ScaleUp: middleware for intelligent environments

Daniyal Alghazzawi [Corresp., 1] , **Ghadah Aldabbagh** [1] , **Abdullah Saad AL-Malaise AL-Ghamdi** [1]

[1] Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah, saudi Arabia

Corresponding Author: Daniyal Alghazzawi
Email address: dghazzawi@kau.edu.sa

The development of the internet of things (IoT) expands to an ultra-large-scale, which provides numerous services across different domains and environments. The use of middleware eases application development by providing the necessary functional capability. This paper presents a new form of middleware for controlling smart devices installed in an intelligent environment. This new form of middleware functioned seamlessly with any manufacturer API or bespoke controller program. It acts as an all-encompassing top layer of middleware in an intelligent environment control system capable of handling numerous different types of devices simultaneously. This protected de-synchronization of data stored in clone devices. It showed that in this middleware, the clone devices were regularly synchronized with their original master such as locally stored representations were continuously updated with the known true state values.

1   **ScaleUp: Middleware for Intelligent Environments**

2   **Running Title:** Middleware for Intelligent Environments

4   **Daniyal Alghazzawi (corresponding author)**

5   **(IEEE Senior member): Faculty of Computing and Information Technology, King**

6   **Abdulaziz University, Jeddah, Saudi Arabia; dghazzawi@kau.edu.sa**

8   **Dr. Ghadah Aldabagh**

9   **Computer Science Department, FCIT, King Abdulaziz University, Saudi Arabia**

11   **Dr. Abdullah Alghamdi**

12   **(IEEE Senior member): Faculty of Computing and Information Technology, King**

13   **Abdulaziz University, Jeddah, Saudi Arabia**

21   **Abstract**

22   The development of the internet of things (IoT) expands to an ultra-large-scale, which provides

23   numerous services across different domains and environments. The use of middleware eases

24   application development by providing the necessary functional capability. This paper presents a

25   new form of middleware for controlling smart devices installed in an intelligent environment.

26   This new form of middleware functioned seamlessly with any manufacturer API or bespoke

27   controller program. It acts as an all-encompassing top layer of middleware in an intelligent

28   environment control system capable of handling numerous different types of devices

29   simultaneously. This protected de-synchronization of data stored in clone devices. It showed that

30   in this middleware, the clone devices were regularly synchronized with their original master such

31   as locally stored representations were continuously updated with the known true state values.

32   **Keywords:** Intelligent Environment Control Systems, Middleware, Pervasive Computing, Smart

33   Devices

1. Introduction

Architectures for intelligent environments typically require some form of necessary middleware layer, enabling installed hardware (smart devices) and software, (agents or controller programs), to link and operate together. Various types of middleware are created to perform this crucial role, such as Universal Plug and Play (UPnP), Web Services, and bespoke mechanisms written by device manufacturers [1].

Middleware may vary in terms of functionality, with different strengths and weaknesses. These strengths and weaknesses largely depend on their deployment conditions. This is challenging for the programmers that adopt unenviable measures when designing control systems, particularly for complex smart devices or intelligent environments. Selecting a particular mechanism requires trade-off of resources or functionality to a certain extent [2]. Some bespoke control system middleware designs help recover functionality loss by reproducing components that help to extend the chosen mechanism. Alternatively, programmers could implement an architecture where multiple independent middleware layers run simultaneously within a single control system. Both of these solutions naturally require a greater coding effort to implement the system, which itself is a lengthy program processing than the default implementation.

A potential risk of data synchronization also prevails within the system [3]. For instance, the device state variability; updated using one middleware system, cannot be reflected by all the alternative mechanisms used. If their state data became desynchronized, smart devices controlled by agents may exhibit erroneous or undesirable behaviours, which could prove costly and dangerous. These issues also emerge due to hardware action request response times. If the new state data needs to be shared amongst several components within the middleware (e.g., to

59 synchronise the system), this additional processing could be slower to changing events within the

60 environment. This could frustrate users when updating their device and makes operating agents

61 more apparent within an intelligent environment. For example, walking into a dark room and

62 needing to wait for the lights to switch on.

63   This paper highlights work performed by an ongoing international research project (ScaleUp

64 project) investigating methods for scaling up intelligent environments. As a consequence of this

65 research, a new middleware was developed, specifically designed to address the trade-off issue,

66 when deciding between functionality provided by UPnP and Web Services implementations. The

67 aim was to amalgamate distinctive beneficial features offered by the UPnP and Web Services

68 approaches into a new middleware design. The new mechanism was also structured to protect

69 intelligent environment control systems against data de-synchronization, and processing action

70 requests in times at least on par with other existing middleware examples. This work was based

71 on ongoing international collaboration between the University of Essex and King Abdulaziz

72 University.

73   The core contribution of this study is an evaluation of both the state-of-the-art and state-of-

74 the-practice in the middleware research offered by existing middleware products. Software

75 engineers progressively utilize middleware for building distributed systems. Any research into

76 ScaleUp project that avoids this trend will merely have confined effect. Therefore, this paper has

77 analysed the effects of middleware on the software engineering research agenda. It is argued that

78 requirements engineering techniques are required for focusing on non-functional requirements,

79 as these impact the selection and use of middleware. Software architecture research can develop

80 methods guiding students towards the selection of right middleware and integrating it so that it

81 fulfils a series of non-functional requirements.

82   **2. Literature Review**

83   **2.1 Related Work**

84   Currently, many off-the-shelf commercial devices possess inbuilt intelligent functionality,

85   (i.e., either contain or can be accessed in some way via a computer system) [4]. To facilitate

86   usability of their products, hardware manufacturers include pre-installed middleware designed

87   for computer control. Previously, control systems were device-centric; for example, the control

88   system for washing machine cycle programming. However, with the widespread use of

89   networking platforms such as Ethernet, Wi-Fi, and Bluetooth, the interconnectivity between

90   different devices began to emerge. Being proponents of Pervasive Computer Science concepts,

91   such as the Internet-of-Things [5], several hardware manufacturers chose to implement

92   middleware for their products around established common frameworks, such as UPnP.

93   This was due to a belief that by adopting common middleware architecture, customers would

94   use smart devices created by rival companies, within their proprietary personal home networks.

95   Though, this concept is yet to be turned into a mainstream commercial reality. Various larger

96   multinational hardware manufacturers, such as Apple or Sony are adopting more protectionist

97   marketing strategies, creating interconnectivity between their products and services. For

98   example, Apple iWatch smart device release will only work when paired with an Apple iPhone 5

99   or newer, though it is not compatible with any other older iPhone models or any rival smart-

100   phone handset [6]. Such smart devices operate using completely bespoke proprietary middleware

101   systems, designed solely by a particular company.

102   The current literature shows that the use of middleware in IoT is limited [7]. Ngu et al [8]

103   have designed an IoT application for real-time prediction of blood alcohol content based on

104   smartwatch sensor data. The study has conducted a survey on the competencies of the current

105 IoT middleware. In addition, the challenges and the enablers associated to the IoT middleware

106 were presented in order to embrace the heterogeneity of IoT devices. Fremantle and Scott [9]

107 have utilized a structured search approach for identifying 54 particular IoT middleware

108 frameworks and examined the security frameworks associated to each middleware. A total of 12

109 requirements (integrity and confidentiality, access control, consent, policy-based security,

110 authentication, federated identity, and device identity) were used from the first stage for

111 validating the competencies of each system.

112 Elkhoder et al [10] have accounted middleware for the emerging attributes such as seamless

113 communications, lightweight aspects, and mobile across different heterogeneous networks and

114 domains. It involves a context-adaptive technique, which allows the user for managing the

115 location information shared by things on the basis of policy enforcement and context-aware

116 mechanism. This mechanism accounted both the preferences and informed consent of a user.

117 Jyothi [11] has managed data volumes and supported semantic modeling in the open issues,

118 specifically managing the crowd sourcing of different domains. There is a scope for research

119 work in order to make a generic IoT-middleware system, which is relevant across all regions by

120 making all the functional aspects reusable and can be included as enabler to the middleware

121 system.

122 Razzaque, Milojevic-Jevric, and Palade [12] survey showed that the use of middleware

123 assists in the development through its integration of heterogeneous communication and

124 computing devices. It also states that middleware provides interoperability support for

125 application across different devices and services. Jeon and Jung [13] have revealed that the

126 average request rate elevated by 25 percent compared to Californium, which is a middleware for

127 effective association in IoT environments with vigorous performance, a power consumption

128   reduced by up to 68% and an average response time reduced by 90% when resource management

129   was utilized. Lastly, the latency and power consumption of IoT devices can be reduced by the

130   proposed platform. According to da Cruz et al [14], an important role is played by middleware as

131   it is accountable for covering the intelligence part in IoT, which make decisions, allow them to

132   communicate, and integrate data from devices on the basis of data collected. Afterward, a

133   reference architecture model was investigated for IoT middleware based on IoT platform

134   requirements, which detail the effective operational approaches of each proposed module, and

135   proposed fundamental security features for this software type. Zhao et al [15] have proposed a

136   stack of support-communication-computing for integrating effective open-source projects in

137   order to devise techniques for allowing sufficient uniform human-thing associations, and

138   developing implementation foundations for cutting-edge technologies including semantic

139   reasoning and fog computing.

