, GGIoT is also independent of networks and communication channels. In middleware, as virtual objects and services use compatible interfaces and receptacles to exchange messages, the entity representation in device messages is globally consistent. Thus, GGIoT has generality to integrate heterogeneous and emerging resources. The features of the architecture are not confined to devices, systems, setting, performance analysis, services and case studies in this evaluation.

5.1 Implementation setup

Figure 7 shows the implementation setup. Device messages was formatted into the Arduino UNO [44], and transmitted via the Zigbee Xbee [11]. A dual-core PC acts a distributed proxy. A ZigBee Xbee Explorer is used as a sink node to aggregate messages from all objects in a WSN, and bridge the WSN with the proxy. The proxy receives the messages from the sink node via a USB port. GGIoT is independent of operating systems; both Ubuntu 10.10 and Windows XP were tested as underlying systems.

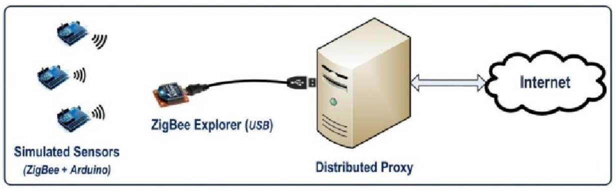


Fig. 7 The setup of the implementation

The proxy runs the virtualization system based on the LooCI/OSGI V1.0 [45]. LooCI middleware was taken off-the-shelf to support evaluation of the virtualization system [46], for three reasons. First, LooCI components run on Java Virtual Machine rather than on physical platforms, the middleware is independent of processors, platforms and systems. Therefore, it can meet the device-independency need. Second, as LooCI supports runtime reconfiguration in the LooCI network, components and bindings between the components can be reconfigured without a recompilation or restart, which can fulfill the loose-coupling need. Compared to middleware, such as OpenCOM [47], LooCI supports multithreading among components, which can enable multipoint communication in the proposed architecture GGIoT.

5.2 Virtualization process

In this test, Ubuntu 10.10 was installed on the proxy. A sensor message has three fields: *ObjectID*, *Temperature value*, and *Sending time*. This test assumes that the data fields of *Temperature value* and *Sending time* are the two dynamic property values of an object. Figure 8 illustrates the logical model and virtualization process workflow. A component of Sensor Relay was designed to interpret gathered sensor messages from the sink node. *ObjectID* are abstracted from each message by the Sensor Relay, and are sent to the Identifier Manager. As an *ObjectID* is pre-registered in the Identifier Manager, the registration can be verified by the proxy. If a device message is from a registered object, the message will be further processed. Otherwise, the message is discarded.

