A Scalable Service Oriented Distributed Computing Architecture : Prototype Design and Implementation

Xitij Shukla
College of Agricultural Information Technology
Anand Agricultural University, Anand, India
xus@aau.in

N. N. Jani
Faculty of Computer Science and IT
KSV University (retd.),
Gandhinagar, India

ABSTRACT

Distributed Computing has remained a phenomenon focusing either on owned infrastructure or on volunteered resources geographically spread presenting only abstract view to its users. The prototype architectural design and its implementation in the present article follow a layered model. The model is implemented to demonstrate an extremely affordable Distributed Computing ecosystem with commonly available hardware such as single board computers and microcontrollers using Free and Open Source Software. This ecosystem presents a platform independent, architectural neutral and programming language agnostic Distributed Computing architecture for sake of simplicity, affordability and ease of deployment.

Keywords— Distributed Computing, IoT, Web Services

I. INTRODUCTION

Distributed computing is an architectural representation of different computational resources such as CPU cycles and storage geographically dispersed are collectively utilised to bring the solution over internet. With the introduction of concept of *virtual organisations* and the Grid, distributed systems gained wider popularity in resource sharing and problem solving in dynamic manner at inter-institutional level. They present a type of parallel and distributed architecture enabling sharing, exchange, selection and aggregation of geographically distributed autonomous resources depending upon their availability, capability, cost and required levels of Quality of Services by the user. The spread of distributed computing is not limited to resource utilisation at intra-institutional level; but it has also been spread towards inter-institutional level for generation of a common resource pool in dynamic manner, in order to use it to bring the desired solution. Foster, redefined and simplified it as a system, "That coordinates resources, that are not subjected to central control, using standard, open and general-purpose protocols and interfaces to deliver nontrivial qualities of service". However, as Foster stressed that, contrary to cloud computing, which focuses on commercial outsourced models for computing the distributed architecture presented here limits itself to non-commercial collaboration of federated computing resources.

II. TOOLS

Resources in the distributed computing infrastructure follow a decentralized and heterogeneous pattern. This attribute eventually necessitates a universal, open, platform independent, architecture neutral set of environment for the reasons of interoperability. Further, this infrastructure not only remains human centric but also remains application (machine) centric ; involving frequent underlying application to application exchange of messages across the its computational resources. A Web Service provides capability of performing certain function over network. In general, a Web Service can be summarized as an interface built using standard Internet technology for certain application functionality accessed over network. The interoperability is achieved through a set of multi-tier operations. A machine-readable interface descriptor i.e. Web Services Description Language (WSDL) describes the Web Service for interaction with other resources. UDDI (Universal Description, Discovery and Integration) is used to advertise these Web Services. SOAP protocol is used to

interact with the Web Service as per the WSDL description. SOAP messages wrapped in XML (eXtensible Markup Language) are exchanged generally through the Hypertext Transfer Protocol (HTTP) over the network for end-to-end transport. Web Services add a layer of abstraction between the legacy code and the application layer, hiding underlying platform specific and language specific implementation details from the user by a programming language neutral, platform independent and architecture agnostic, standard interface accessible over network. Figure 1 depicts this phenomenon.

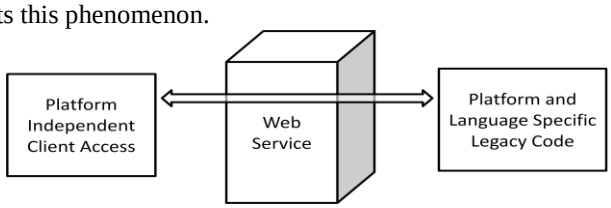

Figure 1 Web Services Abstraction Layer

The architecture is built over SOAP (referred as Simple Object Access Protocol when associated with Remote Procedure Calls style implementation and referred as Service Oriented Architecture Protocol when associated with web service consumption) acts as a common data transport protocol to exchange structured messages. HTTP POST method is the common approach for SOAP transport with encoding MIME type at either ends set at **text/xml**. The concerned HTTP server is required to be equipped with SOAP processor in order to deal with SOAP messages. Every SOAP message is surrounded by a SOAP Envelop consisting of the root element *Envelope*. The SOAP Envelope consists of an optional SOAP header and the mandatory message body. SOAP header may include application specific information such as handshaking information, electronic transaction identifier, security token etc. Whereas, the SOAP message body carries the message as payload consisting of methods, arguments, values etc. The XML schemas associated with the envelope assist in interpretation of SOAP response. Upon success, the HTTP server responds with the HTTP status code 200, however, in case of problems from the server side it responds with the concerned status code starting with 5, indicating server-side error.

