RESTful Platform Broker: A Novel Framework for Engineering Programmable Smart Spaces

Govind Raj Pothengil
Centre For Development of Advanced Computing
Noida, UP, India
pgovindraj@cdac.in

Subrat Kar, Hari Mohan Gupta
Department of Electrical Engineering,
Indian Institute of Technology,
New Delhi, India
subrat@ee.iitd.ac.in, hmgupta@ee.iitd.ac.in

***Abstract*—Designing and developing a Smart Space is a difficult engineering and research problem. This paper presents the design and implementation of a novel framework for engineering Programmable Smart Spaces called RESTful Platform Broker. The RESTful Platform Broker can be considered as a middleware on top of a platform (e.g., Android). It views the network of the Smart Space as a system bus and exposes resources (e.g., Mobile Phone cameras, Smart Phone sensors, etc.) that are present in the platform as RESTful resources. Once the resources are exposed, these can be accessed by other devices present in the Smart Space. This facilitates programming the Smart Space and developing new services from the existing ones. Further, RESTful Platform Broker uses HTTP as a glue to interconnect heterogeneous components within the Smart Space. This enables any programming language, or any tool, which can understand HTTP to manipulate resources and program the Smart Space. This is an advantage as compared to earlier approaches, which warranted the use of specific languages to program a Smart Space or use protocols like 9P/9P2000, which are not as popular as HTTP. Further, the RESTful Platform Broker uses Hypermedia as an engine for application state, which allows various components within the Smart Space to evolve independently with respect to each other, thereby, increasing Smart Space extendability. Therefore, the RESTful Platform broker eases the engineering of a Smart Space by hiding the complexity of the physical infrastructure by providing a standard interface which can be used by Smart Space programmers to extend and program it.**

Keywords-Smart Space; Pervasive Computing; REST.

I. INTRODUCTION

Pervasive computing offers some unique research and engineering challenges [1]. One of the identified research areas in Pervasive computing is a Smart Space. A Smart Space is a physical space consisting of heterogeneous computing devices and services together with a software infrastructure. The users can interact with these devices in an unobtrusive manner through the infrastructure. This software infrastructure abstracts the boundaries of the individual devices, present in the physical space. It allows the user of a Smart Space to interact with the set of devices as a whole, rather than individual devices. In other words, services, applications and peripherals in a particular device can be accessed or used by other devices easily. Most of the first generation Smart Spaces were custom built to implement a particular scenario (e.g., Smart House, Smart Hospital, Smart Meeting Room, etc.). It was difficult to incorporate new technologies and extend the Smart Space by adding new devices and services. This was mainly because these Smart Spaces lacked the ability of being programmed. The Gator Tech Smart House [2] was among the first Smart Spaces that introduced the concept of Programmable Pervasive Spaces. Programmable pervasive spaces are Smart Spaces which exist both as a runtime environment and a software library. Smart Space Programmers can, therefore, extend and evolve the Smart Space by incorporating new devices and services.

However, engineering Smart Spaces is a challenging and difficult task. The basic research challenge in engineering Smart Spaces is in the system level architecture and component level synthesis [1]. This research work aims at designing and developing a framework using which the engineering of a programmable Smart Space can be simplified.

Most of the tools that are available to engineer Smart Space provide a middleware. These middlewares provide APIs so that programmers can use it to incorporate new services and devices into the Smart Space. Though Middleware APIs provide a way to extend the Smart Space, the requirement that all devices must have API compatibility, could be a serious hindrance in extending it. Further, many projects like GAIA [4] also mandated the use of specific programming languages to extend the functionality of a Smart Space. Plan B [11] provided a different approach. It was designed as an OS which exported resources within a device as a filesystem. Any language can therefore be used to program the Smart Space. Though, this alleviated the problems with APIs, however the limitation of Plan B was that it could not be ported easily to different platforms as it was an OS. In order to overcome these limitations, a new framework for engineering programmable Smart Spaces called the RESTful Platform Broker has been designed and developed. It is designed as a middleware, so that it is easier to port to different platforms. Further, it provides a Service Oriented filesystem based mechanism to program a Smart Space so that it is not limited to a particular set of API's or specific languages.

This paper presents the design and implementation of

 ISBN: 978-1-61208-225-7

RESTful Platform Broker. It helps engineers and researchers to develop programmable Smart Spaces. The main objective of the framework is to hide the complexities of the underlying infrastructural elements of a Smart Space and provide a uniform view of the resources present in it in such a way that it is easy to program and extend the Smart Space. The RESTful Platform Broker is considered as a framework because Smart Space applications can be built using it. From an implementation point of view, the RESTful platform broker is a middleware that provides an abstraction between a Smart Space programmer and the individual infrastructure elements present in the Smart Space. It views the network of the Smart Space as a system bus and exports resources present within a device/platform onto the Smart Space network as service oriented file systems. The concept of service oriented file system was proposed by Eric Van Hensberg [3]. Essentially, the RESTful Platform Broker queries the platform for resources. These resources can be in the form of hardware, e.g., Camera, or a software, such as a Text to Speech utility. Once a resource is found, it wraps the resource as a service oriented file system and exposes it as a RESTful resource, which can be accessed using HTTP. The user, however, can specify the resources they want to expose to the Smart Space and thus, control the visibility of the resources within it. The exposed resources can then be used by other devices running on other platforms within the Smart Space by issuing HTTP commands like GET, POST, DELETE and PUT. Using this framework, programmers can extend and program service compositions using any language. Further, any device that can understand HTTP can access the resources exposed by RESTful Platform Broker.