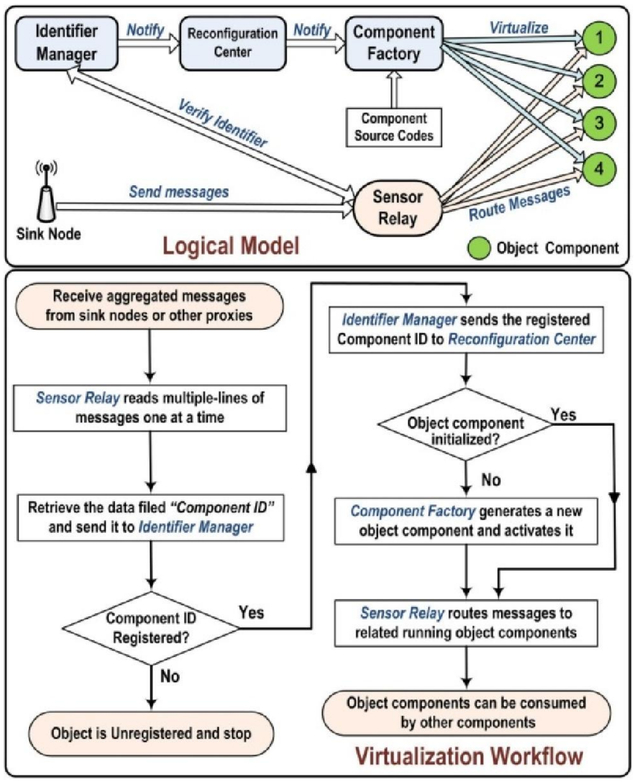

Identifier Manager
Notify
Reconfiguration Center
Notify
Component Factory
Virtualize
1
2
3
4
Verify Identifier
Component Source Codes
Send messages
Sensor Relay
Route Messages
Sink Node
Logical Model
Object Component
Receive aggregated messages from sink nodes or other proxies
Sensor Relay reads multiple-lines of messages one at a time
Retrieve the data filed "Component ID" and send it to Identifier Manager
Component ID Registered?
Yes
No
Object is Unregistered and stop
Identifier Manager sends the registered Component ID to Reconfiguration Center
Object component initialized?
Yes
No
Component Factory generates a new object component and activates it
Sensor Relay routes messages to related running object components
Object components can be consumed by other components
Virtualization Workflow

Fig. 8 The logical model and workflow of object virtualization

Reconfiguration Center can reconfigure components and the binding between components on runtime. It also stores the states of each component, such as initialized, activated or wiring. These states can be requested by the commands, such as GetState and GetInterface [48]. In this evaluation, if state of a target object component is activated, related messages are routed to the object component. Otherwise, a new object component is initialized to receive messages from the object-attached device.

A component of Component Factory was designed to generate object components in middleware. Source codes, used to generate object components, were pre-stored in a Java file. The Component Factory can fill some data fields, such as object identifier, of the source code with captured data from messages and proxies. The filled source code is compiled to generate byte code, such as Class and Jar files. By executing the byte code in JVM, object components can be deployed in heterogeneous platforms. In GGIoT, object components can be transferred from one proxy to another. As byte code of object components is portable, it can be backed up in the GSM, and then moved to other proxies.

In this test, by parsing aggregated messages from the sink node, the virtualization system dynamically generated object components for objects in a WSN. The components were published on the event bus, and can be subscribed by other components in the middleware. Each component was individually accessed by the component ID and IP address of the proxy. As these object components use compatible interfaces to output the dynamic object property values, middleware components in other platforms are able to wire to them without data conversion of adaptors.

5.3 Memory footprint and overhead testing

Compared to executing middleware or Web Services on sensors, GGIoT runs all middleware components in the proxies. Thus, devices with minimal capabilities can be integrated. The previous test shows that the visualization system used 196 KB to initialize the first object component [49]. Upon activating additional components, footprint of each component decreased from the second component (82 KB) and then stabilized at the sixth component (28 KB). The disparity of component size can be inferred by creating components from the same source code; many components shared the same process in memory.

The memory size is acceptable for a distributed proxy in GGIoT. For example, a regular PC with 8 GB of RAM can offer memory for running at least 200,000 components if the operation system uses 2 GB of memory. In GGIoT, a proxy could run in a router, regular PC, private cloud or public cloud. Required hardware specification of proxies depends on many factors, such as number of the connected objects and services at peak times, budget, QoS, user and application needs.

In GGIoT, many primitive objects and services can be combined to provide a composed service without protocol conversion. The overall overhead of a composed service is accumulated by the communication overhead between all the primitive components, which depends on applications.

The test measured the communication overhead of two atomic components. In the proxy, two components were deployed in the proxy. Round trip time (RTT) latencies between the two components were measured, and socket communication was tested for benchmarking. A Java method *System.nanoTime* returned the local system time. In GGIoT, each sensor message only contains an object ID and collected dynamic property values. Considering the message size is typically less than 100 bytes, this test used 100 bytes of data in the message. Two components A and B were wired to test the RTT latency. Component A initially sent a message to component B, and then the message is returned to A. By comparing the sending time and receiving time, the RTT latency was tested.

In GGIoT, it is unnecessary to invoke remote services in other proxies in most applications. Service components can be initialized in a proxy on demand. Comparing to the remote procedure call (RPC) of Web Services, local invocation can significantly reduce the latency when data goes through many networks on the Internet. In this test, the latencies between two atomic components based on local and remote invocation were both tested. The tests are repeated 100 times, and the average values are used for comparison. Figure 9 illustrates the latency test results based on the local invocation, socket and RPC.