140   Due to the lack of common frameworks adoption for building complex intelligent

141   environments or devices, the addition of a bespoke middleware layer is critical. This layer is

142   designed to collectively handle the respective control systems of each installed smart device

143   type. This extra layer of middleware wraps the individual bespoke device controller programs

144   into a single API, representing the entire intelligent environment. This is also created by different

145   Computer Science projects, which create middleware for intelligent environments, linking their

146   creations into bespoke device controllers. Although all might concentrate on the same area of the

147   control system architecture, the actual functionality can have a broad design. For example,

148   Román et al. [16] created their bespoke Gaia infrastructure to allow distributed collections of

149   smart objects and environments, represented and accessed via an interface. Later, Roalter et al.

150   [17] chose to integrate a control system for a robot into their intelligent environment middleware,

151   allowing it to seamlessly access sensors and actuators within the smart space. In their OpenCOPI

152   middleware, Lopes et al. [18] utilized a web ontology language based around semantic web

153   services to create a mediator regulating access between ubiquitous applications and available

154   service providers. Stavropoulos et al. [19] also used web Services as the base for their

155   aWESoME infrastructure, which in addition to being an intelligent environment controller,

156   focused on promoting "energy savings" by consuming low power in its operations. Finally,

157   Olaru et al. [20] used a context-aware multi-agent system as their middleware base for

158   controlling ambient intelligence exhibited within an environment.

159   Different middleware platforms were discussed on these criteria for networked robotic

160   systems [21]. Majority of the middleware platforms have varied objectives including reusability,

161   development process, self-discovery, self-configuration, supporting QoS, flexibility, and

162   integration. In addition, several middleware platforms were discussed in the study for networked

163   robotic systems such as self-adaptation, discovery and higher-level abstractions, collaborations

164   support, and other advanced characteristics for integration.

165   Rodriguez-Molina and Kammen [22] have apparently demonstrated the collection of services

166   demonstrated by community of researchers, developers, and scientists. Furthermore, middleware

167   solution utilizes an API that explains how services were accessed from both the applications and

168   the hardware that has been embedded for a Smart Grid-like deployment. Moreover, the authors

169   have indicated that boundaries exist between the hardware located and the network, which are

170   compliant with a standard demonstrating the sub-systems of the software. Vikash et al [23] have

171   conducted a survey on middle-wares of WSNs towards IoT in offering a comparative view of

172   different middle-wares and how this middleware technology can be utilized for implementing

173   several issues emerging in the development of IoT applications. Rodriguez-Molina et al [24]

174   have utilized different aspects with several functionalities preferred as essential for a semantic

175   middleware architecture conjoined with maritime operations including access to the application

176   layer, context awareness, and device and service registration. On the contrary, Rodriguez-Molina

177   et al [24] have interweaved other technologies with middleware such as acoustic networks and

178   wireless communications. Under such circumstances, Rodriguez-Molina et al [25] have

179   established an approach for interchanging information at the data level among independent

180   maritime vehicles, which is of significant importance as the required information will have to be

181   defined, along with the size of transferred data. Rodriguez-Molina et al [25] have forwarded the

182   Maritime Data Transfer Protocol for interchanging standardized aspects of information at the

183   data level for maritime independent maritime vehicles, and the procedures that are needed for

184   information interchange.

185

186   **2.2 Techniques/Protocols/Paradigms used in the Development of Proposed Work**

187   Figure 1 highlights the integration of the additional middleware layer within the control

188   system architecture using the 'Device Wrapper' node. Without this layer of middleware, it would

189   be necessary first to learn the use of interface and to control each smart device individually. This

190   requires the use of multiple different programming languages, such as Java, Python, C, C++ or

191   C#, and possibly OS-dependent software packages. For example, the University of Essex

192   currently has two full-scale intelligent environment test-beds, each using a different style of

193   additional middleware layer to amalgamate devices from various manufacturers. These include

194   iClassroom and iSpace.

195   [Insert Figure 1 Here]

196   The iClassroom [26] is an intelligent environment customised to resemble a university or

197 school teaching room (Figure 2). Most smart technologies are used directly to augment

198 presentations or other teaching strategies to enhance student learning experiences. The

199 middleware in the environment uses Web Services to wrap diverse collection of devices into a

200 single common interfacing mechanism. The Web Services system uses Eclipse's Jetty Web

201 Server [27] for its operations.

202 [Insert Figure 2 Here]

203 The iClassroom uses a centralized configuration, with all the middleware running from a

204 single server. In terms of functionality, the main user interface is in bespoke website form hosted

205 on the environment server. This server contains numerous hyperlinks to the Web Services used

206 to control each device. The user interface can be loaded onto any standard browser application

207 such as smart phones and tablets. Through the user interface, smart device in the environment

208 can be connected, controlled, and monitored to the Web Services system from any remote

209 location capable of accessing the iClassroom network.

210 [Insert Figure 3 Here]

211 The iSpace [28, 29] is an intelligent environment test-bed customized to resemble a typical

212 household environment. The multi-roomed space includes a full-sized lounge/kitchen, study,

213 bathroom, and bedroom, each connected by a central hallway. Unlike the iClassroom, most of

214 the smart technologies in the iSpace are deliberately concealed within hollow walls and ceilings

215 (Figure 3). This provides unknowing visitors to space an initial impression that they are in a

216 normal (i.e., non-augmented) environment. The middleware in this intelligent environment is

217 based upon UPnP, with controllers for over sixty smart devices linked together into a single API.

218 More specifically, Youpi UPnP stack [30] was used to implement wrappers for each smart

219 device.

220   The iSpace comprises a distributed configuration, with the middleware and control code for

221   devices which split across several different computers, each connected via a common network.

222   The Youpi UPnP wrappers allow each smart device to broadcast its existence on the same

223   network [31]. iSpace allows programmers to create a bespoke controller code to integrate the

224   smart devices into their projects better. To discover them, programmers must create a Youpi

225   UPnP control point within their code and perform a search. This can be either for a specific

226   device or a general search that returns a set of every smart device discovered on the network.

227   Using a returned smart device instance, a control program can then isolate specific state

228   variables, actions and arguments to either monitor or modify the associated hardware. Using

229   UPnP, user control programs can 'subscribe' to individual state variables within a smart device

230   instance. The state of a subscribed variable changes can automatically flag an associated listener

231   within the user's interface program.

232   For a large environment like the iSpace, implementation and maintenance of UPnP

233   middleware require significant programming effort. To operate a wrapper for each smart device

234   instance, a significant amount of information concerning hardware and its uses are required.

235   Initially, each device type needed every state variable, action, and argument that can be

236   individually declared. These were associated with dedicated UPnP state listeners to allow the

237   correct functioning of the subscription system. Finally, it was necessary to individually initialize

238   and start each instance of a smart device in the iSpace, which included assigning unique

239   attributes such as names and UUIDs.

240   **3. The Environment Development Kit (EDK)**

241   The new system is created from scratch and is not an extension or reconfiguration of a pre-

242   existing architecture. Thereby, steps should be taken to address the synchronisation and resource

243   allocation issues. This led to the creation of the Environment Development Kit (EDK), which

244   was written using only standard Java SDK, with no extensions or third-party APIs. The EDK

245   architecture design declared smart device present within an Intelligent Environment to be;

246   a) Discoverable on a network.

247   b) Subscribed to using listeners, which monitored for state change events.

248   c) Accessible and/or controllable via a common API.

249   d) Accessible and/or controllable via a set of auto-generated Web Services.

250   e) Assigned with a unique controller program instance or one collectively shared by a group.

251   The system operates in both a centralized or distributed context and is not OS dependent.

252   Controllers assigned to individual or groups of devices could be updated using a 'hot-swap',

253   without needing to restart any part of the system, (as long as the device is not being accessed at

254   the time of modification). Additionally, the inventory of devices is not to be declared before the

255   system could run, as new devices could be discovered and handled by the system at any time.

256   The case is similar to various middleware previously designed for intelligent environments, such

257   as UPnP.

258   A multicast communications system allows devices to be linked to agents and other user

259   programs. Multicast is less dependable than alternative communication methods such as TCP. It

260   is because there is no guarantee that intended targets would receive the message broadcasted to a

261   network. However, it is significantly much faster than TCP and can share message across

262   multiple targets in a subscribed group simultaneously, which determines the main reasons behind

263   its usage in this middleware system.

264   Similar multicast systems are in use in computer game virtual worlds, where the environment

265   is divided into regions, each with their assigned group containing all the players, objects and

266   other updatable information present in that area [32, 33]. When an entity moves from one region

267   to another, it should also change its subscription to the corresponding multicast group while

268   simultaneously leaving the previous one.

269   [Insert Figure 4 Here]

270   Users were able to access the EDK network, search for represented smart devices and send

271   action requests to specific discovered instances via an inbuilt control point. Devices discovered

272   by each control point instance were automatically 'cloned' and locally stored. Clone devices

273   were exact copies of their originals, with the exception that they possessed no controller, so they

274   could not directly interface with real hardware in the environment. Cloned devices received state

275   updates from the original 'master' device. These synchronisation updates occur regularly.

276   Typically, this requires an update several times a minute, although the interval period between

277   messages could be increased or decreased according based on its importance for clones to return

278   the most up-to-date state information possible.

279   For actuators, EDK control points bypass the clone representation and transmit a state change

280   action request directly to the master device. Multicast communication used received response

281   from the device indicating concerning the change of request wither its success or failure.

282   However, if the new state was applied to the master device, then alteration of the local clone

283   would occur during the next synchronisation update. To speed this process, the EDK was

284   designed where changes are automatically and immediately prompted to a synchronisation

285   update, temporarily overriding the scheduled system, (which had its internal timer reset). Figure

286   4 shows complete EDK system architecture for controlling a single, smart device via a third-

287   party interface program.

