Extensions to Middleware Ginga for Integration with IoT Environments

Bruno Carvalho de Farias
TPV Technology and Institute of Computing - IComp
Federal University of Amazonas
Manaus, Brazil 69077-000
Email: bruno.farias@tpv-tech.com

Eddie B. de Lima Filho and Orlewilson Bentes Maia
TPV Technology
Manaus, Brazil 69058-830
Email: {eddie.filho, orlewilson.maia}@tpv-tech.com

Eduardo Souto
Institute of Computing - IComp
Federal University of Amazonas
Manaus, Brazil 69077-000
Email: esouto@icomp.ufam.edu.br

Abstract—Ginga is a middleware specified to offer support to execution of interactive applications, on digital TV (DTV) receivers. Although it offers an extensive application programming interface (API), it does not provide integration with IoT devices, which would favor its use in Internet of things (IoT) environments. This work aims to integrate Ginga with the IoT technology, through support to the message queuing telemetry transport (MQTT) communication protocol. Ginga’s API has been extended with support to MQTT, which resulted in an integrated framework regarding IoT and DTV. As a result, reference applications were developed, in order to validate the proposed framework, in a home network.

I. INTRODUCTION

Internet of things (IoT) environments are composed of several physical devices with different communication protocols, over a network [1]. In addition, cloud computing platforms are usually employed in this domain, as they complement this type of environment and meet important requirements, such as scalability, storage capacity, and availability. Therefore, developers need to be aware of different protocols, data formats, and software tools, i.e., libraries and utilities, in order to be able to develop complex and value-added applications.

In that context, a middleware layer presents potential to minimize this problem, which would provide uniform and generic interfaces for communication between different types of devices and reduce complexity and development costs. Moreover, in home environments, digital television (DTV) receivers play central roles, as in Brazil, where more than 95% of its population uses them for information and entertainment [2], which leads to the need for their integration into IoT networks. As a consequence, a possible solution that incorporates all those aspects and also takes into account the Brazilian context would be to use the middleware defined in the Brazilian digital television system (SBTVD), known as Ginga; however, it does not provide software components capable of providing integration with IoT networks.

Ginga [3] provides execution of interactive applications, on DTV receivers, and a significant portion of the latter already has an embedded version of Ginga, along with network connections and other hardware interfaces. As a result, this infrastructure could be exploited, in application development, for interconnecting DTV receivers and other IoT devices, in a home network. Moreover, as current DTV devices have multiple hardware interfaces, and their computational power has been enhanced in recent years, e.g., with the integration of graphics processing units (GPU), mainly due to demanding applications, such as hypertext markup language (HTML) version 5 engines, they could even be used to enable home automation and then favor communication with IoT devices, cloud computing platforms, and other DTV applications.

Such an approach was already exploited by other authors, such as Cavalcante [4] and Teixeira [5], and the incorporation of IoT protocols, together with computing platforms, could result in higher adoption of Ginga, since it is often superficially explored. In addition, as already mentioned, since DTV devices are central in Brazilian homes and often have multiple interfaces (e.g., remote control, Wi-Fi, and Bluetooth), it makes sense to use them as IoT gateways, in home environments.

The last paragraphs inspired this work, which proposes a framework aiming at IoT-application development through an API in Ginga, based on the message queuing telemetry transport (MQTT) protocol, in order to allow IoT applications, for DTV environments, capable of running on different TV platforms. Such an integration will certainly make Ginga more aligned with usual needs from IoT applications and capable of serving a wider range of applications, in different domains.

Finally, from a practical point of view, this will result in the development of networked DTV receiver applications being accessed by regular DTV users, such as active IoT devices able to perform data acquisition, processing, and visualization, through APIs written specifically for those purposes.

II. THE MQTT PROTOCOL

As already mentioned, an API in middleware Ginga for communication with IoT devices, was based on MQTT, which is an asynchronous communication protocol standardized by the International Organization for Standardization that had been proposed and implemented by IBM, in 1999 [6]. MQTT is commonly used for communication between devices, in IoT networks, due to some interesting features, such as being lightweight and capable of using a communication mode known as publish-subscribe, which reduces network use and favors application development. This model is composed by
a central node, named as Broker, which is in charge for managing connections and messages, and associated clients, which send and receive messages related to a topic, through a Broker, as can be seen in the architecture shown in Fig. 1.

![Broker Architecture](Image)

This way, every time a client publishes a message, it must be associated to a topic. All clients that subscribe to a specific topic receive messages associated to that, through a Broker, which manages communication and acts as a gateway. Consequently, clients must connect to a Broker, in order to receive and publish messages related to a topic, which are structured as follows: a fixed header, a variable complementary header, which depends on the type of a given message, and payload, which is composed by a command and respective data.