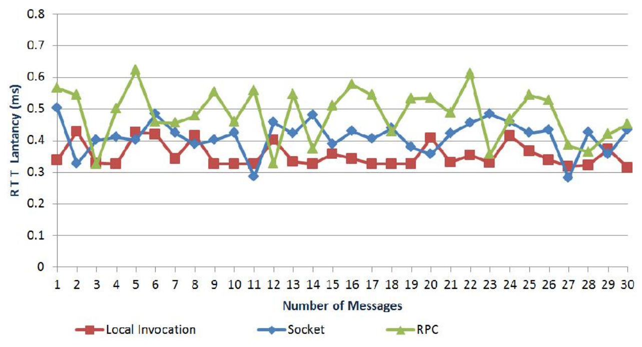


Fig. 9 The latency between two primitive components

The results show the average RTT latency based on local invocation is 0.356ms; the average latency on the socket communication is 0.412ms. The added 0.056ms can be inferred that the socket API operations, such as reading and writing, can be blocked by the underlying operation systems for context switch [50]. The average RTT latency based on RPC communication is 0.483ms. Compared to the socket communication, the added 0.071 ms of latency can be explained by the need to firstly relay the data to the Macro-component [45]. The results show that the local invocation has the smallest variation and the RPC has the largest variation in execution time.

In GGIoT, as O2O communication is coordinated and processed by distributed proxies rather than centralized web servers. The test results indicate a significantly low overhead between the two primitive components, which enables a composed service to consist of many primitive components on a proxy. For RPC communication, as the middleware uses the UDP protocol, the added overhead is less than 0.1ms. The purpose of this test is to measure the minimized overhead between the two components in the specific proxy. In practice, the performance of a service will depend on the many factors, such as the underlying systems, devices, networks and applications.

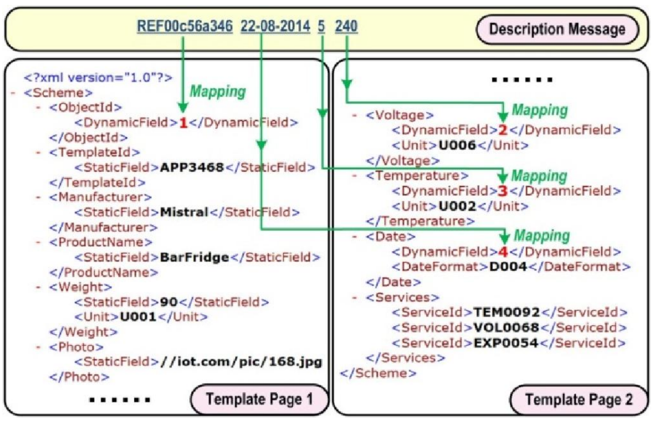
5.4 Case studies

GGIoT aims to provide a generic IoT architecture across domains. Due to the global scale, implementing a concrete architecture needs a great deal of cooperation and efforts from many parties. In this section, two case studies were designed to demonstrate basic features of monitoring and tracking services, and O2O communication in GGIoT. The illustrated principles have generality to adapt to services in different domains. XML was used as the data format for the involved object and service templates in the ontologies.

5.4.1 Monitoring and tracking services

This section utilizes a service of *Temperature monitor* to illustrate monitoring services. An object component was deployed to receive and relay the dynamic property values of a simulated refrigerator. By wiring to the refrigerator component, a *Temperature Monitor* component can receive and monitor the temperature values of the refrigerator. The *Temperature Monitor* sends a notification to subscribers if the temperature value is above a predefined threshold. The threshold value can be customized for reusing the service. In this test, the threshold was set as "5". If the monitored temperature value is above 5 ℃, the notification message is "Your refrigerator is too hot". Otherwise, the notification is "Your refrigerator is normal".