The RESTful Platform Broker has been implemented over the Android Platform and has been installed on commercially available Samsung Galaxy Pop GT-S5570 mobile phone. Interaction of RESTful Platform Broker with GNU/Linux (Ubuntu 11.04) has been achieved. The experimental setup for this has been discussed in this paper. Though the implementation of the RESTful Platform Broker is done on the Android Platform, it can run on any platform on which there is a suitable JVM. The design of the RESTful Platform Broker is such that porting to a different platform can be done with minimal changes to its codebase.

This paper is divided into 6 Sections. Section 1 introduces and presents a background of the research work. Section 2 presents the work which is related to the current research. Section 3 presents the Design and implementation of the RESTful Platform Broker. This section also presents a DNS based Service Discovery mechanism implemented as a part of this project. Section 4 presents usage scenarios of the RESTful Platform Broker. Section 5 presents results on RESTful Platform Brokers performance. Section 6 concludes the paper.

II. Related Work

The main aim for our research work is to design and develop a pervasive computing framework which will help engineers and researchers to develop programmable Smart Spaces. This section focuses on projects, using which one can design and develop pervasive computing application. This section also compares their approach with that of ours.

Gaia [4] is a Meta operating system for Smart rooms. The goal of this project was to design and develop a middleware operating system (Meta operating system), that manages the resources in a Smart Space. Conceptually it is similar to a traditional operating system which manages tasks that are common to all applications programs. Gaia extends typical operating system concepts to include context awareness, provide location awareness, detect when new devices are spontaneously added to the Smart Space and adapt when data formats are not compatible with output devices. From an architecture point of view, Gaia was implemented as a CORBA middleware. System components were implemented as distributed objects with CORBA IDL interfaces. To some extent, Gaia supports Smart Space programmability. Using Gaia, programmers can integrate devices and create new services in a Smart Space. However, Gaia [4] requires special programming language like Olympus [5] to program the Smart Space and utilize specialized mechanisms like Microservers [6] to induct a device onto a Smart Space. In contrast, in our approach any language that can handle HTTP can be used to program the Smart Space. Further, any device that supports HTTP can become a part of the Smart Space without implementing any specialized software like Microservers.

Stanford Universitys initiative, iROS [7][8] is an open source software platform for designing and developing Smart Spaces. Like Gaia, iROS can also be considered as a meta operating system for Smart Spaces. The programming model of iROS is to ensemble independent entities in the Smart Space through the use of its subsystems. iROS consists of three subsystems: EventHeap for application coordination, DataHeap for data movement and transformation and ICrafter for user control of resources. A Smart Space designed and developed using iROS as the platform needs to wrap its resources using ICrafter to function within a Smart Space. When using RETSful Platform Broker, devices need to interact through a more generic HTTP based Domain Application Protocol rather than specific mechanisms like ICrafter. This enables integration of more devices into the Smart Space, as greater numbers of devices are using the HTTP protocol [9].

EQUATOR Component Toolkit (ECT) [10] provides a mechanism for constructing pervasive computing applications by integrating sensing devices (inputs from sensors, such as *phidgets*, *motes* and *d.tools* boards) and actuation devices (of physical actuators including X10 modules, output

 ISBN: 978-1-61208-225-7

of Internet Applications, etc.). The project uses a visual graph based editor to allow run-time interconnection of modules.

Plan B OS can be considered as a software infrastructure for Smart Spaces [11]. Plan B is an operating system based on Plan 9 [12][13] which exports the resources (both hardware and software of a device) as synthetic file systems. These resources can then be shared with other devices in the Smart Space using 9P protocol and can be used through file system operation. Our research prototype shares a common design principle with Plan B, which is to expose resources as file systems. However, our approach relies on HTTP, which is a very common protocol whereas Plan B relies on 9P/9P2000 for sharing resources. Implementing 9P protocol even on open source operating system like GNU/Linux is challenging [14]. Moreover PlanB has been implemented on desktop systems, whereas, the RESTful Platform Broker has been implemented on top of Android, which is a mobile platform. Such a platform is more apt for developing Smart Spaces as mobility of devices is one of the key characteristics of a Smart Space. Further, PlanB is an operating system and hence it is difficult to port as compared to RESTful Platform Broker which is implemented as a middleware.

Nokia Research has designed and developed a RESTful framework [15] for Smart Spaces. It supports resource discovery, authorizing access to resources with group-based security and sharing context information on a device with other devices in the Smart Space. Though their research also uses REST paradigm, they have not reported the use of Hypermedia in their architecture. RESTful Platform Broker uses Hypermedia as an engine for application state and hence has a higher level of maturity in accordance with the 3 Level REST Maturity Model (RMM), which was proposed by Richardson [16]. The advantage of using Hypermedia as an engine for application State is that, all the operations supported by resources need not be presented to a client up front. They get to know them, as they interact with the resources and are presented to them during the appropriate time. This allows better separations between the client and server component within a Smart Space and allows them to evolve independently of each other.

It is also remarked that Google has recently designed a framework called Argos [17], which aims at building a Web Centric Application Platform on top of Android. Argos allows developers to access the components of the platform (e.g., Media Player, Camera, Sensors, etc.) by using Java Script instead of Java Programming Language. It does not, however, expose Android resources over the network as done by the RESTful Platform Broker. Further, the aim of the project is to tap a large potential of developers who are trained in Web Application programming and are familiar with scripting languages such as JavaScript but not Java.