WSDL (Web Services Description Language from version 2 onwards instead of Web Services Definition Language) acts as glue between the Web Services Provider and the Consumer. A SOAP web service is described using WSDL confirming to XML grammar in a platform-independent, programming-language agnostic and architecture neutral fashion. It provides information regarding publically available functions, data types, protocol bindings and URL of associated web service. The concerned WSDL file must be accessible by the both Client and Server to initiate the web service transaction. WSDL allows the web service consumer to locate the web service and execute the public function. Client and Server identify data exchange according to a pre-agreed XML schema. A WSDL file wrapped as a valid XML document consists of the elements viz. **definitions**, **types**, **message**, **portType** and **binding**.

The **definitions** element is the root element of a WSDL description file consisting of references to various XML namespaces used elsewhere in the WSDL document. Remaining XML elements are wrapped within the **definitions** element. The **types** element consists of description for data types of messages exchanged across the client and the server. W3C XML Schema compliant type definition is the default approach for type definition within the WSDL document. The **message** element consists of a message from the single side i.e. from client to the server and *vice-versa*. It may carry message parts acting as input arguments and return values. The **documentation** is an optional element providing human readable service description. This element can be embedded inside other WSDL elements.

The **portType** or interfaces element allows grouping of multiple associated SOAP operations from anyone of operations viz. one-way, complete roundtrip, wait-for-response or just-notify. The **binding** element defines the actual service transport for operation and message exchange associated to the respective i.e. *bound* **portType** in either *document* style or *RPC* style binding. The **service** element carries URL of the hosted SOAP service along with the service endpoint i.e. **port**. Figure 2 depicts WSDL element stack.

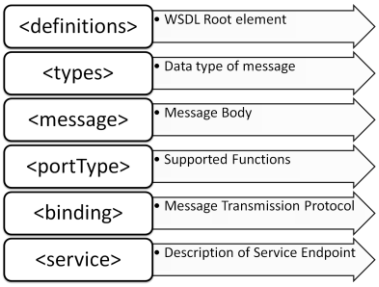

Figure 2 WSDL element stack.

III. PROTOTYPE DESIGN AND IMPLEMENTATION

Design of a Hybrid, Modular, Scalable and Service Oriented distributed computing model is presented as a prototype. The architecture comprises of four layers. It has been termed Hybrid since; it follows a tightly coupled approach at the bottom layers, simultaneously maintaining loosely coupled approach at the higher layers. These layers are namely; Resource Layer, Resource Abstraction Layer, Service Layer and Application Layer. Principal components of this prototype architecture are classified into four distinct categories, namely; The Resource, The Resource Aggregator, The Service Request Negotiator, and The Consumer. These components correspond with layers of this architecture respectively as depicted in figure 3.

Figure 3, The Four-layered Prototype Architecture

Table 1 consists of listing of hardware and software used for prototyping.

Table 1 Summary of Hardware and Software Components

Sr.	Layer	Hardware Components	Software Components
1.	Resource	Arduino Microcontroller	Arduino IDE
2.	Resource Abstraction	Raspberry Pi	Python
3.	Resource Negotiation	Generic Computer	SOAP, JAVA, WSDL
4.	Application	Generic Computer	Python, Java, C++

A. The Resource Layer

The Resource layer is the bottom most component of the prototype architecture. It is modelled for aggregation of geographically distributed sensors sensed via micro-controllers. For the prototype, Arduino UNO series microcontroller board has been identified. It allows interfacing various sensors – *resources* and read from, and written to values to the same such as reading current temperature etc. This layer being the underlying one, deals directly with the resources however, the systematic in cooperation of upper layers actual location, physical implementation and other lower level details insignificant from the end-users perspective are hidden.