Figure 10 shows the object template used to describe the refrigerator. The refrigerator template has six data fields to describe static property values of the refrigerator, such as <*TemplateId*> and <*Weight*>. The template also has three dynamic properties including <*Temperature*>, <*Voltage*> and <*Date*>. The fields of the dynamic properties are left blank for mapping values from the sensor messages. The values of <*Unit*> are linked to the Unit Ontology, and the <*DateFormat*> fields describe various date formats. It is unnecessary to parse all data fields of the template, which depend on application needs. In this service, it is required to interpret the values of <*ObjectId*> and <*Temperature*>.



```
<?xml version="1.0"?>
- <Scheme>
  - <ObjectId>
      <DynamicField>1</DynamicField>
    </ObjectId>
  - <TemplateId>
      <StaticField>APP3468</StaticField>
    </TemplateId>
  - <Manufacturer>
      <StaticField>Mistral</StaticField>
    </Manufacturer>
  - <ProductName>
      <StaticField>BarFridge</StaticField>
    </ProductName>
  - <Weight>
      <StaticField>90</StaticField>
      <Unit>U001</Unit>
    </Weight>
  - <Photo>
      <StaticField>//iot.com/pic/168.jpg
    </Photo>
    ......
```

```
......
  - <Voltage>
      <DynamicField>2</DynamicField>
      <Unit>U006</Unit>
    </Voltage>
  - <Temperature>
      <DynamicField>3</DynamicField>
      <Unit>U002</Unit>
    </Temperature>
  - <Date>
      <DynamicField>4</DynamicField>
      <DateFormat>D004</DateFormat>
    </Date>
  - <Services>
      <ServiceId>TEM0092</ServiceId>
      <ServiceId>VOL0068</ServiceId>
      <ServiceId>EXP0054</ServiceId>
    </Services>
</Scheme>
```

Fig. 10 The object template of the refrigerator

The object component is restricted to interact with three types of services. Template IDs of the supported service types were pre-defined in the <*Services*> field. Figure 11 demonstrate a workflow of the *Temperature Monitor*. The refrigerator component relays the device messages to the service components of *Temperature Monitor* in real-time. By mapping the received messages with the associated template, the *Temperature Monitor* interprets meaning of each data field in the messages. A device message does not contain a template ID. The *Temperature Monitor* looks up the template ID of the refrigerator in the Identifier Manager before it parses the dynamic property values.

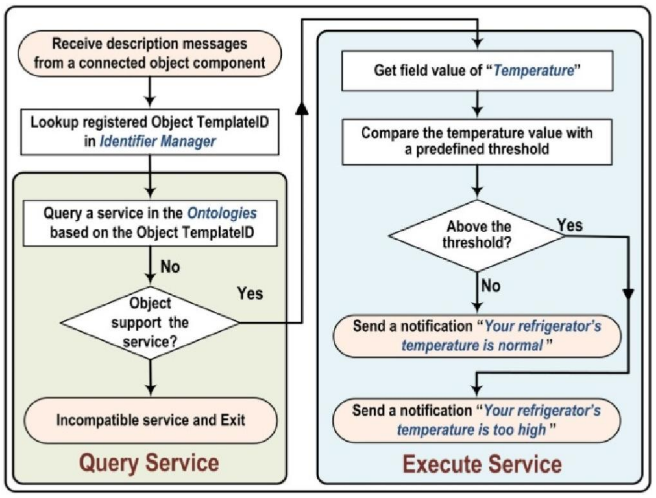

Receive description messages from a connected object component
Lookup registered Object TemplateID in Identifier Manager
Query a service in the Ontologies based on the Object TemplateID
No
Object support the service?
Yes
Incompatible service and Exit
Query Service
Get field value of "Temperature"
Compare the temperature value with a predefined threshold
Above the threshold?
Yes
No
Send a notification "Your refrigerator's temperature is normal"
Send a notification "Your refrigerator's temperature is too high"
Execute Service

Fig. 11 The workflow of the Temperature Monitor

The module of *Query Service* parses the ontologies, and verifies if the service of *Temperature Monitor* is compatible with the object component. If the verification is passed, the module of *Execute Services* retrieves the current temperature value from the latest sensor message. By comparing the retrieved temperature values with the predefined threshold "*5*", temperature of the refrigerator can be monitored in real-time. The test results show the notification messages were displayed as expected. In this case, the triggered action is sending notifications. Other actions can also be designed to meet different application needs. In GGIoT, current location of objects can also be parameterized as a dynamic property value. As a result, the data processing of tracking services is similar to the monitoring services.