III. RESTful Platform Broker

The RESTful Platform Broker is designed and developed using the REST paradigm [18]. REST is an acronym for Representational State Transfer. REST by itself is not an architecture, but a set of constraints, which, when applied, leads to a System Architecture. In order to design or architect a RESTful system, a system designer/architect should visualize the system through the following abstractions:

Resource: A resource is a fundamental building block of a RESTful architecture. A resource is anything that is exposed by the system and is addressable. By addressable, we mean that it can be accessed by other components of the system. Using the REST paradigm, anything can be modeled as a resource and can be manipulated by the components within the system. For example, a camera in a mobile phone, a media player, the reading of a sensor, etc., can be considered as a Resource.

Representation: All the identified resources in the system will have a representation. The representation of a resource is the data being transferred from a server to a client and vice-versa. The representation contains the state of the Resource at a given point in time.

Uniform Resource Identifier (URI): A Uniform Resource Identifier can be considered as a Hyperlink to a resource. By accessing the URIs, the clients and servers can exchange Representations of Resources in the system.

The major advantage of REST architecture as compared to a typical RPC based architecture is that by using REST principles, one can organize a very complex application into simple resources. Once the resources are identified, all operations on these resources can be broken down into Create, Read, Update and Delete (CRUD) operations. This simplicity makes it easy for new clients to use an application, even if it was not specifically designed for them.

The RESTful Platform Broker is implemented as a middleware on top of the Android Platform. It views the network of the Smart Space as a system bus and exposes resources present on the Android platform like Camera, Media Player, Sensors, etc., onto a Smart Space network as a Service Oriented Filesystem [3]. Service Oriented Filesystem provides a Service Oriented Interface (usually based on REST) to the resources in a system. These resources can thus, be accessed by other devices present in the Smart Space. This systemic view has been shown in Figure 1.

Each of the exported resource has a URI associated with it. For example, the Accelerometer Sensor present on the Android Platform may have a URI as shown in [24]. Once the resource has been exposed onto the network, it can be used by other devices, which are present in the Smart Space through its URI . The interaction between a device and a resource exposed by the RESTful Platform Broker is through HTTP. For example, sending a HTTP GET to URI of the Accelerometer will return the current reading of the

 ISBN: 978-1-61208-225-7

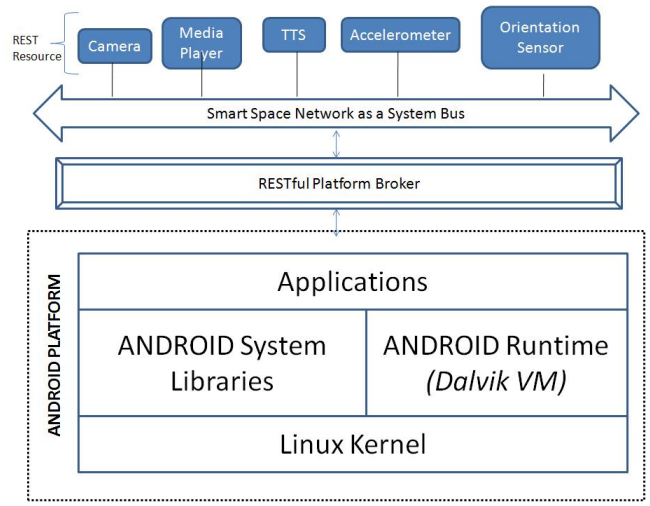

Figure 1. Systemic view of RESTful Platform Broker

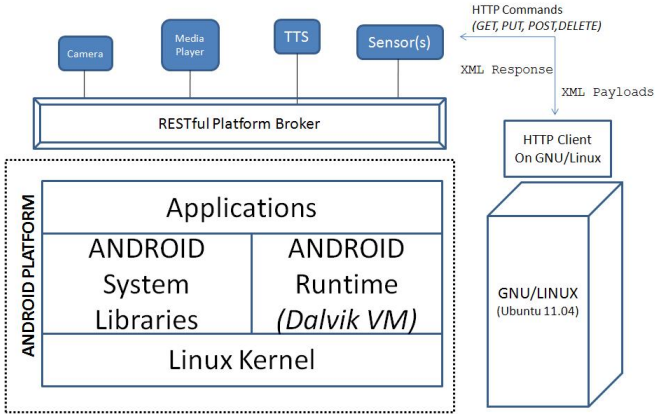

Figure 2. Interaction of RESTful Platform Broker with GNU/Linux

accelerometer. This way, resources are not confined to the physical boundaries of a device, but are available throughout the Smart Space.

Therefore, through the use of the RESTful Platform Broker, devices which may not have a particular resource may use a resource which is present in some other device. The interaction of RESTful Platform Broker with a GNU/Linux system is show in Figure 2.

RESTful Platform Broker uses Hypermedia as the engine of application state (*HATEOAS*). A Hypermedia system is characterized by the transfer of links in the resource representations, which are exchanged between the client and the server. These links advertise other resources that are participating in the application protocol. For example, consider a media player resource which is exposed by the RESTful Platform Broker. An initial HTTP GET on its entry point, e.g., [26] would result in an XML File as shown in Listing 1.

The XML in Listing 1 is an XML representation of the media player resource. The representation has a semantic markup definition provided through *link* and *rel* tags. Consider the first *link* tag, which points to the URI of Media Player Resource itself. The associated *rel* tag has the value `self`. This represents that the URI [26] can be accessed via a HTTP GET to retrieve the latest representation of the Media Player resource. It is remarked that in RESTful Platform Broker, a resource representation will always have a link to itself. This is because when using GET, we always get the latest state of a resource. For example, a subsequent GET to the URI [26] may result in an XML with the status value as Playing instead of Idle. This information can be used by layers above the platform broker to make decisions. For example, a client program can view the status of the media Resource and make a request to playa file only if it is idle. The XML representation of the MediaPlayer Resource has link to another resource namely the SongList.