288   Allowing users to create control points, the API provides a series of expandable classes and

289   methods that are used with a broad range of bespoke sophistication for devices or environments.

290   The EDK came packaged with classes for representing several basic actuator and sensor types,

291   used by master device representations, and subsequently by control point instances when

292   creating their clones. Furthermore, the system was equally capable of handling encountering

293   unknown device types in an ad-hoc manner. This was achieved by creating a cloned instance of

294   the discovered device using a generic class appropriate for representing its declared features and

295   variables. Several different generic classes existed depending on the device, whether the device

296   was an actuator or a sensor and the number variables it possessed. Because of this feature,

297   control points did not require setup or inventory of an intelligent environment before being

298   connected. Additionally, it was possible to use the EDK with intelligent environments based

299   around both centralized and distributed architectures, as there was no requirement to host master

300   devices and clones on the same computer to communicate.

301   **3.1 Implementing Master Devices**

302   Three key components are required to exist on an EDK network;

303   1) A representation identifies the attributes of the device and declaring whether it is an actuator

304   or sensor.

305   2) A gateway to the target EDK network from where the device will be accessible.

306   3) A controller connecting the virtual EDK representation with some counterpart hardware that

307   exists in the real-world.

308   Together these components allow a "master" device to be created using the EDK API. Once

309   deployed on the network, the virtual device representation would monitor or change the state of

310   the real hardware-based upon received user instructions. It would also periodically broadcast

311   details of itself and the current state of the hardware to the EDK network, which is received by

312   any active control points and used to create or synchronize associated "clone" devices.

313   Master devices can be implemented and deployed remotely on one or several networked

314   computers. Depending upon personal preference, a master device application could also be used

315   to generate one or several different actuators and sensors, with no requirement of device type.

316   Multiple different control programs can be imported into the same implementation, and if

317   desired, a group of devices can either share or each is allocated their instance of the same

318   controller.

319

320   **3.1.1 Step One: Declaring Constants**

321   [Insert Figure 5 Here]

322   Firstly, several constants need to be declared to allow communication and different control

323   systems to operate correctly. Figure (5) provide examples of these, where each attribute is

324   explained further in the implementation process.

325

326   **3.1.2 Step Two: Creating Device Instances**

327   [Insert Figure 6 Here]

328   [Insert Figure 7 Here]

329   The EDK API contains several different classes that can be used to create new

330   representations of individual master devices. The selection of the class for a given situation

331   depends upon several factors, specifically, the number of state variables supported by the device

332   along with its identification as an actuator or a sensor. For common device types, the EDK API

333   contains several dedicated classes, many of which contain bespoke convenience methods that

334   provide better control of specific state variables. For example, Figures 6 and 7 show the creation

335   of an instance of two different types of light emitting devices. Both types have a state variable

336   "Power" declared, which controls whether the device is on or off. To support these actions, both

337   the "BooleanLight" and "DimmableLight" classes contain two convenience methods "turnOn"

338   and "turnoff" which allow the "Power" state variable to be set without the need to declare or

339   process any new values directly. The "DimmableLight" class used in Figure 7 additionally

340   contains "getBrightness" and "setBrightness" methods, as compared to the declared "Brightness"

341   state variable, used to control the emitted light level.

342   Without these convenience methods, the state variable name would need to be declared along

343   with any new state value (if applicable), in a formatted String in order to perform the same

344   function. If the device being used is uncommon or unknown for some reason, the EDK API

345   generic series classes can be created. This creates representations based purely on the number of

346   declared state variables. For actuators, these would be "SingleVariableActuator" and

347   "MultiVariableActuator", whereas for sensors the appropriate generic classes would be

348   "SingleVariableSensor" and "MultiVariableSensor". Each class contains several different

349   constructors based on the available information of the created device. For instance, in Figures 6

350   and 7, the constructor is provided with a name for the device instance, the supported state

351   variables (provided in a string array for multi-variable devices, as seen in Figure 7), and a

352   description of the device itself. In this instance, all other required variables, such as a unique

353   UUID, are allocated to the new device by the EDK API. Figures 6 and 7 also show how a

354   "StateChangeListener" can be attached to individual master device instances, which flag

355   whenever the value of any supported state variable is changed either by some attached hardware

356   or due to a user request.[1]

---

[1] Note: When implementing master devices, it is highly recommended that each instance is given unique name value. It should always be ensured that device instances always have different uuid values, which can be generated randomly using the Java SDK UUID class.

Note: All supported state variable names must be declared when creating a new device instance. Several pre-formed device types are included within the EDK API,

357 **3.1.3 Step Three: Adding Communications**

358 [Insert Figure 8 Here]

359 An instance of the "Communications" class must be created in order for master devices to be

360 able to connect to an EDK network. Figure 8 shows how to do this using two of the variables

361 from Figure 5 mentioned earlier to supply values for the network address and communications

362 port variables. The EDK uses multicast communication to allow master devices to send updates

363 to any running control points that have joined the same group. As a consequence, it is important

364 to ensure that the value used for the "NETWORK_ADDRESS" variable is a valid multicast

365 address. It may also be necessary to open the value used for "COMMUNICATIONS_PORT" on

366 firewalls, which may be blocking the sending or receiving of multicast communications packets

367 on the network.

368

369 **3.1.4 Step Four: Loading Device Controllers**

370 [Insert Figure 9 Here]

371 Control system programs for individual smart devices are created independently of the EDK.

372 Once implemented, a device control system can be uploaded into a master device via the EDK

373 API. Figure 9 shows how to do this. "DeviceController.jar" is the filename of the controller

374 being uploaded, from the designated "CONTROLLER_DIRECTORY". To integrate the

375 controller program with the master device, the EDK needs to know the package

376 ("CONTROLLER_PACKAGE") and the name of the main class ("CONTROLLER_CLASS").

377 As before, examples of these values are provided in Figure 5.

---

which often contain several convenience methods for performing device-specific actions. To use these methods, state variables must be declared using the specific names specified by the Javadoc information for the relevant class.

Note: If an application is intended to deploy multiple master device instances, it is recommended to create each instance at this stage before continuing.
Note: To prevent processing errors, individual state variables should never include the colon or tilde characters (i.e., : or ~) in their names or possible returnable values.

378

3.1.5 Step Five: Creating a Device Processing Hub

380   [Insert Figure 10 Here]

381   The next step is to create a hub to process devices. The "communications" variable of the

382   "DeviceHub" constructor should be the same instance of the "Communications" class created

383   back in Step Three. Once the hub is initialised, each of the smart device instances created in Step

384   Two needs to be individually added. As these are master devices, they also need to be associated

385   with their respective controller programs, loaded during Step Four. Figure 10 provides an

386   example of how to implement this for light devices.

387   The purpose of the "DeviceHub" class differs slightly depending upon whether it is used

388   within an implementation for master devices or an EDK control point. For master devices, the

389   hub liaises with the communication system created in Step Three, to have each of its stored

390   devices access their associated controllers and perform a live update of their recorded state

391   variables. Typically, this would involve each device accessing the real-world hardware to which

392   its control system connects it. Once acquired, the up-to-date state variable information is then

393   passed back to the communications system, where it is transmitted to the EDK network and used

394   synchronizing any listening clone device representations.

395

3.1.6 Step Six: Adding Web Services, (Optional)

397   [Insert Figure 11 Here]

398   Using the Web Services interface method with an EDK middleware implementation is

399   optional, so this step can be ignored if desired. However, enabling the Web Services system only

400   requires the code shown in Figure 11 to be added after the creation of the device hub. The

401   variable "WEBSERVICES_PORT" is the last of the declared constants create back in Step One,

402   which in the case of this example (Figure 5).

403   Once added to an instance of the "DeviceHub" class, the EDK mechanism will auto-generate

404   a Web Services control interface for each of the declared smart devices and add them to the

405   internal HTTP Server. Currently, two different acceptable commands are implemented for

406   sensors, (i.e., about and get), while three for actuators (i.e., about, get and set). The Web Services

407   interface can be loaded using any standard Web Browser, including on most mobile devices,

408   such as smartphones and tablet computers. The syntax for an EDK Web Service is naturally

409   bespoke to each situation where it is used, but the basic URL structure is as follows;

http://\<Computer IP Address>:\<Web Server Port>/\<Device Name>/\<Command>

410

411   So, based upon the "BooleanLight" and "DimmableLight" smart device examples used

412   throughout this tutorial, some acceptable URLs would be;

413

414   http://127.0.0.1:8000/Light1/about   http://127.0.0.1:8000/Light1/get

415   http://127.0.0.1:8000/Light1/set?Power:1   http://127.0.0.1:8000/Dimmer1/get?Brightness

416   http://127.0.0.1:8000/Dimmer1/set?Brightness:75

417

'127.0.0.1'	The IP address of the computer running the master device representation being accessed, (likely to be different from localhost).
'8000'	The port number for the Web Server, as declared by the value of the „WEBSERVICES_PORT" constant created back in Step One.
'Light1' 'Dimmer1'	The names of the smart devices being accessed, as declared when their representations were created in Step Two.
'about'	A command to display general information about the specified device.

'get'	A command to return the name of each state variable supported by the specified device, along with its currently recorded value.
'set?Power:1'	A command to set state variable „Power" to a value of „1".
'get?Brightness'	A command to return the current value of state variable "Brightness".
'set?Brightness:75'	A command to set state variable "Brightness" to a value of "75".