5.4.2 Object-to-object (O2O) communication

Most IoT services are designed for O2O communication. Compared to the monitoring and tracking services, some differences exist. Monitoring services can be applied to a single object; O2O communication needs participation of at least two objects. Therefore, a third-party component is required to coordinate O2O communication. Moreover, most monitoring and tracking services are provided to specified objects, while objects in O2O communication are unpredictable. Objects may move between networks, which causes interaction with new objects nearby.

This section presents a service of *Expiration Manager* to demonstrate O2O communication in GGIoT. In this test, two object components were deployed to receive messages from two sensors that describe a refrigerator and a bottle of milk respectively. The refrigerator uses the same object template in Figure 10. In the middleware, outputs messages of the refrigerator component contain a dynamic property value of <*Date*> to describe local time of the refrigerator. Another template was used to describe a bottle of milk. The milk template has some static object properties, such as <*Manufacturer*> and <*Volume*>, and a dynamic object property <*Expiration*> describes the expiration date of the milk. Although expiration date of a product is static after the product is produced, the same types of milk may have different property values of <*Expiration*>. It is inefficient to duplicate many templates to describe the same product in mass production. This case assumes that <*Expiration*> is a dynamic property of the milk, and formatted into messages of the attached RFID tag.

A component of *Expiration Manager* was designed and multithreaded to the two object components to receive the messages. By comparing property values of <*Date*> of the refrigerator to property values of <*Expiration*> of the milk, the service of *Expiration Manager* can decide if the milk is expired, and send notifications to the subscribers. If the expiration date of the milk is after the local date of the refrigerator, a message "Your milk has expired" is notified. Otherwise, the notification message is "Your milk is still fresh". Figure 12 illustrates the workflow.

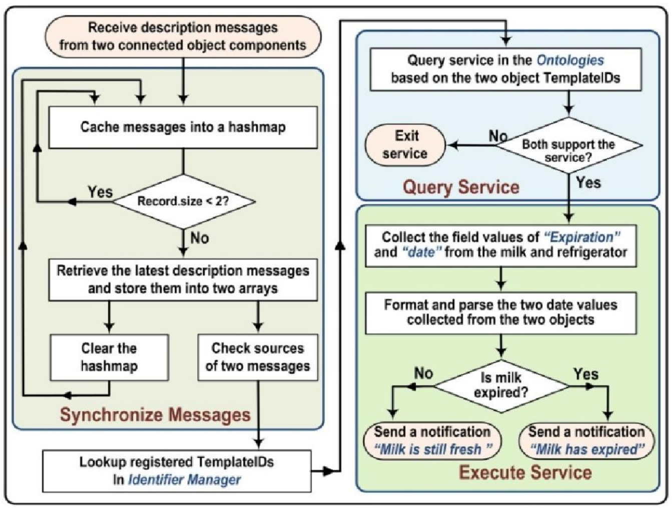

Receive description messages from two connected object components
Cache messages into a hashmap
Yes
Record.size < 2?
No
Retrieve the latest description messages and store them into two arrays
Clear the hashmap
Check sources of two messages
Synchronize Messages
Lookup registered TemplateIDs In Identifier Manager
Query service in the Ontologies based on the two object TemplateIDs
Exit service
No
Both support the service?
Yes
Query Service
Collect the field values of "Expiration" and "date" from the milk and refrigerator
Format and parse the two date values collected from the two objects
No
Is milk expired?
Yes
Send a notification "Milk is still fresh"
Send a notification "Milk has expired"
Execute Service

Fig. 12 The workflow of the Expiration Manager

In GGIoT, as third-parties may set different intervals to send device messages, it needs to synchronize received messages of different objects before executing services. In this case, the *Expiration Manager* adds a new module of *Synchronize Messages* to retrieve the latest messages from the two objects. A hashmap instance was created to cache device messages of the two object components. When the Expiration Manager receives a message from the milk or the refrigerator, each data field of the message is separated and cached into an array. The value of object ID is used to index the message in the hash map. When two messages with two object IDs are cached, the two messages provide the latest dynamic property values of the two objects.