B. The Resource Abstraction Layer

The Resource Abstraction Layer stays between the lower most Resource Layer and the Resource Negotiation Layer and follows the principle of state machine. It maintains a tightly coupled sync with the microcontroller at the Resource Layer. This layer initiates the transaction to trigger the underlying resources via the Resource Layer through atomic getter and setter instructions as received from the higher layer. It conveys back the associated values for further processing to the higher level layer. In order to balance the overhead caused by polling, fixed time intervals are introduced to maintain the sync. This layer is implemented via Raspberry Pi, – a singleboard, and affordable computer with a credit card sized form-factor. Tightly coupled link between this layer and the bottom layer is achieved using the GPIO (General Purpose Input Output) feature of this board.

C. The Service Layer

The Service layer acts as a fabric across the proposed infrastructure. It augments platform independence, architectural neutrality and programming language agnostic behaviour across heterogeneous hardware to the proposed distributed computing ecosystem over TCP/IP. This layer broadcasts available services and undertakes implicit negotiation with attached resources abstracted through the underlying Resource Abstraction Layer. Loosely coupled link between the Service Layer and Resource Abstraction Layer is

implemented using TCP/IP network. This approach lets the Service Layer to hide actual implementation of underlying layers from the consumers.

D. **The Application Layer**

The end user accesses the computing resources through this layer. However, the underlying layers hide the actual resource location and its associated network address offering location transparency to the system. The Application Layer comprises of client utilities to incorporate with the Service Layer. Since, the backbone of the prototype distributed computing infrastructure is built over the web services; the client can request access, consume and subsequently releases the computing resources marshalled through the service stub advertised by the Service Layer. This layer maintains loosely coupled approach with the undeerlying Service layer over TCP/IP network. It can be implemented on the variety of hardware, platforms and programming languages, as it needs a mere working network connectivity with the Service Layer computer. The hybrid setup i.e. loosely coupled connectivity at the higher level layers and tightly coupled connectivity at the lower level layers represent a Service Oriented distributed computing architecture providing transparent access to services.

The model with respective hardware – when prototyped with associative layers appears as shown in figure 4.

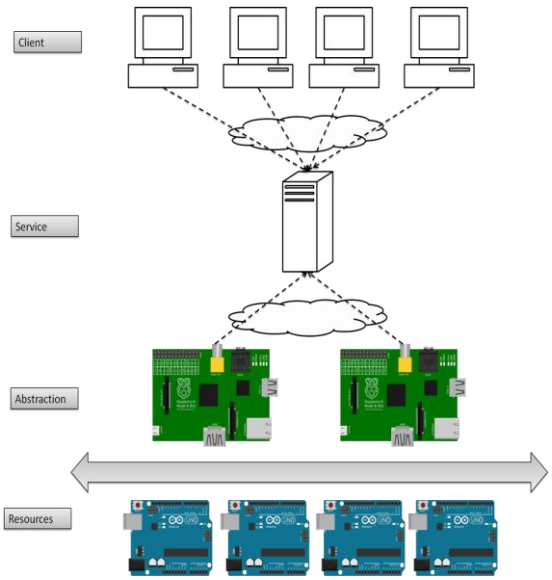

Figure 4 Components corresponding to layers

E. **Implementation**

For the prototype ecosystem, the Arduino board is connected as slave with Raspberry Pi GPIO (General Purpose Input Output) connectivity. The I2C (Inter Integrated Circuit) bus master slave protocol is selected over the other alternatives, being limited number of USB ports (two) on the Raspberry Pi board itself. I2C carries two wire lines viz. SDA (data) and SCL (clock). The I2C approach allows multiple (up to 128) devices as slaves to be connected. The **`Wire.h`** Arduino library enables the Arduino board to communicate with I2C bus. The Raspberry Pi single board computer at the Resource Abstraction Layer carries 17 GPIO pins allowing interaction with other devices. Amongst these pins, GPIO2 and GPIO3 pins act as I2C data (SDA) and clock (SCL) respectively. However, the difference of working voltage ratings i.e. Raspberry Pi at 3V and Arduino at 5V is addressed through a logic level converter. Arduino I2C and Raspberry Pi I2C pins are bridged through the logic level converter as per the schematic as shown in the figure 5. Figure 6 represents the actually implemented system.