The SongList Resource can be accessed using HTTP GET, which is inferred by looking at the value of the *rel* tag. A GET to the URI [25] results in an XML representation of a Song List. The XML representation provides the information of the current state and also provides a mechanism to advance to the next state. By reading the XML representation mentioned in Listing 1, a consumer can infer that there are two states possible from the current state namely; getting the representation of the media player by issuing a HTTP GET to the URI [26] or, navigate to get the Song List by issuing a HTTP GET on the URI [25]. A consumer of a resource can therefore, advance through the Domain Application Protocol of the resource exposed by the RESTful Platform Broker. When a consumer issues a HTTP GET on the URI [25], the RESTful Platform Broker generates the following XML Representation of the Song List as shown in Listing 2.

Listing 1. Response of RESTful Platform Broker on the initial GET of the Media Player Resource

```
<mediaresource
xmlns=http://platformbroker.iitd.ac.in
xmlns:mediadap="http://schemas.platformbroker.iitd.ac.in/mediadap">

  <mediadap:linkmediaType="application/vnd.platformbroker+xml"
  uri=http://ipAddress/mediaPlayer/ rel="self"/>

  <mediadap:linkmediaType="application/vnd.platformbroker+xml
  uri="http://ipAddress/mediaPlayer/songList/"rel="self"/>

  <status>Idle</status>

</mediaresource>
```

The XML representation shown in Listing 2 of the song List shows different URIs of songs and their associated *rel* values which provide the semantic markup definition.

Listing 2. XML Representation of Song List

```
<songList xmlns="http://platformbroker.iitd.ac.in"
```

```
xmlns:mediadap="http://schemas.platformbroker.iitd.ac.in/mediadap">

<mediadap:linkmediaType="application/vnd.platformbroker+xml"
uri=http://ipAddress/mediaPlayer/songList/ rel="self"/>

<song mediadap:linkmediaType="application/vnd.platformbroker+xml"
      uri="http://ipAddress/mediaPlayer/songName1/ rel="self"/>

<song mediadap:linkmediaType="application/vnd.platformbroker+xml"
      uri="http://ipAddress/mediaPlayer/songName2/ rel="self"/>

</songList>
```

The XML presented in Listing 3 is generated when a consumer issues a GET to URI [27]. In this XML, apart from *self*, a song's *rel* tag points to `service.put`. By accessing this URI, a consumer can play the Song mentioned in the URI by using HTTP PUT along with a suitable payload, as defined in the platform broker media Type. Therefore, through the use of Hypermedia, a client can proceed ahead to relevant states of a resource by following the corresponding URIs. This is very similar to the way in which a user navigates the World Wide Web. The advantage offered by Hypermedia is therefore loose coupling between the client and server components of the Smart Space. This allows the server components to change its rules and expand available states as needed without affecting the clients.

Listing 3. XML Representation of Song

```
<songxmlns="http://platformbroker.iitd.ac.in"
xmlns:mediadap="http://schemas.platformbroker.iitd.ac.in/mediadap">

<mediadap:linkmediaType="application/vnd.platformbroker+xml"
  uri="http://ipAddress/mediaPlayer/songName1/"
  rel="self"/>

<mediadap:linkmediaType="application/vnd.platformbroker+xml"
  uri="http://ipAddress/mediaPlayer/play/songName1"
  rel="service.put"/>

</song>
```

A. Implementation

The implementation of the RESTFUL Platform Broker is given in Figure 3. The implementation is done on top of the Android Platform. A small footprint HTTP Server has been placed on top of Android. The HTTP server has been implemented as a pure Java based HTTP Server and can be treated as a Virtual Appliance on top of the Dalvik Virtual Machine. The RESTful Platform Broker has a layered architecture and is implemented as a hosted service on top of this HTTP server. The system's core layer consists of the Namespace Generator, which generates a namespace entry of a resource by the information given in the Namespace.xml. The Namespace.xml indicates the URIs of the Resources. The advantage of separation of URIs from

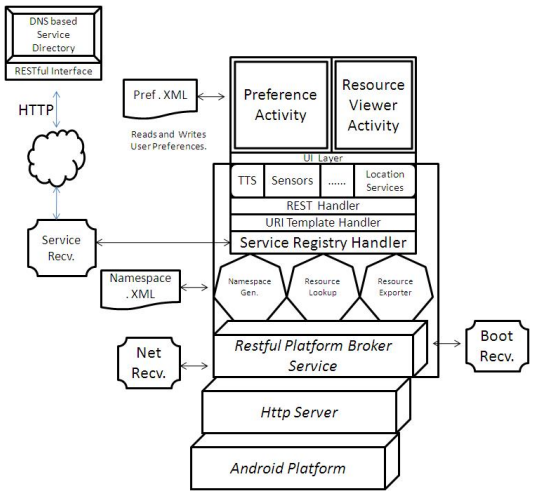

Figure 3. Implementation of RESTful Platform Broker

the RESTful Platform Brokers codebase is that the URIs of the resource can be changed without changing the code. Another component at this layer is the Resource Lookup Manager. The Resource Lookup Manager is a component, which queries the Platform for resources. Once a resource is found it works with the Namespace Generator and the Resource Exporter to export the Resource. The components of this layer of the RESTful Platform Broker, namely the Namespace Generator, Resource Lookup Manager and Resource Exporter are implementations of specific interface *INamespaceGenerator*, *ResourceLookUpManager* and *IResourceExporter* respectively. Throughout the implementation of Platform Broker, programming to an interface paradigm has been followed. This paradigm is useful as it allows multiple implementation of the same concept. For example, the *IResourceLookUpManager* can have different implementation depending on the Platform, it is being implemented for. This helps in easier porting of the RESTful Platform Broker to other Platforms.