In MQTT, a client starts a communication attempt by sending a command CONNECT, in order to establish connection with a Broker, which then responds with a message CONNACK that bears parameters sessionPresent and returnCode. Indeed, the latter must be 0, in case of a successful connection; otherwise, in case of fail, other values are returned.

After a connection is established, clients can send messages SUBSCRIBE, in order to tell a Broker that they intend to receive messages related to a given topic, which bear parameters qos and topic. The former indicates the quality of service (QoS) level for messages delivery, as requested by a client, which can assume values between 0 and 2, while the latter is a character string that indicates the subject which a client wants to be notified about. When receiving command SUBSCRIBE, a Broker responds with SUBACK, along with a parameter returnCode, which may be assigned with values between 0 and 2, in case of success, and 128, in case of fail.

Clients intending to send information to other nodes, in an IoT network, should post messages of type PUBLISH, along with parameters topicName, qos, and retainFlag in their headers, in addition to information in their payload sections. In case a message-publishing procedure is successful, the respective sender receives a message PUBACK from the managing Broker, which reports that a message was correctly registered.

Given that MQTT uses the transport control protocol (TCP), it ensures message delivery to a given Broker; however, access to specific messages depends on the QoS level chosen by a sender. For instance, if the parameter qos is received with value 0, a message is immediately sent and, in case a client is not connected at that time, it does not receive it; however, if qos is equal to 1, a Broker ensures that a message will be received by a client, but it may be sent more than once, which is indicated in its header. Finally, in case qos is equal to 2, a handshake procedure is executed, which avoids message duplication.

In summary, MQTT messages are composed by parameters and associated information, which can be regarded as a simple stream of bits. As a consequence, Lua events [3] can be used for both sending and receiving them, in a well formatted framework. Additionally, Ginga already provides support to communication through TCP, which can be used as basis for implementing an MQTT API, which will be tackled ahead.

III. THE TCP API IN GINGA

Ginga applications are based on two key technologies: nested context language (NCL) and Lua. NCL documents are written for a declarative application, with Lua scripts for coding in an imperative paradigm [7], as happens with HTML and JavaScript. An example of NCL code with a Lua script can be seen in Listing 1, where a Lua script script.lua is executed, which, at its beginning, triggers the exhibition of image Im.jpg.

In Lua, communication with NCL and external elements is achieved with events [3], which can be posted and received. For instance, a TCP communication session can be started with an event $evt = \{ class='tcp', type='connect', host=addr, port=number, [interface_addr=addr], [local_port=number], [data_type=string], [timeout=number] \}$ [3]. Indeed, there already is network communication, which provides means for sending and receiving data through TCP connections, with an asynchronous communication mechanism, and a function should be registered, in order to provide handling code.

As a consequence, a Ginga-application developer may use a specific function (register) for registering a callback, when a connection with a TCP server is established, as shown in Listing 2. Besides, this same callback is called whenever data are received through a TCP connection. An application can also be disconnected from a server, with the goal of ending a TCP communication cycle.

```
<body>
  <port id="pStart" component="lua"/>
  <media id="lua" src="script.lua"
        descriptor="dsLua"/>
    <area id="Ev" label="comm"/>
</media>
  <media id="Im" src="Im.jpg"
        descriptor="dsIm"/>
    <link xconnector="onBeginStart">
      <bind role="onBegin" component="lua"
            interface="Ev"/>
    </link>
  </body>
```

Listing 1. NCL code example with Lua script.
Finally, such a TCP API presents all functions necessary for the development of an API capable of providing support to MQTT communication, which was used in the present work.

```lua
event.register(connectionHandler, 'tcp')
...

function connectionHandler(evt)
  if (evt.type == 'connect') then
    print("TCP connection event")
    print("*** host: ", evt.host)
    print("*** port: ", evt.port)
    print("*** connection: ",
          evt.connection)
    print("*** error: ", evt.error)
  return
  ConnectEvent = evt
end

Listing 2. Code in Lua for registering a handler for TCP events.
```

IV. THE PROPOSED FRAMEWORK

This study proposes a framework for integrating DTV receivers, with a middleware Ginga port, into MQTT-based networks, through the development of an MQTT API in Lua, which was based on Eclipse Paho [8] and relies on the middleware Ginga’s Lua TCP module. Such an approach requires the understanding of MQTT message assembling, in order to send TCP-based messages capable of being received and understood by an MQTT broker. The proposed Lua API was implemented in the form of a table named `mqtt` [3], which contains definitions of methods that provide support to MQTT messages, as described below.

In order for a client to be able to send MQTT messages, it must establish connection with a Broker. Consequently, a method `mqtt:connect(uri)` has been defined and must be called by an application developer, in order to establish connection with a server, where parameter `uri` contains a server’s internet protocol (IP) address. The proposed API also defines `mqtt:onConnect(callback)`, which allows the definition of a function that should be called when a Broker responds to a connection request previously performed with `mqtt:connect(uri)`. That procedure can be used for checking if a connection has been correctly established and thereby perform a given action, such as subscribing to some topic. It is recommended that `onConnect` be called before `connect` is invoked, so that a client can properly verify a connection result.