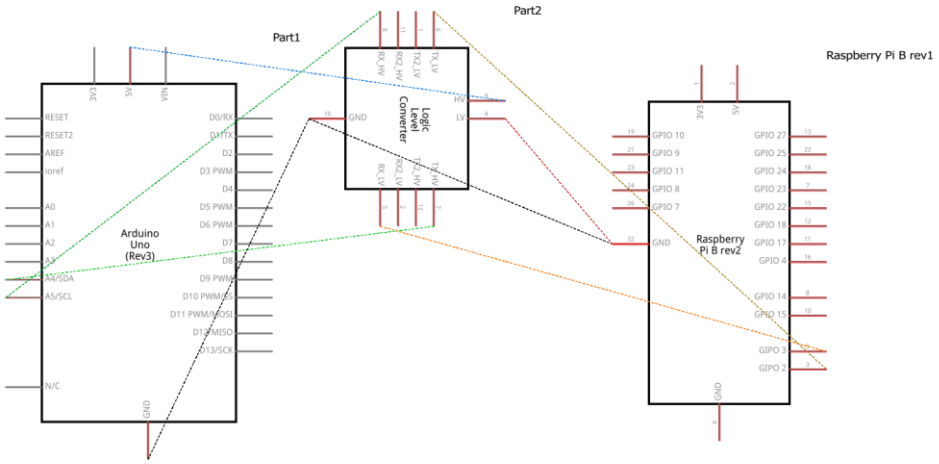

Figure 5 Schematic I2C Bridge for the Prototype

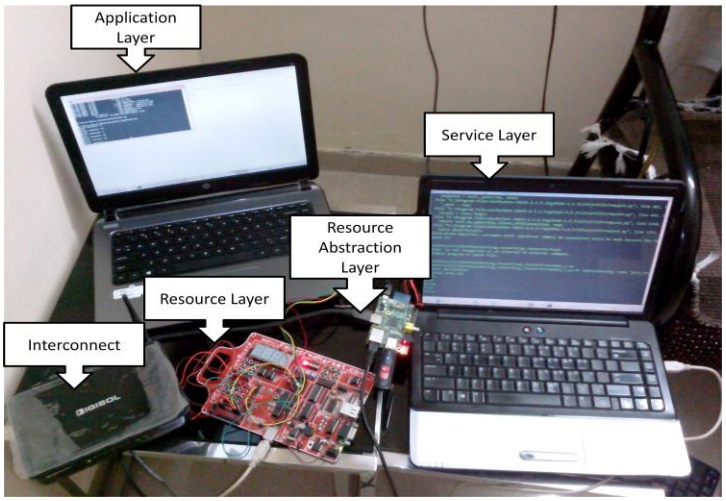

Figure 6 Actual Implementation of the Prototype

The Raspberry Pi singleboard computer natively supports Python, the general purpose programming language. Two Python libraries are used to implement the Resource Abstraction Layer; the **smbus** library for interfacing GPIO I2C components on the Raspberry Pi board and **Suds (Jurko's Fork)** library for negotiation with SOAP Web Services. The **smbus** library allows to read from and write to I2C GPIO of the Raspberry Pi with the functions **read_byte()** and **write_byte()** respectively using predefined I2C address of the SLAVE device (**0x04 Arduino Pin 4** in the cases).

The Service Layer acts as the heart of the entire prototype ecosystem using a document oriented, SOAP Web Service. It augments this system with the capability to negotiate with the resources and access the same in a stateless, loosely coupled, platform independent, architectural neutral, programming language agnostic fashion over HTTP protocol. SOAP Web services are used to implement the Service Layer in Java on the Eclipse IDE. The Service Layer is powered by the Apache Tomcat application container on Microsoft Windows platform along with Apache CXF, the Open Source SOAP Services framework.