418

419 **3.1.7 Creating a Control Point**

420   [Insert Figure 12 Here]

421   An instance of the "ControlPoint" class allows external client programs to access devices via

422 an EDK network. Figure 12 shows the API code required to perform this task. The String

423 variable used in the "ControlPoint" constructor is a bespoke name for the specific instance being

424 created. If more than one control point is being used within a client program, (i.e., to access

425 different EDK networks), this variable can be used to identify specific instances.

426   A control point effectively acts as a portal into the EDK middleware system, allowing users

427 to search for groups or specific master devices on the associated network. Its associated

428 communications system provides details of the network which is accessed by a control point. In

429 addition to providing details of an EDK network, the communication system supplied to a

430 control point is also responsible for processing state update messages for the master devices. The

431 control point itself automatically generates an internal "DeviceHub" instance, which is used to

432 create and store clone device instances based on information received by the communications

433 system from the EDK network. The control point accesses the stored clone devices and uses their

434 information to provide returnable results for user searches.

435 **3.1.8 Searching for Devices**

436   [Insert Figure 13 Here]

437   To search for known master devices, present on an EDK network, a "ControlPoint" instance

438   can use its "searchForDevices" method (Figure 13). Instances of "ControlPoint" will only

439   become aware of master devices upon receiving an update packet from them. Therefore, upon

440   initially starting, the delay might occur before all master devices present on a network are

441   discovered, concerning their update cycles when the control point joins the EDK multicast

442   group. It is typically a good idea to enclose the search command in a "for", "do-while" or

443   "while" loop to keep the control point scanning the network until a non-empty array is returned

444   or the desired device is found. Alternatively, this command could also be repeatedly called from

445   within an isolated thread, which runs continuously in the background, allowing devices that start

446   broadcasting after the control point's initial search also to be detected.

447

448   **3.1.9 Searching for Specific Devices**

449   [Insert Figure 14 Here]

450   If details of the target device or devices are known ahead of time, a client program can

451   alternatively use one of the more specific search methods of the "ControlPoint" class. Figure 14

452   shows three such methods, each targeting different attributes of master devices. Firstly, the

453   "SearchForDevicesByType" method can be used to filter the known list of master devices by

454   their type, returning an array of any matching instances stored in the control point "DeviceHub".

455   The topmost example in Figure 14 uses this method to search for all instances of a

456   "BooleanLight" The "Actuator" and "Sensor" classes in the API both contain numerous other

457   declared variables that can be used with this method each representing one of the pre-formed

458   smart device types included in the EDK. Alternatively, to search for a device type not included

459   within the standard API, programs can use the "SINGLE_VARIABLE_DEVICE" and

460   "MULTI_VARIABLE_DEVICE" variables, (also found in the "Actuator" and „Sensor" classes),

461   or a bespoke device name entered as a String.

462   The two remaining search methods, shown in Figure 14, are each designed only to return a

463   single device, matching either a specified name or uuid criterion. If no matching device is found,

464   then a "null" value is returned. If, for some reason, two different master devices existed on an

465   EDK network and both were called "Light1", the "searchForDeviceByName" method will only

466   return the first instance it encounters upon contents scanning of control point device hub. This

467   also applies to the "searchForDeviceByUUID" method if both devices share the same aid value.[1]

468

469   **3.1.10 Processing EDK Smart Devices**

470   The instance of "Device Hub" created by a control point is used to store clones created to

471   represent networked master devices locally. The device hub creates the clones, which is the

472   automatic response when information is received from the communication system that does not

473   match any previously known representation. Clone devices are generally stored locally on the

474   same computer as the control point used to create them. Aside from not possessing controllers

475   for hardware, they are identical to the master devices that spawned them in every way. Even the

476   unique attributes of the original master device are copied, including its name and quid value. As

477   such, by using the associated classes within the EDK API, the details and states of each clone

478   can be read, and in the case of actuators manipulated, in the same manner as if connecting

479   directly to any master device instance.

480   [Insert Figure 15 Here]

481   The EDK API contains several different methods of reading the state of smart devices. A

---

[1] Note: When implementing master devices, it is highly recommended that each instance is given unique name value. It should always be ensured that device instances always have different uuid values, which can be generated randomly using the Java SDK UUID class.

482   selection of the possible options is listed in Figure 15. The determination of best method largely

483   depends upon what information is required about the state of a device and how it is subsequently

484   used.

485   From Figure 15 examples, the topmost method is a general "getState" command, which is

486   common to every EDK device. When called, the "getState" command will return a single string

487   representation of the entire device, or more specifically, its state variable values. Responses are

488   always sent in pairs, with the name of the state variable and its current value. For instance, in the

489   case of the "Light1" "BooleanLight" device used in the implementation examples earlier, an

490   "etState" request could result in either of the following String responses;

491

Power:0
Power:1

492

493   Where, "Power" is the name of the only state variable included in a "BooleanLight" object,

494   while zero (off) and one (on) are the current values of that state. The colon separating the two

495   values acts as a key in a split command to allow easy separation of variable name and its value.

496   In the case of multi-variable devices, which contain more than one state variable, an additional

497   tilde key ("~") is added to separate the individual attributes. So, for the "Dimmer1"

498   DimmableLight" used in earlier examples, a "getState" request could return;

Power:0~Brightness:100

499

500   Where, "Power" is the first state variable and "Brightness" is the second, with current

501   recorded values of zero (off) and one hundred (percentage of maximum illumination),

502   respectively[1].

---

[1] Note: To prevent processing errors individual state variables should never include the colon or tilde characters (i.e.: or ~) in their names or possible returnable values.

503   Figure 15 shows three methods that API can be used to return only the current value of a

504   specific state variable, rather than the name/value pairs shown above. Generally, this is achieved

505   by specifying the name of the state variable whose value is required as a variable in the method.

506   The second and third down examples in Figure 12 demonstrate this process for a "BooleanLight"

507   and "DimmableLight" respectively. Typically, the returned state values are in a String format,

508   but can easily be converted as shown with the "Brightness" variable example, where the value is

509   converted into an integer once returned.

510   In many cases, the string can be avoided to integer conversion, as performed in the third

511   example of Figure 15. This is based on EDK API device's inclusion of convenience methods,

512   which return state variable values in their most appropriate format automatically. This is

513   demonstrated by the bottommost method in Figure 15, which uses a "getBrightness" method

514   found in the "DimmableLight" class, which automatically returns the state value as an integer.

515   All that is required to use these bespoke convenience methods is to cast the generic "Device"

516   object returned by a control point search into the appropriate device type class, as is shown in the

517   example.

518

519   **3.1.11 Writing to Smart Devices**

520   [Insert Figure 16 Here]

521   In the example provided by Figure 16, an array returned by a control point search (as

522   described in Section 2.2), is scanned for a specific device called "Dimmer1", which is also a

523   "DimmableLight". If found, the value of the state variable "Power" is requested from the device.

524   If the value returned for "Power" is zero (off) then a command is sent to turn the light on. For

525 any other circumstance, the command is to turn the light off.[1]

526 In the example, the generic "device" variable taken from the "Device" array is cast into a

527 "DimmableLight" object. This allows the programmer to access the two convenience methods

528 "turnoff" and "turnOn," which are contained within the Actuator class. The EDK API contains

529 several models for intelligent devices that can be used in place of the more generic actuator and

530 sensor classes. Some of these classes also contain further convenience methods specific to that

531 device type. For example, the "DimmableLight" class also contains a "setBrightness" method to

532 allow a specific value to be entered for a "Brightness" state variable, controlling the amount of

533 light emitted by the device.

534

535 **3.1.12 Implementing EDK Compatible Device Controllers**

536   [Insert Figure 17 Here]

537 Recalling back to Section 3.1.1, two of the constants that needed to be declared when

538 implementing a master device (as shown in Figure 5) were "CONTROLLER_CLASS" and

539 "CONTROLLER_PACKAGE". The origins of the values used in the Figure 5 example, can be

540 seen in Figure 17. The value that should be used for the "CONTROLLER_CLASS" constant is

541 the name of the main class of the control system program, which in the example is simply

542 "Controller". Additionally, the value for the "CONTROLLER_PACKAGE" is the name of the

543 package containing the declared main class, in this case "devicecontroller".

544 When implementing a control program for smart devices to be used with an EDK

545 middleware system, Figure 17 shows the minimum classes required for integration. More

546 specifically, the "getState" and "setState" methods are both essential and should be used to

---

[1] Note: In the example provided in Figure 16, "Actuator.DIMMABLE_LIGHT" could also have been used instead of the String value "DimmableLight."

547 directly return or update the current state of the associated hardware, respectively. If additional

548 code is required to create a link with the associated hardware, such as using a third-party

549 software package (e.g., RXTX or a manufacturer API), then all this code should be placed into a

550 constructor within the main class, as shown by the "Controller" constructor in Figure 17.[1] If

551 necessary, the constructor code should establish a connection with the smart device hardware and

552 then maintain it as a global variable that can be accessed directly by the "getState" and "setState"

553 methods. Alternatively, the constructor could be used to start an isolated thread, which uses

554 locally declared variables, also accessible by the "getState" and "setState" methods, to handle

555 state action requests. It is essential that no code necessary to directly establish a connection with

556 hardware is included in either the "getState" or "setState" method as doing so could lead to

557 overall instability in the EDK middleware system.