The REST Handler and the URI Handler are handlers for the HTTP commands namely GET, PUT, POST and DELETE. These HTTP commands, when sent to the resources (e.g., TTS, Sensors, etc.), are handled by the REST Handlers and URI Handlers for these resources. The difference between the REST handler and the URI handler is as follows. The REST handler can be considered as a singleton class [19] which provides the default behavior to the HTTP Commands whereas, each URI handler for a resource can be multiple for a resource, each implementing a specialized functionality of the resource. For example, the Media Player resource may support two different URIs; one for managing the media file and another for enumerating the media files within the system.

The Service Registry Handler is the component that handles Service Registry in the Platform Broker. It interacts

 ISBN: 978-1-61208-225-7

with DNS based Service Registry to register and query for services that are registered with it. It also interacts with the Service Receiver. The Service Receiver is a broadcast receiver, which informs the Service Registry Handler of new services that have been registered. The RESTful Platform Broker has got two User Interface (UI) components. First, the Preference Activity UI, through which the user can specify the list of resources that can be exported onto the network. Second, the Resource Viewer UI, which shows a list of resources that have been found by Resource Lookup Manager and are available throughout the Smart Space. The Preference Activity writes the list of resources that the user wants to share across a Smart Space persistently in a file called Prefs.xml. The Platform Broker reads this file before exporting the resources to the Network. Further, the user can change the preference of exporting a resource at any point of time. Any such change triggers the export mechanism to reflect the changes in the preferences. This change also triggers the Service Registry Handlers to make suitable changes in the DNS based Resource Registry. Mandatory Access Control (MAC) mechanisms are implemented within the RESTful Platform Broker to prevent the use of resources which are not exposed by the user. The RESTful Platform Broker service interacts with two kinds of Broadcast Receivers namely the Network Receiver and the Boot Receiver. The Network Receiver is an important component especially from the mobility point of view. The Network receiver informs the Platform Broker, when the network is not available. This allows the Platform Broker to stop its activities, thereby conserving the battery of the Mobile device, on which the Platform Broker is installed. Further, when the Network strength is getting weak, it can de-register resources from the DNS Resource Registry. Similarly, when the device moves from a location where there is no network, to a place where network is available, the Network Broadcast Receiver informs the Platform Broker, to start its activities. The Boot receiver is used to start the RESTful Platform Broker, whenever the Android Platform is booted. These are examples of context aware adaptation that has been implemented in the RESTful Platform Broker.

The DNS Server is not a part of the RESTful Platform Broker, but has been implemented separately to provide a centralized mechanism for service registry and discovery. The implemented DNS Server provides a RESTful interface for service registration and service discovery. The RESTful Platform Broker interacts with the DNS Server by using HTTP constructs. The RESTful Platform Broker uses HTTP POST to register a service with DNS Server. The HTTP POST request to register a resource, consists of an XML payload that specifies the resource to be registered. The XML payload consists of the details of the SRV, PTR and TXT records that are required by the DNS Resource Registry. The details of the records are mentioned in Section III-B. The RESTful Platform Broker uses HTTP GET to get information of the available resources within the Smart Space, HTTP PUT to update a resource and HTTP DELETE to remove a resource from the resource registry.

Listing 4. SRV Record of DNS

```
_pb_http._tcp.local IN SRV 0 0 80 GovindPhone.local
```

Listing 5. DNS Reply to Query

```
_pb_http._tcp.local IN SRV 0 0 80 GovindPhone.local
_pb_http._tcp.local IN SRV 0 0 80 GovindLaptop.local
_pb_http._tcp.local IN SRV 0 0 80 GovindTablet.local
```

B. DNS Based Service Discovery

The RESTful Platform Broker supports DNS based service discovery. DNS Service Discovery is a way of using standard DNS programming interfaces, servers, and packet formats to browse the network for services [20]. Since, Hypermedia has been introduced as an engine for application state within the RESTful Platform Broker; a client only need to know the entry URI of a resource. Other URIs are presented as the user uses the resource. For example, a client may only need to know the URI of a media player which may be available in a particular device. Once a client issues a GET command onto this URI, other related URIs relating to usage of the Media player like getting a Song List, Playing a song, Pausing a song, Stopping a song, etc., is presented as and when needed.

A DNS Server can be used to associate various kinds of Resource Records on a particular domain. Apart from records like A-record (for address lookup) or MX-records (for mail server records), DNS also defines resource record types SRV (used to provide location and port for service instances.), TXT (Text Record, used to provide additional meta data about service instances) and PTR (Pointer Record, used to map service types to named service instances). A DNS can be used for service discovery by a combination of SRV, TXT and PTR records. SRV, TXT and PTR records are described below.