SOAP can be implemented either by writing a WSDL as a starting point of development and generation of associated the Java code or implementing a Java code and SEI (Service Endpoint Interface) as a starting point and generation of associated WSDL accordingly. The latter A Service Endpoint is a network accessible stub which allows the stakeholders (i.e. the Service Provider and Service Consumer) of the SOAP Web Service to communicate by exchange of machine processable messages in from of XML. The latter approach is considered for the prototype development. The Application Layer allows the end-user to access the service. The client code can be written in any language supporting SOAP. The prototype implements client using Python programming language.

IV. CONCLUSION AND FUTURE WORK

While existing implementations remain framework dependent and heavyweight causing challenges to design, learn, collaborate and modify the infrastructure, the prototype demonstrates decentralized control, platform independence, architectural neutralism and programming language agnostic and transparent access behaviour. It allows the end-user to query, use and release the resources aggregated through heterogeneous hardware without knowing about the underlying geographical location and technologies. The prototype design can also be used for IoT (Internet of Things) by allowing sensors and actuators to be remotely sensed and managed.

Secured implementation over single sign on (SSO) credentials and a trusted certificate authority, a Service Layer portal for effective deployment of services and monitoring, implementation of RTC (Real Time Clock) at the bottom layers for time stamping of resource utilization and an application layer portal that allows the end-users to identify occupied and available resources according to their location and nature are some of the potential future thrust areas to consider.

REFERENCES

[1] J. Holland, ―A Universal Computer Capable of Executing an Arbitrary Number of Subprograms Simultaneously‖, Proc. East Joint Computer Conference, Vol. 16, pp. 108-113, 1959.

[2] I. Foster, C. Kesselman and S. Tuecke, ―The Anatomy of Grid: Enabling Scalable Virtual Organisations‖, International Journal of Supercomputer Applications, 2001.

[3] Madhu Chetty and Rajkumar Buyya, ―Weaving Computational Grids: How Analogous Are They with Electrical Grids?‖, Computing in Science and Engineering (CiSE), Vol. 4, Issue 4, IEEE CS Press, USA, July-August 2002

[4] I. Foster, ―What is Grid? A Three Point Checklist‖, Argonne National Laboratory & University of Chicago, Grid Today, 2002.

[5] Foster, I., Zhao, Y., Raicu, I., & Lu, S. (2008, November). Cloud computing and grid computing 360-degree compared. In *2008 grid computing environments workshop* (pp. 1-10). IEEE.

[6] Booth, D. (2004). Web services architecture. *W3C note*.

[7] The Web Services Description Language, version 2.0. See http://www.w3.org/TR/wsdl20

[8] Banzi, M., & Shiloh, M. (2014). Make: Getting Started with Arduino: The Open Source Electronics Prototyping Platform. Maker Media, Inc..

[9] Mills, B. I. (2005). Theoretical introduction to programming. Springer Science & Business Media.

[10] Semiconductors, N. X. P. "I2C-bus specification and user manual." Rev 6 (2014).

[11] Arduino Wire Reference‖, https://www.arduino.cc/en/reference/wire (2015).

[12] Bi-Directional Logic Level Converter Hookup Guide‖https://learn.sparkfun.com/tutorials/bi-directional-logic-level-converterhookup-guide (2015).

[13] Raspberry Pi Reference‖ https://www.raspberrypi.org/products/raspberry-pi-2- model-b/ (2015).

[14] Raspberry Pi, FAQs‖, https://www.raspberrypi.org/help/faqs/ (2015).

[15] smbus Python I2C Library‖, https://pypi.python.org/pypi/smbus-cffi (2015).

[16] Jurko's Suds SOAP fork‖, https://pypi.python.org/pypi/suds-jurko/0.6 (2015).

[17] Apache CXF: An Open Source Service Framework. http://cxf.apache.org/ (2015).

[18] Apache Tomcat Application Container, http://tomcat.apache.org (2015).

[19] Developing a Service with JAX-WS, http://cxf.apache.org/docs/developing-aservice.html, (2015).