558

559 **3.1.13 Controllers for Multi-Variable Devices**

560   [Insert Figure 18 Here]

561   Figure 18 highlights how the controller code in Figure 17 can be extended to handle smart

562 devices with multiple state variables. The code presented (in Figure 17), could function with a

563 multi-variable smart device, but it is often desirable to separate certain state variables to better

564 structure program code, or for efficiency, etc. Figure 18 shows a template for the controller used

565 with the "DimmableLight" device "Dimmer1" example mentioned throughout this guide. Notice

566 how the "getState" and "setState" methods have been retained (for handling the "Power" state

567 variable), although the "Brightness" state variable has been separated and given its handling

568 methods, namely "getBrightness" and "setBrightness." To add additional "get" and "set"

---

[1] Note: Methods used in device controllers must return state values in as a single String using the expected EDK formats, as discussed in Section 3.3.1.
    Note: It is recommended that the "setState" method returns the same value as an action request to "getState" once completing its operation.

569   methods, it is necessary to ensure that their suffix is named the same as the state variable they are

570   expected to handle, (e.g. "getPower", "getBrightness", "setPower", "setBrightness", etc.).[1]

571   When accessing a loaded device controller, an EDK implementation will first search through

572   the methods of the declared controller class (i.e. "CONTROLLER_CLASS") to see if it contains

573   a bespoke match for the current state variable it needs to process. If no appropriate method

574   matching the state variable can be found, the system will then automatically default to either the

575   "getState" or "setState" method, depending upon which action is being performed.

576   Additionally, in Figure 15, the package name has changed, reflecting that it is a different

577   device controller program from Figure 14 example. To be loaded correctly into the EDK and

578   used with a master device, Figure 15 example would need to specify "devicecontrollermultiple"

579   as the value for "CONTROLLER_PACKAGE" during the first implementation step.

580

581   **4. Evaluation**

582   **4.1 Evaluation Strategy**

583   One of the criteria used to measure the success of the EDK is its effective processing of

584   action requests for smart devices, compared to existing middleware solutions. Since UPnP and

585   Web Services largely inspired the new mechanism, examples of these architectures are used as a

586   benchmark. Additionally, EDK testing with sensors and actuators is necessary, to acquire

587   accurate results for both sending and receiving state variable data. Therefore, the middleware

588   was integrated into the control systems of two different intelligent environments, located at the

589   University of Essex, namely the iClassroom and the iSpace test-beds.

590   The intelligent environments chosen for the evaluation were selected as benchmarks to take

---

[1] Note: Any "set" methods should also only expect to receive a single variable containing the new state value to be processed, as shown in Figures 17 and 18. No other variables must be added for the methods to be compatible.
    Note: As with "setState" it is recommended that any "set" method returns the same value as an action request to "getState" once completing its operation.

591   advantage of the established middleware in their existing control systems. During the evaluation,

592   EDK and benchmark middleware systems were tested by controlling a DMX-based dimmable

593   spotlight and a Phidgets light sensor. Primarily, it assesses the average processing speed required

594   by the different components of the EDK, (i.e., API control and Web Services) compared to that

595   needed by the equivalent benchmark systems. If the EDK was capable of performing the same

596   tasks as an existing middleware system in approximately the same time frame, it is considered a

597   viable alternative for use in an intelligent environment control system. Incidentally, it was

598   anticipated that the EDK would actually require a significantly lower processing time than UPnP

599   or Web Services, mostly due to the greater emphasis on locally processed variables.

600

601   **4.2 Evaluation Results**

602   To evaluate the EDK middleware system, each mechanism was tested separately, with no

603   other programs running at the time. For each middleware system, five hundred 'get' or 'set'

604   requests were made to the same sensor or actuator, respectively. This workload was split across

605   five sessions of one hundred requests each. To keep the test fair, each session was performed

606   using the same computer, which ran both timer program measuring the processing intervals, plus

607   made the state requests to the relevant middleware implementation.

608

609   **4.2.1 System Validation: Sensors**

610   The first experiment in the project evaluation focused specifically on comparing middleware

611   used to access sensors. In other words, devices whose functionality consisted only of sending

612   information about their current state back to a requesting program. The Jetty-based Web Services

613   system in the iClassroom linked directly into individual device control programs, which

614   automatically requested a current reading from the relevant sensor hardware when called.

615   However, the iSpace UPnP mechanism functioned in a very similar style to the EDK middleware

616   design. Specifically, whenever called, the 'getState' command would access the control system

617   for a specific device and return the last recorded value obtained from the actual sensor itself. A

618   'getState' method call never actually resulted in the hardware being accessed directly in order to

619   take a reading. Updates to the recorded light level value were handled by an independent thread

620   located within the program code for the sensor, which automatically took a new reading from the

621   hardware intermittently. In the case of the iSpace UPnP system, a new reading was recorded

622   approximately once every ten seconds. For EDK, this interval was reduced to approximately

623   three seconds, in order to reduce the margin of error between the recorded and actual sensor

624   values.

625   [Insert Table 1 Here]

626   The experiment involved measuring the time required for each middleware system to return a

627   lux value for a specific light sensor Phidget. As can be seen from the results displayed in Table 1,

628   on average for each session run, both device interface methods offered by the EDK middleware

629   required significantly less time to process individual action requests than both the Jetty-based

630   Web Services or Youpi UPnP implementations. In the case of the Direct API interface method,

631   the processing time was reduced to nanoseconds as the sensor value being returned in response

632   to each action request was taken from a locally stored variable within the clone of the real sensor

633   generated by the EDK. As mentioned above, the Youpi UPnP middleware used a similar method,

634   but the recorded values for the light sensor were stored on a remote server, hence required

635   additional processing time for the system to access and get the data values from. Unlike the API

636   interface method, action requests made using the EDK Web Services were directed at the real

637   light sensor, yet they still required a fraction of the processing time of the Jetty-based system,

638   consistently throughout the evaluation.

639

640   **4.2.2 System Validation: Actuators**

641   The second experiment performed for the evaluation focused on actuators. More specifically,

642   this involved devices that contained one or more variables that could have their state modified,

643   via the middleware accessing an attached control system. When setting the state of a variable, the

644   success of the corresponding hardware being updated was not always guaranteed. This is because

645   many devices do not provide any feedback to the controller code, concerning whether a

646   transmitted command has been received or acted. Both, Jetty-based Web Services and Youpi

647   UPnP middleware sent acknowledgments back to the control programs where action requests

648   originated but always assumed a positive outcome. In contrast, the EDK did not send any

649   acknowledgments in direct response to individual state change requests. The success or failure of

650   an update could be determined by observing the next set of variable settings transmitted by a real

651   device to its clones.

652   [Insert Table 2 Here]

653   This experiment involved each middleware system accessing a single DMX-controlled

654   dimmable spotlight and alternating its brightness level between fifty and one hundred percent.

655   Table 2 lists the average processing times required by each batch of one hundred action requests

656   made during the experiment. As with the sensor experimentation, both EDK interface

657   mechanisms were found on average to require significantly less processing times than the Jetty-

658   based Web Services and Youpi UPnP systems. It is worth noting that the difference between the

659   EDK and Jetty-based Web Services was not as profound as for the sensors. This was due to the

660 controller code for the DMX lights adding a delay of approximately one hundred milliseconds to

661 the processing of each action request via a compulsory sleep command. As the EDK Web

662 Services were directed at the representation of the actual light rather than a local clone, the delay

663 was unavoidable.

664 Unlike for the sensor 'getState' action requests, when setting a device's state, the EDK's

665 Direct API interface method also targeted the original 'master' representation rather than a clone.

666 Effectively, a state change action request was forwarded to the original device by sending out a

667 state change request using the inbuilt multicast communication system. As the system did not

668 require any acknowledgment, the process could be ended at that point, unlike UPnP and the Web

669 Services mechanisms, which required some kind of response, even if based upon an assumption.

670

671 **4.3 Evaluation Summary**

672 Concerning the processing times for action requests, on average, in every session of the

673 evaluation, both interface mechanisms of the EDK were found to be significantly superior to the

674 benchmark Jetty-based Web Services and Youpi UPnP middleware systems. This result was not

675 entirely unexpected as the EDK utilised locally stored variables much more than either of the

676 benchmark systems. Furthermore, in cases where remote access is necessary, (i.e. when setting

677 the state of an actuator), the EDK does not require any confirmation of a device receiving an

678 action state request to be returned, further reducing the processing times.

679 For experiment two, the additional 'power' state variable of the DMX spotlight was manually

680 set to an 'on' state prior to each experiment session to observe the brightness changes but was

681 otherwise unused. However, it should be noted that for each action request sent to the UPnP

682 middleware system, it was necessary for its control point to first perform a search of all the DMX

683   light state variables in order to isolate the one controlling the brightness before the new state

684   could be set. As more than one state variable existed within the device (i.e., power and

685   brightness), this may have caused the action request processing time to be extended, depending

686   upon which was discovered first. This additional step was not necessary for the EDK or Jetty-

687   based Web Services, where the individual state variables were already declared within the device

688   controller code. This may explain why the UPnP mechanism required more time than any other

689   mechanism in each session when handling an actuator, despite being faster than Jetty-based Web

690   Services in three out of five sessions during the earlier sensor experiment.

691   [Insert Figure 19 Here]

692   [Insert Figure 20 Here]

693   Finally, another observation made during the evaluation was that for each batch of one

694   hundred action requests performed, while the average processing times generated by Jetty-based

695   Web Services and Youpi UPnP fluctuated noticeably between sessions, the times for both EDK

696   methods remained largely static. This effect is illustrated by the graphical representations of the

697   results from the evaluation presented in Figures 19 and 20. These figures also clearly highlight

698   how the EDK was capable of outperforming the two benchmark middleware mechanisms

699   consistently throughout the entire evaluation.