Listing 6. DNS SD Configuration

```
_camera_pb_http._tcp PTR GovindPhoneCamera._pb_http._tcp.local
_camera_pb_http._tcp PTR GovindTabletCamera._pb_http._tcp.local
_camera_pb_http._tcp PTR GovindLapCamera._pb_http._tcp.local
_mPlayer_pb_http._tcp PTR GovindPhoneMPlayer._pb_http_.tcp.local
_mPlayer_pb_http._tcp PTR GovindLapMPlayer._pb_http._tcp.local
_tts_pb_http._tcp PTR GovindPhoneTTS._pb_http._tcp.local
_accmeter_pb_http.tcp PTR GovindPhoneAccmeter._pb_http._tcp.local

GovindPhoneCamera._pb_http._tcp SRV 0 0 80 GovindPhone.local.
                                    TXTpath=/camera
GovindTabletCamera._pb_http._tcp SRV 0 0 80 GovindTablet.local.
                                    TXTpath=/camera
GovindLapCamera._pb_http._tcp SRV 0 0 80 GovindLap.local.
                                    TXTpath=/camera
GovindPhoneMPlayer._pb_http SRV 0 0 80 GovindPhone.local.
                                    TXTpath=/mediaPlayer
```

SRV Resource Records: SRV Resource Records are used to provide information about host and port within a zone on which a service is available. As an example, the SRV record for a camera resource is given at Listing 4. The line in the Listing specifies that a Platform Broker is accessible at port 80 on the device whose name is GovindPhone.local. The *.local* is a pseudo top level domain identifying a local domain. The Listing 5 shows a reply to a DNS query. This shows that Platform Broker is running on three devices namely GovindPhone, GovindLaptop and GovindTablet at port 80.

PTR Records: The SRV records have a limitation that they cannot be used to configure named instances of a service type. Further, they only support a single service for any given host and port combination. For example, just by using SRV records, we may not be able to specify all the resources that Platform Broker has exported on a specific device. Therefore, PTR records are used to map service type names to service instance names. The configuration snippet displayed in Listing 6 shows that, on the left hand side of each PTR line, a service type domain name is given and on the right hand side, a corresponding instance of that type is given.

TXT Records: The TXT Records are in the form of a key value pair. In order to use a resource exposed by the RESTful Platform Broker, an entry point of the resource is required. Once this entry point is discovered, other related URIs are presented by the RESTful Platform Broker as and when needed. TXT Records is used for providing an entry point into the resource. The example configuration in Listing 6 specifies that the service instance GovindPhoneCamera._pb_http._tcp can be accessed on GovindPhone.local. at port 80. The TXT record specifies path parameter, which the client must use for constructing the entry URI of the service. In this case, the resource GovindPhoneCamera._pb_http._tcp can be accessed by the URI [28].

The configuration file snippet below in Listing 6 provides PTR, SRV and TXT records for the various resources registered with the DNS Server of the Smart Space.

IV. Application Development Using RESTful Platform Broker

Wireless Accelerometer Based Mouse: This scenario shows the usage of reading sensor information which is available in Android on a Linux Platform. The sensor in this case is an accelerometer. The sensor reading is then used to move a mouse on the laptop by tilting the Android Phone. To construct an accelerometer based mouse using the RESTful Platform Broker, an HTTP client is designed and developed which issues a GET command to an URI at [24] to get the acceleration in X, Y and Z axis. Thereafter, a translation algorithm is used to convert the sensor values onto mouse movements on the target device. By using Gesture Based Toolkits [21][22] mouse events like clicks and double clicks can be made just using Mouse Motion. This could be useful in Smart Space, wherein the users of the Smart Space needs to Control/navigate a big display screen by using their smart phone as a remote mouse. This application demonstrates that while using the RESTful Platform Broker, resources are not constrained onto a particular device and can be made available Smart Space wide. Further, Smart Space programmers can evolve the Smart Space by composing resources (e.g., Accelerometer) and making new services (Wireless Mouse), which were not earlier available.

Reading Contents of a File using a Remote Text To Speech (TTS): Most of the commercially available Android Platforms contains a Text to Speech Engine. This TTS can be made available across the Smart Space through the RESTful Platform Broker and can be used by other devices inside a Smart Space. This arrangement can be used to read a content of file which is present on a remote system (e.g., Linux based Laptop) through TTS present in the Android device. The RESTful Platform Broker exports the TTS at [29]. The Smart Space programmer can write a small program which issues a HTTP PUT command and send a payload consisting of an XML which contains the content to be spoken by the TTS in the Android. This way, the TTS is not restricted to the Android device but is available throughout the Smart Space.

Recording and Playback of Meeting Notes: A common scenario in a smart meeting room is to record minutes of the meeting as soon as the meeting starts. Thereafter the minutes can be played back at a different point in time. RESTful Platform Broker can be used to realize this scenario. The RESTful Platform Broker, exports the Media Recorder at [30]. A very simple client application can be made on any platform to send a HTTP PUT command to start and stop the Recording. The stimulus to start/stop recording may be gathered by systems like RFID or simply by the meeting convener. Later on, the recorded information can be played back using the exported Media Player resource.