The two messages are retrieved from the hash map, and then the hash map is cleared to cache new coming messages from the two object components. The sources of the two retrieved messages can be judged by object IDs. Then Template IDs of the two objects are looked up in the Identifier Manger. The module of *Query Service* verifies if both the two objects both support the service of *Expiration Manager* by parsing the ontologies. Then the *Execute Service* module can compare dynamic propriety values of <*Expiration*> and <*date*> of the two objects, and decide if the milk is expired.

This method caches the latest sensor messages of the two objects to synchronize dynamic property values of the two objects. Compared to indirectly accessing stored messages in web servers, the messages of the two objects are synchronized in RAM of the distributed proxy. A few milliseconds of latency are enough to meet requirements of most IoT applications. If O2O communication crosses many networks on the Internet, two approaches can be applied to guarantee the QoS. For sensors with accurate timing capability, timestamps are added into the device messages for comparison. Otherwise, timestamps can be appended to device messages at a local proxy when the proxies receive the device messages. As a result, service components can verify if received messages are time-effective for specific services.

6 Conclusions and future works

The 6A Connectivity of the future IoT has been proposed for a decade [3]. The IoT is still constrained to particular devices, platforms, networks, applications and domains, and many barriers may hinder development. In this paper, the proposed IoT architecture provides potential solutions to overcome these barriers. Due to limitations of device cost, object size and energy consumption, it is difficult to embed Web Services into all devices in the future IoT. GGIoT is independent of device, platform, network and system, and aims to meet the 6A Connectivity in term of *anytime*, *anyplace*, *anything* and *any network/path*.

As the modularized object and service components are loosely-coupled and the interfaces are globally consistent, many object and service components can be combined to provide a composed service. Services can be customized, shared and discovered by third-parties on a global public platform. Cost and time in development can be reduced. These features aims to fulfill the *any service* requirement. Third-parties are allowed to look for an existing template to describe an object or service. The off-the-shelf feature allows non-expert users to provide and consume services without programming. Other features, such as simplified deployment, personalization and device independency enables GGIoT to meet the *anyone* requirement.

In GGIoT, a sensor message consists of an object ID and dynamic property values collected from the object. Other descriptions, such as data schema, unit of measure, static property and static property values, are stripped from sensor messages. This method can minimize message size to fit devices with minimal capabilities. In GGIoT, objects are restricted to interact with predefined types of objects and services. On a distributed proxy, a service composer verifies the compatibility of all participating objects and services before coordinating O2O communication. Thus, the unexpected interaction among multiple objects can be controlled when objects are moving between spaces. Most O2O communication occurs within specific networks or areas. To reduce network traffic and access latency, GGIoT uses distributed proxies and binary protocols to coordinate the local O2O communication without Internet traffic.

Building a global IoT architecture needs a lot of efforts across many domains. The development of the future IoT will be determined by many factors, such as government policies, academia and markets. This paper illustrates the initial design concept of the proposed architecture GGIoT, and mostly focuses on device and data integration in the virtualization system. A lot of mentioned concepts, such as the GMS and backup mechanism, can be extended in the future work. This paper emphasizes on the middleware tier of GGIoT. If applying the architecture in IoT applications, the design of application tier needs to be completed. Future work will address service composition and optimization, and discovery mechanisms in GGIoT. It is the authors hope that the proposed architecture in this paper will contribute to evolution of the future IoT.