700

701   **5. Conclusions**

702   This paper discussed recent work performed into creating a new middleware architecture for

703   an intelligent environment control system. It shared insights on how control system developers

704   often require some functionality trade-off when integrating one of the numerous existing

705   architectures. Alternatively, an existing mechanism may be manually augmented with additional

706 code to overcome any missing functionality, which could result in detrimental synchronization

707 issues within the resulting control system. This project aimed to create a new middleware,

708 capable combining the positive features of two well-known architectures (namely UPnP and

709 Web Services) into a single mechanism, to overcome functionality and synchronisation issues.

710 Most importantly, it envisaged that the new mechanism would be required to process state action

711 requests as quickly as the two benchmark middleware architectures.

712 Complex, intelligent environment developers must implement their additional bespoke layer

713 of middleware to link devices and services provided by different manufacturers. iClassroom and

714 iSpace control system architectures in two full-scale intelligent environments are also presented.

715 It also discusses the Environment Development Kit (EDK) as the new alternative middleware as

716 an outcome of the research performed by this project. It assesses how viable the new middleware

717 design was as an alternative to UPnP and Web Services. This strategy involved two separate

718 experiments testing sensors and actuators independently due to their different properties. It

719 showed that a comparative analysis of the average action request processing times required by

720 the EDK, plus implementations of the benchmark UPnP and Web Services systems.

721 The evaluation experiments both indicated overwhelmingly that the two interface methods

722 offered by the EDK architecture implementation could process state action requests to sensors

723 and actuators faster than either benchmark mechanism. Out of the ten sessions performed, either

724 benchmark did not outperform the EDK in a single case. EDK and UPnP used stored variables

725 when returning sensor state data, rather than accessing the actual device directly. These stored

726 variables were updated periodically by an independent thread, which noted the actual sensor

727 readings. However, in the UPnP implementation, a ten-second delay between individual readings

728 was observed along with an update of the stored variable. For the EDK implementation, this

729   delay was reduced to three seconds, yet the mechanism was still capable of processing action

730   requests faster than the UPnP benchmark. The results strongly indicated that the EDK is a viable

731   alternative to UPnP and Web Services in a smart device or intelligent environment control

732   system.

733   Another important issue for EDK was to resolve the data de-synchronised using multiple

734   device interfacing methods. As the new mechanism was designed from the ground up, steps

735   could be taken to allow synchronisation between the API and Web Services interface methods to

736   be practically guaranteed. For this, both interface methods use the same sets of variables to store

737   descriptions of individual master or clone devices. Also, clone devices were regularly

738   synchronised with their original master; locally stored representations were constantly being

739   updated with the known true state values. This helped defense against de-synchronization of data

740   stored in clone devices.

741

742   **5.1 Future Work**

743   With the initial framework for the EDK established, future researches can expand the

744   existing middleware architecture to integrate additional functionality. Firstly, an internal

745   mechanism can be introduced to add an internal authentication system, which allows instances of

746   smart devices represented by the mechanism to optionally be assigned with a password. Unless

747   the correct password was supplied during communication, neither control points nor other

748   interfaces could access the device, nor a clone of the original be created as a result of a search

749   request or update processes.

750   Therefore, future study should focus on EDK development, where its variables and methods

751   allow control points and other interfaces to identify the specific environment where an instance

752   of a smart device is located. Technically this could already be achieved to some extent using the

753   existing mechanism, for example, by manipulating the name variable when creating device

754   instances. However, with the augmentations would allow the handling of scenarios such as the

755   device changing locations to a different intelligent environment. It would also make it easier to

756   represent multiple different environments on the same network. This could potentially allow the

757   creation of agents or control programs that use devices from several different environments

758   collaboratively, as part of their functionality.

759

760   **Acknowledgement**

761   The author is highly grateful for all the associated personnel who contributed in the completion

762   of this Scale Up Project. The author declares that the study is not funded through any source,

763   neither it holds conflicts of interest.

764

References

1. S. Tan, J. Zhang, F. Sun, and S. Wang, an approach to support the interoperability of intelligent grouping and resource sharing (IGRS) and universal plug and play (UPnP) in home network environment. In 2015 IEEE International Conference on Computational Intelligence & Communication Technology, (2015), 682-686. IEEE.

2. A. Rafique, D. Van Landuyt, B. Lagaisse and W. Joosen, On the performance impact of data access middleware for nosql data stores a study of the trade-off between performance and migration cost, *IEEE Transactions on Cloud Computing* **6**(3) (2015), 843-856. https://doi.org/10.1109/tcc.2015.2511756

3. D. Bruneo, A. Puliafito and M. Scarpa, *Mobile Middleware, The Handbook of Mobile Middleware*, (2006), 145–166. https://doi.org/10.1201/9781420013153.ch7

4. B. Yuan, Z. Xiaolei and W. Yitao, Identifying Vehicle's Steer Change via Commercial Smartphones. *In 2019 International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS)*, (2019), 181-185. IEEE. doi:10.1109/hpbdis.2019.8735446.

5. J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, Internet of Things (IoT): A vision, architectural elements, and future directions, *Future Generation Computer Systems*, **29**(7) (2013), 1645–1660. https://doi.org/10.1016/j.future.2013.01.010

6. Z.A. Almusaylim and Z. Noor, A review on smart home present state and challenges: linked to context-awareness internet of things (IoT), *Wireless Networks.* **25**(6) (2019), 3193-3204. https://doi.org/10.1007/s11276-018-1712-5

7. A. Farahzadi, S. Pooyan, R. Javad and F. Reza, Middleware technologies for cloud of things: a survey. *Digital Communications and Networks (DCN)* **4**(3) (2018), 176-188.

787 8. A. H. H. Ngu, M. Gutierrez, V. Metsis, S. Nepal, and M. Z. Sheng, "IoT Middleware: A

788   Survey on Issues and Enabling technologies," *IEEE Internet of Things Journal*, pp. 1–1,

789   2016. Doi: https://doi.org/10.1109/jiot.2016.2615180

790 9. P. Fremantle and P. Scott, "A survey of secure middleware for the Internet of Things," *PeerJ*

791   *Computer Science*, vol. 3, p. e114, May 2017. Doi: https://doi.org/10.7717/peerj-cs.114

792 10. M. Elkhodr, S. Shahrestani and H. Cheung, "A middleware for the internet of things," arXiv

793   preprint arXiv:1604.04823. 2016 Apr 17. Doi: 10.5121/ijcnc.2016.8214

794 11. T. Jyothi. Open Source Middleware for Internet of Things, *International Journal of*

795   *Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering*.

796   2016. Vol. 4, Issue. 11. 71-75. Doi: 10.17148/IJIREEICE.2016.41113

797 12. M. A. Razzaque, M. Milojevic-Jevric, A. Palade and S. Clarke, Middleware for internet of

798   things: a survey, IEEE *Internet of things journal*, **3**(1) (2015), 70-95.

799   https://doi.org/10.1109/jiot.2015.2498900

800 13. S. Jeon and I. Jung, "MinT: Middleware for Cooperative Interaction of Things," *Sensors*, vol.

801   17, no. 6, p. 1452, Jun. 2017. Doi: https://doi.org/10.3390/s17061452

802 14. M. A. A. da Cruz, J. J. P. C. Rodrigues, J. Al-Muhtadi, V. V. Korotaev, and V. H. C. de

803   Albuquerque, "A Reference Model for Internet of Things Middleware," *IEEE Internet of*

804   *Things Journal*, vol. 5, no. 2, pp. 871–883, Apr. 2018. Doi:

805   https://doi.org/10.1109/jiot.2018.2796561

806 15. R. Zhao, L. Wang, X. Zhang, Y. Zhang, L. Wang, and H. Peng, "A OneM2M-Compliant

807   Stacked Middleware Promoting IoT Research and Development," *IEEE Access*, vol. 6, pp.

808   63546–63559, 2018. Doi: https://doi.org/10.1109/access.2018.2876197

809   16. J. Rivera and R. van der Meulen, Gartner says the internet of things installed base will grow
810   to 26 billion units by 2020. Stamford, conn., (2013).

811   17. C. Perera, P.P. Jayaraman, A. Zaslavsky, P. Christen and D. Georgakopoulos, Mosden: An
812   internet of things middleware for resource constrained mobile devices. *In 2014 47th Hawaii*
813   *International   Conference   on   System   Sciences,*   IEEE.   (2014).   1053-1062.
814   https://doi.org/10.1109/hicss.2014.137

815   18. M. Bazzani, D. Conzon, A. Scalera, M.A. Spirito and C.I. Trainito, Enabling the IoT
816   paradigm in e-health solutions through the VIRTUS middleware. *In 2012 IEEE 11th*
817   *international conference on trust, security and privacy in computing and communications,*
818   (2012), 1954-1959). IEEE. https://doi.org/10.1109/trustcom.2012.

819   19. Echelon,   Available   at: "http://www.echelon.com/products/i.lon-600-lonworks-ip-852-
820   router", 2015.

821   20. Phidgets, Available at: "http://www.phidgets.com", 2015.

822   21. Mohamed N, Al-Jaroodi J, Jawhar I. A review of middleware for networked robots.
823   International Journal of Computer Science and Network Security. 2009 May 30;9(5):139-48.

824   22. J. Rodriguez-Molina and D. M. Kammen, "Middleware Architectures for the Smart Grid: A
825   Survey on the State-of-the-Art, Taxonomy and Main Open Issues," IEEE Communications
826   Surveys & Tutorials, vol. 20, no. 4, pp. 2992–3033, 2018.