Location Aware Services: The RESTful Platform Broker can be used to make Location aware services. Android Based systems typically have a GPS. The RESTful Platform Broker can be used to share the location information from the GPS on the Android throught the Smart Space. The RESTful Platform Broker exports the GPS on the Android device as a URI at [31]. A client application can issue a HTTP GET command to read a geographic location sensed at a particular time (also called a ”fix”). The location information consists of latitude and longitude, a UTC timestamp and optionally, information on altitude, speed, and bearing. This information can then be used by other systems within the Smart Space to design and develop Location Based Application. For example, an application could be designed and developed on a client to do Google search based on the location provided by the GPS. Therefore, a case like searching a hospital through Google could lead to results which are nearer to

 ISBN: 978-1-61208-225-7

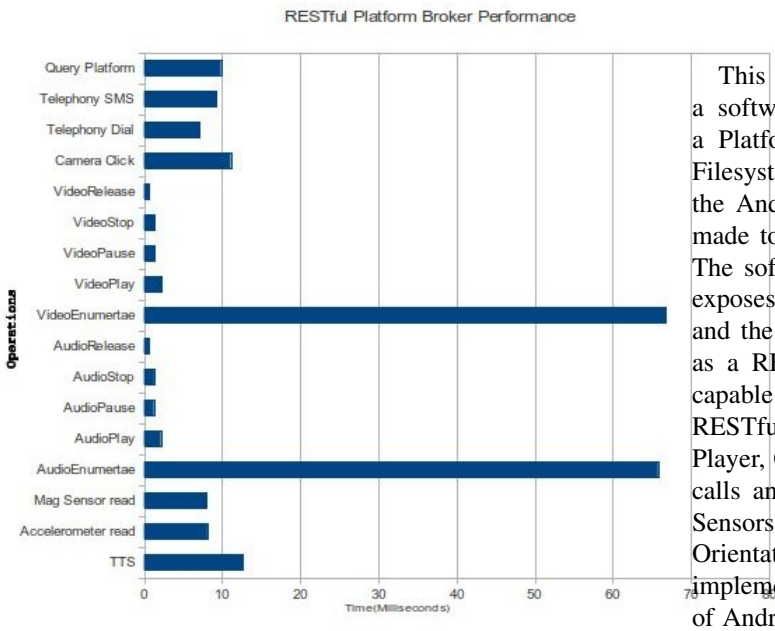

Figure 4. Performance of RESTful Platform Broker

a person who is in the Smart Space.

V. Application Profiling

The test setup for profiling RESTful Platform Broker includes a DELL Latitude E5400 Laptop machine, with Intel Core 2 Duo CPU T7250 at 2.00GHz, 1 GB RAM, 160 GB Western Digital Hard Disk Drive running GNU/Linux. This laptop interact with the RESTful Platform Broker installed on Samsung Based Android Phone (Samsung Galaxy Pop GT-S5570) using a Wireless LAN. In the GNU/Linux system Laptop, a shell script using curl [23] was developed to send HTTP Request to the RESTful Platform Broker on the Phone. While the HTTP Requests was being handled by the RESTful Platform, an application trace file is created in the sdcard of the Android Phone. Android SDKs Trace view utility was then used to analyze the created trace file. The Trace captures the time taken while processing the request apart from other information. A graph is prepared based on the Trace file created while the RESTful Platform Broker was processing the HTTP requests. The graph in Figure 4 shows the comparisons of the time taken to complete various operations by the RESTful Platform Broker. It can be seen that the slowest operations were enumerating media files. This may be attributed to the nature of this operation, which involved searching the Flash Drive for media files. Other operations do not require much of an I/O operation and are hence much quicker. We plan to further enhance the responsiveness of RESTful Platform broker with respect to these operations.

VI. Conclusion

This paper described the design and development of a software system capable of exposing the resources on a Platform (e.g., Android) as RESTful Service Oriented Filesystem. This software has been implemented on top of the Android Platform and an experimental setup has been made to inter-operate with other systems like GNU/Linux. The software is called the RESTful Platform Broker, as it exposes, the entire platform namely the software resources and the hardware resources onto the Smart Space network as a RESTful resource. The RESTful Platform Broker is capable of exposing almost the entire Android Platform as RESTful resources. The exported resources include Media Player, Camera, Telephony Stack (through which Telephone calls and SMSs can be sent), Text to Speech Engine and Sensors such as Accelerometer, Magnetic Field Sensors, Orientation Sensors, Proximity Sensor, etc. Though the implementation of the RESTful Platform Broker is on top of Android, it can be easily ported to other platforms which support a Java Virtual Machine.

RESTful Platform Broker has been designed as a framework for engineering programmable Smart Spaces. Therefore Smart Space programmers can evolve the Smart Space by introducing new services by composition of existing services. However, unlike previous attempts like Gaia [4], our approach does not require use of a particular programming language or use a particular API to do Smart Space programming. Though this feature is similar to PlanB Operating system, the advantage of our approach is that, since it is based on a popular virtual machine, namely the Java Virtual Machine, it can be installed on larger number of device and platforms as compared to PlanB. Further, PlanB was built on Desktop based system, whereas our approach has been to use a Mobile Platform, namely the Android Platform from the inception of the project. One of the major advantages of the RESTful Platform Broker is the use of Hypermedia. Through the use of Hypermedia, it is envisaged that there would be a greater decoupling between the RESTful Platform Broker and the HTTP Clients that uses it. This decoupling, therefore, allows RESTful Platform Broker to grow independently with respect to the clients that use it. The paper also describes some of the scenarios which are typical in a Smart Space environment that can be easily implemented using RESTful Platform Broker, with some amount of programming. It is however, envisaged that with the improvements in Visual Programming, Scripting Languages and Domain Specific Languages (DSL), the use of RESTful Platform Broker, may become easier and maybe used by non programmers also. The current version of the RESTful Platform Broker uses the MAC mechanism to control the visibility of resources within the Smart Space. In the future versions of RESTful Platform we plan to incorporate OpenID [32] and OAuth [33] for implementing

 ISBN: 978-1-61208-225-7

a more robust security mechanism.