References

1. Vermesan O, Friess P, Guillemin P, Gusmeroli S, Sundmaeker H, Bassi A, Jubert IS, Mazura M, Harrison M, and Eisenhauer M, Internet of things strategic research roadmap. Internet of Things-Global Technological and Societal Trends, 2011: p. 9-52.
2. Al-Ofeishat HA and Al Rababah MA, Near Field Communication (NFC). International Journal of Computer Science & Network Security, 2012. 12(2).
3. ITU, ITU Internet Reports - The Internet of Things, 2005.
4. Lee K, Murray D, Hughes D, and Joosen W. Extending sensor networks into the Cloud using Amazon Web Services. in IEEE International Conference on Networked Embedded Systems for Enterprise Applications (NESEA). 2010.
5. Uckelmann D and Harrison M, Architecting the Internet of Things2011, Heidelberg, Germany: Springer. 347.
6. Evans D, The internet of everything: How more relevant and valuable connections will change the world. Cisco IBSG, 2012: p. 1-9.
7. Gyumyang L and Crespi N, Shaping future service environments with the cloud and internet of things: networking challenges and service evolution, in Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation - Volume Part I2010, Springer-Verlag: Heraklion, Crete, Greece. p. 399-410.
8. Huadong, Internet of things: Objectives and scientific challenges. Journal of Computer Science and Technology, 2011. 26(6): p. 919-924.
9. EPCglobal. Standards Development. 2014; Available from: http://www.gs1.org/gsmp/kc.
10. ISO/IEC. list of ISO/IEC JTC 1/SC 31 standards. 2014.
11. ZigBeeAlliance. The ZigBee Alliance creates IoT standards that help Control Your World2015; Available from:

http://www.zigbee.org/zigbeealliance/.