827   23. Vikash, L. Mishra, and S. Varma, "Middleware Technologies for Smart Wireless Sensor
828   Networks towards Internet of Things: A Comparative Review," Wireless Personal
829   Communications, Aug. 2020.

830	24. J. Rodríguez-Molina, S. Bilbao, B. Martínez, M. Frasheri, and B. Cürüklü, "An Optimized,

831	 Data Distribution Service-Based Solution for Reliable Data Exchange Among Autonomous

832	 Underwater Vehicles," Sensors, vol. 17, no. 8, p. 1802, Aug. 2017.

833	25. J. Rodríguez-Molina, B. Martínez, S. Bilbao, and T. Martín-Wanton, "Maritime Data

834	 Transfer Protocol (MDTP): A Proposal for a Data Transmission Protocol in Resource-

835	 Constrained Underwater Environments Involving Cyber-Physical Systems," Sensors, vol. 17,

836	 no. 6, p. 1330, Jun. 2017.

837	26. M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R.H. Campbell and K. Nahrstedt, Gaia: a

838	 middleware platform for active spaces. *ACM SIGMOBILE Mobile Computing and*

839	 *Communications Review*, **6**(4) (2002), 65-67.

840	27. L. Roalter, M. Kranz, and A. Möller, A middleware for intelligent environments and the

841	 internet of things. In International Conference on Ubiquitous Intelligence and Computing,

842	 Springer, Berlin, Heidelberg. 267-281.

843	28. F. Lopes, F.C. Delicato, T. Batista, E. Cavalcante, T. Pereira, P.F. Pires, P. Ferreira, R.

844	 Mendes, OpenCOPI: middleware integration for Ubiquitous Computing. International

845	 *Journal of Parallel, Emergent and Distributed Systems,* **29**(2) (2014), 178-212.

846	 https://doi.org/10.1080/17445760.2013.831415

847	29. T.G. Stavropoulos, K. Gottis, D. Vrakas and I. Vlahavas, aWESoME: A web service

848	 middleware for ambient intelligence. *Expert Systems with Applications*, **40**(11) (2013), 4380-

849	 4392.

850	30. A. Olaru, A.M. Florea, and A.E.F. Seghrouchni, A context-aware multi-agent system as a

851	 middleware for ambient intelligence. Mobile Networks and Applications, **18**(3) (2013), 429-

852	 443. https://doi.org/10.1007/s11036-012-0408-9

853   31. J. Dooley, M. M. Henson, V. Callaghan, H. Hagras, D. Al-Ghazzawi, A. Malibari A, M. Al-

854   Haddad M, and A.A. Al-Ghamdi, A formal model for space based ubiquitous computing. *In*

855   *2011 Seventh International Conference on Intelligent Environments*, (2011), 74-79. IEEE.

856   https://doi.org/10.1109/ie.2011.63

857   32. J.-E. Simó-Ten, E. Munera, J.-L. Poza-Lujan, J.-L. Posadas-Yagüe, and F. Blanes,

858   "CKMultipeer: Connecting Devices Without Caring about the Network," *Advances in*

859   *Intelligent Systems and Computing*, (2017), 189–196.

860   33. J. Eclipse, 2015, http://www.eclipse.org/jetty, retrieved: June 2015.

861

862   **Figure Legends**

863   **Figure 1:** An example of a two-tiered middleware architecture for controlling a smart device

864   **Figure 2:** The iClassroom intelligent environment and its middleware architecture

865   **Figure 3:** The iSpace intelligent environment and its middleware architecture

866   **Figure 4:** The EDK middleware architecture

867   **Figure 5:** An Example of Communication and Controller Constants.

868   **Figure 6:** Creating a New Single Variable Smart Device

869   **Figure 7:** Creating a New Multi-Variable Smart Device

870   **Figure 8:** Creating the Communications System

871   **Figure 9:** Loading device control systems

872   **Figure 10:** Creating a Hub and Adding Master Devices

873   **Figure 11:** Enabling the EDK Web Services

874   **Figure 12:** Code for creating an EDK Control Point instance

875   **Figure 13:** Searching for Known Master Devices on an EDK Network

876   **Figure 14:** Three Methods for Finding Specific Devices

877   **Figure 15:** Methods for Reading the Values of Device State Variables

878   **Figure 16:** An example of sending action state change requests to a master device.

879   **Figure 17:** A Controller Template for a Single Variable Smart Device

880   **Figure 18:** A Controller Template for a Dimmable Light, (multi-variable smart device)

881   **Figure 19:** A graph showing average processing times for a sensor request (experiment one)

882   **Figure 20:** A graph showing average processing times for setting a light (experiment two)

Average middleware processing times for taking a reading from a light sensor

1   **Table 1:** Average middleware processing times for taking a reading from a light sensor

	Average Processing Time Required (milliseconds)				
	Session 1	Session 2	Session 3	Session 4	Session 5
Jetty-based Web Services	80.03	77.43	55.58	61.48	60.65
Youpi UPnP	52.51	35.51	99	22.01	106.56
Web Services (EDK)	1.99	1.66	1.61	1.64	1.40
Direct API Control (EDK)	0.00542	0.00532	0.00516	0.00517	0.00519

Average middleware processing times changing the brightness of a dimmable light

1   **Table 2:** Average middleware processing times changing the brightness of a dimmable light

	Average Processing Time Required (milliseconds)				
	Session 1	**Session 2**	**Session 3**	**Session 4**	**Session 5**
Jetty-based Web Services	127.09	133.46	112.57	113.92	118.29
Youpi UPnP	171.36	232.33	212.22	238.08	180.49
Web Services (EDK)	101.83	102.45	102.35	102.26	102.09
Direct API (EDK)	0.67	0.64	0.60	0.59	0.55

Figure 1

Figure1: An example of a two-tiered middleware architecture for controlling a smart device

Figure1: An example of a two-tiered middleware architecture for controlling a smart device
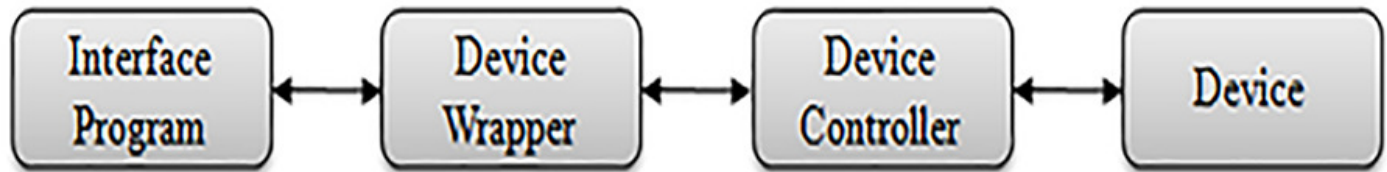
Figure 2

Figure 2: The iClassroom intelligent environment and its middleware architecture

Figure 2: The iClassroom intelligent environment and its middleware architecture
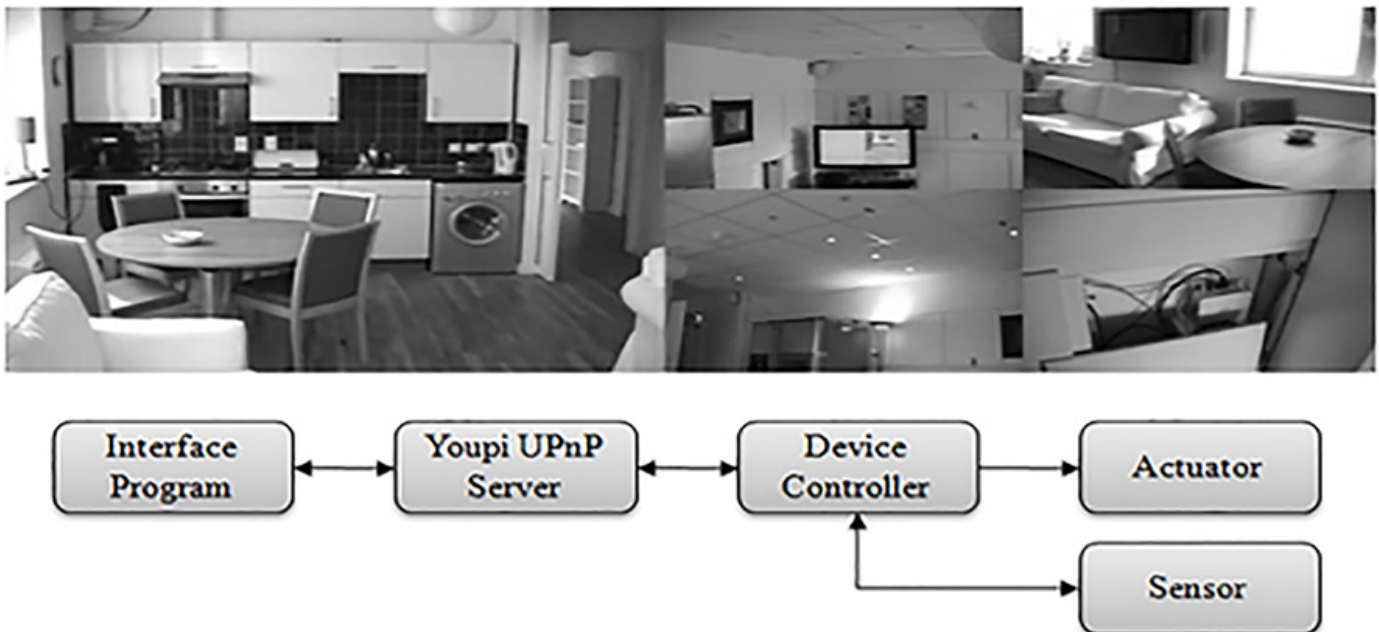
Figure 3