REFERENCES

[1] M. Satyanarayanan, “Pervasive computing: Vision and challenges”,IEEE Personal Communications, vol. 8, pp. 10-17, 2001.

[2] S. Helal, W. Mann, H. El-Zabadani, J. King, Y. Kaddoura and E. Jansen, “The Gator Tech Smart House: A Programmable Pervasive Space”, Computer, vol. 38, March 2005 pp. 50-60.

[3] E. Hensberg, N. Evans and P. S. Marble, “Service oriented filesystems”, IBM Research Report, RC24788 (W0904-091), IBM Research Division Austin, IBM Research Zurich, 2009.

[4] M. Roman and R.H. Campbell, “Gaia: Enabling active spaces”, Proc. 9th Workshop on ACM SIGOPS European Workshop, ACM Press, New York, pp. 229-234, 2000.

[5] J. Al-Muhtadi, R. Campbell, A. Ranganathan, S. Chetan and M. Mickunas, “Olympus: A high-level programming model for pervasive computing”, Proc. 3rd IEEE Intl. Conf. on Pervasive Computing and Communications, pp. 7-16, March 2005.

[6] J. Al-Muhtadi, R. Campbell, E. Chan, and J. Bresler, “Gaia Microserver: An Extendable Mobile Middleware Platform”, Proc. 3rd IEEE Intl. Conf. on Pervasive Computing and Communications, pp. 309-313, March 2005.

[7] B. Johanson, A. Fox and T. Winograd, “The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms”, IEEE Pervasive Computing 1(2), pp.67-74, April-June 2002.

[8] A. Fox, B. Johanson, P. Hanrahan and T. Winograd, “Integrating Information Appliances into an Interactive Workspace”, IEEE Computer Graphics & Applications, pp.54-65, May-June 2000.

[9] S. Sen, J. Erman and A. Gerber, “HTTP in the Home: It is not just about PCs”, Proc. ACM SIGCOMM Workshop on Home Networks (HomeNets), pp.90-96, 2010.

[10] J. Mathrick, J. Humble, C. Greenhalgh, S. Izadi and I. Taylor, “ECT: A Toolkit to Support Rapid Construction of Ubicomp Environments”, Proc. UbiComp 2004, pp.207-234, Springer, 2004.

[11] F. J. Ballesteros, E. Soriano, G. Guardiola, K. Leal, “Plan B: Using Files Instead of middleware abstractions for pervasive computing environments”, IEEE Pervasive Computing 6(3) pp. 58-65, Aug 2007.

[12] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom, “Plan 9 from bell labs”, Computing Systems, vol. Vol. 8, pp.221-254, Summer 1995.

[13] R Pike, D. Presotto, K. Thompson, H. Trickey and P. Winterbottom, The Use of Name Spaces in Plan 9“, ACM SIGOPS Operating Systems Review, Vol.27(2), pp.72-76, Apr. 1993.

[14] E.V. Hensbergen and R. Minnich, ”Grave robbers from outer space: Using 9P2000 under linux“, USENIX 2005 Annual Technical Conference, FREENIX Track, pp.83-94, 2005.

[15] C. Prehofer, J. Gurp, V. Stirbu, S. Sathish, P.P. Liimatainen, C. Flora and S. Tarkoma, ”Practical web- based Smart Spaces“, IEEE Pervasive, Vol 9(3), pp.72-80, 2010.

[16] L. Richardson, ”Justice will take us millions of intricate moves“, http://www.crummy.com/writing/speaking/2008-QCon/, QCon San Francisco, 2008 [retrieved: March, 2012].

[17] R. Gossweiler, C. McDonough, J. Lin, and R. Want, Argos: Building a web-centric application platform on top of android, IEEE Pervasive Computing, vol.10(4) pp.10-14, 2011.

[18] R. Fielding, Architectural Styles and the Design of Network based Software Architectures, PhD thesis, Doctoral dissertation, University of California, Irvine, 2000

[19] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison- Wesley, 1994.

[20] Dns service discovery, http://www.dns-sd.org/ [retrieved: March, 2012].

[21] Easystroke: A gesture recognition application for x11, http://sourceforge.net/apps/trac/easystroke/wiki, [retrieved: March, 2012].

[22] Gestikk - Mouse gesture recognition in Ubuntu, https://launchpad.net/gestikk, [retrieved: March, 2012].

[23] Curl and libcurl, http://curl.haxx.se/, [retrieved: March, 2012].

[24] http://ipAddressOfAndroid/Sensors/Accelerometer [retrieved: March, 2012]

[25] http://ipAddress/mediaPlayer/songList/ [retrieved: March, 2012]

[26] http://ipAddress/mediaPlayer [retrieved: March, 2012]

[27] http://ipAddress/mediaPlayer/songName1 [retrieved: March, 2012]

[28] http://GovindPhone.local/camera [retrieved: March, 2012]

[29] http://ipAddress/tts [retrieved: March, 2012]

[30] http://ipAddress/MediaRecord [retrieved: March, 2012]

[31] http://ipAddress/GPS [retrieved: March, 2012]

[32] B. Ferg et al., OpenID Authentication 2.0Final, OpenID Community, Dec. 2007; http://openid.net/specsopenid-authentication-2_0.html [retrieved: March, 2012]

[33] M. Atwood et al., OAuth Core 1.0, OAuth Core Workgroup, Dec. 2007; http://oauth.net/core1.0. [retrieved: March, 2012]

 ISBN: 978-1-61208-225-7