12. Trifa V, Wieland, S., Guinard, D., Bohnert, T. M. Design and implementation of a gateway for web-based interaction and management of embedded devices. in 2nd International Workshop on Sensor Network Engineering (IWSNE 09). 2009. CA, USA.
13. Saint-Exupery A, Internet of things, strategic research roadmap, 2009.
14. Terziyan V, Kaykova O, and Zhovtobryukh D. Ubiroad: Semantic middleware for context-aware smart road environments. in Internet and Web Applications and Services (ICIW), 2010 Fifth International Conference on. 2010. IEEE.
15. Moritz G, Zeeb E, Golatowski F, Timmermann D, and Stoll R. Web services to improve interoperability of home healthcare devices. in Pervasive Computing Technologies for Healthcare, 2009. PervasiveHealth 2009. 3rd International Conference on. 2009. IEEE.
16. Guinard D, Trifa V, Mattern F, and Wilde E, From the internet of things to the web of things: Resource-oriented architecture and best practices, in Architecting the Internet of Things2011, Springer. p. 97-129.
17. Larizgoitia I, Muguira L, and Vazquez JI, Architecture for WSN nodes integration in context aware systems using semantic messages, in Ad Hoc Networks2010, Springer. p. 731-746.
18. Paridel K, Bainomugisha E, Vanrompay Y, Berbers Y, and De Meuter W, Middleware for the internet of things, design goals and challenges. Electronic Communications of the EASST, 2010. 28.
19. W3C, Semantic Sensor Network XG Final Report, 2011.
20. Kim J-H, Kwon H, Kim D-H, Kwak H-Y, and Lee S-J. Building a service-oriented ontology for wireless sensor networks. in Computer and Information Science, 2008. ICIS 08. Seventh IEEE/ACIS International Conference on. 2008. IEEE.
21. Walewski JW, Initial architectural reference model for IoT. EC FP7 IoT-A (257521) D, 2011. 1: p. 2.
22. Spiess P, Karnouskos S, Guinard D, Savio D, Baecker O, Souza L, and Trifa V. SOA-based integration of the internet of things in enterprise services. in Web Services, 2009. ICWS 2009. IEEE International Conference on. 2009. IEEE.
23. Petritsch H Service-Oriented Architecture (SOA) vs. Component Based Architecture. 2005.
24. Milanovic N. Service engineering design patterns. in Service-Oriented System Engineering, 2006. SOSE'06. Second IEEE International Workshop. 2006. IEEE.
25. Bao F, Chen I-R, and Guo J. Scalable, adaptive and survivable trust management for community of interest based Internet of Things systems. in Autonomous Decentralized Systems (ISADS), 2013 IEEE Eleventh International Symposium on. 2013. IEEE.
26. Petriu EM, Georganas ND, Petriu DC, Makrakis D, and Groza VZ, Sensor-based information appliances. Instrumentation & Measurement Magazine, IEEE, 2000. 3(4): p. 31-35.
27. Valente B and Martins F. A middleware framework for the Internet of Things. in AFIN 2011, The Third International Conference on Advances in Future Internet. 2011.
28. Nain G, Fouquet F, Morin B, Barais O, and Jézéquel J-M. Integrating iot and ios with a component-based approach. in Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on. 2010. IEEE.
29. Bandyopadhyay S, Sengupta M, Maiti S, and Dutta S, A survey of middleware for internet of things, in Recent Trends in Wireless and Mobile Networks2011, Springer. p. 288-296.
30. Tracey D and Sreenan C. A Holistic Architecture for the Internet of Things, Sensing Services and Big Data. in Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on. 2013. IEEE.
31. Xively. Xively – Public Cloud for the Internet of Things. 2014; Available from: https://xively.com/.
32. Axeda. Axeda Tutorials. 2015 [cited 2015 4th, July]; Available from: http://developer.axeda.com/learn/by-type/tutorial.
33. AMRmbed. Let's Connect Everything. 2015 [cited 2015 4th, July]; Available from: http://mbed.com/.
34. Arrayent. The Arrayent Connect Platform. 2015 [cited 2015 4th, July]; Available from: http://www.arrayent.com/.
35. Carriots. Create amazing products and services with our Internet of Things Platform. 2015 [cited 2015 4th, July]; Available from: https://www.carriots.com/.
36. Bugswarm. Bugswarm documentation. 2015 [cited 2015 4th, July]; Available from: http://developer.bugswarm.net/.
37. DIGI. Digi Device Cloud. 2015 [cited 2015 4th, July]; Available from: http://www.digi.com/cloud/digi-device-cloud#docs.
38. Evrythng. How it works. 2015 [cited 2015 4th, July]; Available from: https://www.evrythng.com/technology/.
39. Thingspeak. Getting Started. 2015 [cited 2015 4th, July]; Available from: https://thingspeak.com/docs.
40. Nimbits. Nimbits Manual. 2015 [cited 2015 4th, July]; Available from: https://docs.google.com/document/d/1aOEpfeJOtV-v0diDBAQ9e2hOjXhs0uInA2QE--OS8Pk/view.
41. GroveStreams. GroveStreams Development. 2015 [cited 2015 4th, July]; Available from: https://grovestreams.com/developers/developers.html.
42. KAA. The truly open-source Kaa IoT Platform. 2016 [cited 2016 28, April]; Available from: http://www.kaaproject.org/.
43. Juric MB, Rozman I, Brumen B, Colnaric M, and Hericko M, Comparison of performance of Web services, WS-Security, RMI, and RMI–SSL. Journal of Systems and Software, 2006. 79(5): p. 689-700.
44. Arduino. Arduino Uno Overview. 2014; Available from: http://arduino.cc/en/Main/arduinoBoardUno.
45. Hughes D, Thoelen K, Horré W, Matthys N, Cid JD, Michiels S, Huygens C, and Joosen W. LooCI: a loosely-coupled component infrastructure for networked embedded systems. in Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia. 2009. ACM.
46. LooCI. LooCI OSGi implementation. 2014; Available from: https://code.google.com/p/looci/wiki/OSGiImpl.
47. Coulson G, Blair G, Grace P, Taiani F, Joolia A, Lee K, Ueyama J, and Sivaharan T, A generic component model for building systems software. ACM Transactions on Computer Systems (TOCS), 2008. 26(1): p. 1.
48. LooCI. LooCI: Command Overview. 2014; Available from: http://code.google.com/p/looci/wiki/Commands.
49. Wang W, Lee K, and Murray D. Integrating sensors with the cloud using dynamic proxies. in Personal Indoor and Mobile Radio Communications (PIMRC), 2012 IEEE 23rd International Symposium on. 2012. IEEE.
50. Hruby T, Crivat T, Bos H, S. aA, Tanenbaum, and Amsterdam VU, On Sockets and System Calls: Minimizing Context Switches for the Socket API, in Conference on Timely Results in Operating Systems (TRIOS 14)2014, USENIX Association: Broomfield, CO.