Figure 3: The iSpace intelligent environment and its middleware architecture

Figure 3: The iSpace intelligent environment and its middleware architecture
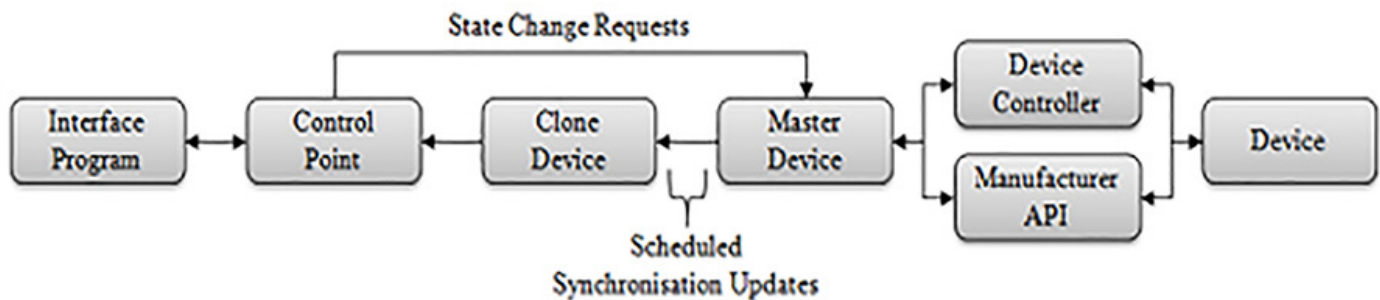
Figure 4

Figure 4: The EDK middleware architecture

Figure 4: The EDK middleware architecture

```
private final int COMMUNICATIONS_PORT = 4446;
private final int WEBSERVICES_PORT = 8000;
private final String CONTROLLER_CLASS = "Controller";
private final String CONTROLLER_PACKAGE = "devicecontroller";
private final String CONTROLLER_DIRECTORY = "controllers";
private final String NETWORK_ADDRESS = "230.0.0.1";
```

Figure 5

Figure 5: An Example of Communication and Controller Constants

```
BooleanLight light1 =
        new BooleanLight("Light1", "Power", "Light Mk 1");
light1.addStateChangeListener(sl);
```

Figure 6

Figure 6: Creating a New Single Variable Smart Device

Figure 6: Creating a New Single Variable Smart Device

```
DimmableLight dimmer1 = new DimmableLight("Dimmer1",
        new String[] { "Power", "Brightness" }, "Dimmer Light Mk 1");
dimmer1.addStateChangeListener(sl);
```

Figure 7

Figure 7: Creating a New Multi-Variable Smart Device

Figure 7: Creating a New Multi-Variable Smart Device

```
DeviceHub lightHub = new DeviceHub(communications);
lightHub.addDevice(light1, lightController);
lightHub.addDevice(dimmer1, dimmerController);
```

Figure 8

Figure 8: Creating the Communications System

```java
InetAddress environmentAddress =
        InetAddress.getByName(NETWORK_ADDRESS);
Communications communications =
        new Communications(environmentAddress, COMMUNICATIONS_PORT);
```

Figure 9

Figure 9: Loading device control systems

Figure 9: Loading device control systems

```java
File controllerJARFile = new File(CONTROLLER_DIRECTORY +
        File.separator + "DeviceController.jar");
Controller lightController = new Controller(
        controllerJARFile, CONTROLLER_PACKAGE, CONTROLLER_CLASS);
```

Figure 10

Figure 10: Creating a Hub and Adding Master Devices

Figure 10: Creating a Hub and Adding Master Devices

```
DeviceHub lightHub = new DeviceHub(communications);
lightHub.addDevice(light1, lightController);
lightHub.addDevice(dimmer1, dimmerController);
```

Figure 11

Figure 11: Enabling the EDK Web Services

```
Webserver webServer = new Webserver(WEBSERVICES_PORT);
lightHub.addWebServices(webServer);
```

Figure 12

Figure 12: Code for creating an EDK Control Point instance

Figure 12: Code for creating an EDK Control Point instance

```
InetAddress environmentAddress =
        InetAddress.getByName(NETWORK_ADDRESS);
Communications communications =
        new Communications(environmentAddress, COMMUNICATIONS_PORT);

ControlPoint controlPoint =
        new ControlPoint(communications, "ControlPoint1");
```

Figure 13

Figure 13: Searching for Known Master Devices on an EDK Network

Figure 13: Searching for Known Master Devices on an EDK Network

```
Device[] deviceList = controlPoint.searchForDevices();
```

Figure 14

Figure 14: Three Methods for Finding Specific Devices

Figure 14: Three Methods for Finding Specific Devices

```
// Search for all instances of a specific device type
Device[] deviceList = controlPoint.
        searchForDevicesByType(Actuator.BOOLEAN_LIGHT);

// Search for a single device with the specified name
Device device = controlPoint.searchForDeviceByName("Light1");

// Search for a single device with the specified UUID
UUID uuid =
        UUID.fromString("2deb19fc-5d41-4ac9-a0ca-4e86ff7a121e");
Device device = controlPoint.searchForDeviceByUUID(uuid);
```

Figure 15

Figure 15: Methods for Reading the Values of Device State Variables

Figure 15: Methods for Reading the Values of Device State Variables

```
String deviceState = device.getState();

String powerState = device.getState("Power");

int brightnessState =
        Integer.parseInt(device.getState("Brightness"));

int brightnessState =
        ((DimmableLight)device).getBrightness();
```

Figure 16

Figure 16: An example of sending action state change requests to a master device

```java
for (Device device : deviceList) {
    if (device.getName().equalsIgnoreCase("Dimmer1") &&
            device.getType().equals("DimmableLight")) {
        if (device.getState("Power").equalsIgnoreCase("0")) {

            ((DimmableLight)device).turnOn();

        }

        else {

            ((DimmableLight)device).turnOff();

        }
    }
}
```

Figure 17

Figure 17: A Controller Template for a Single Variable Smart Device

Figure 17: A Controller Template for a Single Variable Smart Device

```
package devicecontroller;

public class Controller {

    public Controller() {

    }

    public String getState() {

    }

    public String setState(String state) {

    }
}
```

Figure 18

Figure 18: A Controller Template for a Dimmable Light, (multi-variable smart device)

```
package devicecontrollermultiple;

public class Controller {

    public Controller() {

    }

    public String getBrightness() {

    }

    public String getState() {

    }

    public String setBrightness(String value) {

    }

    public String setState(String state) {

    }
}
```

Figure 19

Figure 19: A graph showing average processing times for setting a light (experiment one)

Figure 19: A graph showing average processing times for setting a light (experiment one)
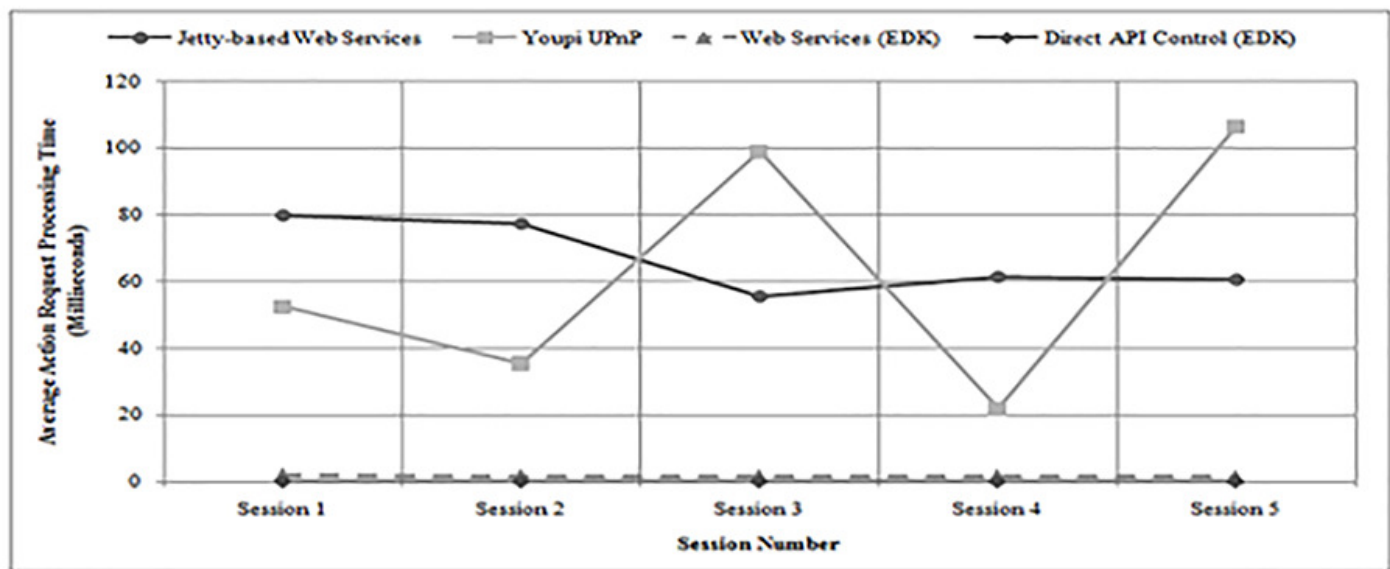
Figure 20

Figure 20: A graph showing average processing times for setting a light (experiment two)