Any subscription to a topic must be done through `mqtt:subscribe(topic)`, which expects a string containing the topic to which an application needs to register to. Next, method `mqtt:onMessage(callback)` should be used for registering a callback that is invoked whenever a message associated with `topic` reaches a Broker and that must contain two strings as parameters (one containing the message and other which will contain the topic related with the message). Message publishing is done through method `mqtt:publish(topic, message)`, which expects two strings that define related topic and message to be transmitted. The `mqtt:unsubscribe(topic)` method allows a client to unsubscribe from `topic`, which then stops receiving messages associated to that. Finally, Fig. 2 contains a resume regarding the organization of the proposed solution.

![Fig. 2. Structure of the proposed Lua MQTT approach.](image)

It is worth mentioning that the Lua MQTT framework developed in this work can be easily incorporated into NCL applications, using native Lua language features [9], which allows inclusion of external scripts. Thus, definitions performed in script `mqtt.lua` can be used by interactive applications to access the MQTT communication infrastructure. Indeed, if one considers that definitions regarding MQTT functionalities can be included through a separate Lua file, no middleware update, regarding legacy implementations, is needed, because a broadcaster can send that file along with a Ginga interactive NCL application, in a digital storage media command and control (DSM-CC) object carousel [10]. Consequently, the adoption of such a framework would be transparent to DTV users, who should only tune to a given DTV channel and perform the same actions. In summary, this approach favors middleware extensions, by exempting receiver manufacturers of TV firmware upgrades. Indeed, support to new features usually requires development and, ultimately, legacy products are replaced by a service center or some sort of technical support. This way, the approach developed in this paper provides a way of reducing complexity regarding extensions to middleware Ginga.

Finally, one discussion that may arise is the location of a Broker, for the proposed scenario, in which a Ginga-enabled DTV receiver is part of an IoT network. First, such an environment is not inherently centered around a DTV receiver, because the latter is actually being integrated into it. Actually, that choice depends, for instance, on which element presents the most suitable features, e.g., processing power or longest...
active period, in a given configuration, which could indeed be a DTV receiver and that might even reduce the number of devices in a network. Nonetheless, if a given DTV receiver is a low-end product, an external device could also be employed as MQTT broker. In order to provide a broader evaluation scenario, a Broker, in the experiments performed here for validating the proposed methodology, will be available in a personal computer (PC).

V. EXPERIMENTAL RESULTS

The validation of mqtt.lua was performed through communication with an IoT node, which demonstrated the effectiveness of the proposed framework. In the experiments carried out during the present work, a set-top box EiTV SmartBox ET-SBX01 [11] executed an interactive Ginga NCL application, from an USB pen drive, and established communication with a NodeMCU board with chip ESP8266 [12], using the structure provided by mqtt.lua. The Mosquitto software was installed from an apt-get package in a personal computer, running a Ubuntu Linux distribution [13], and used as Broker. A summary of this organization can be visualized in Figure 3.

During the mentioned tests, a light dependent resistor (LDR), a temperature and humidity sensor DHT-22, a distance sensor HC-SR04, and an light emitting diode (LED) were connected to NodeMCU, which read values from attached sensors and LED status, and sent them via MQTT to a Broker running on a PC and responsible for forwarding messages to the EiTV set-top box. The main application developed with the proposed MQTT framework, provided by mqtt.lua, was also able to send commands to the same LED, through the set-top box’s remote control. In summary, whenever key RED [3] was pressed, the application published a message OFF in topic nodemcu/control. Then, NodeMCU [12], which had already subscribed to that topic, checked if a message OFF had arrived, which led to a command for turning off the LED, on a specific digital port of its embedded microcontroller. If a user pressed key GREEN [3], the same processing chain occurred, but, in that case, the receiver sent a message ON, which made NodeMCU to turn the LED on. Finally, Listing 3 shows a snippet of Lua code developed for the present experiments, which exemplifies the use of the proposed framework. Figure 4 shows the developed application being executed on the chosen set-top box and exhibited by a commercial TV set. In summary, a complete integration between DTV receiver and IoT network was achieved, which demonstrated the effectiveness of the proposed framework and even suggested other applications, such as the ones relying on DTV receivers as IoT hubs.

```
function connectHandler()
  mqtt:subscribe("nodemcu/state/led");
  mqtt:subscribe("nodemcu/state/ldr");
  mqtt:subscribe("nodemcu/state/dist");
  mqtt:subscribe("nodemcu/state/temp");
  mqtt:subscribe("nodemcu/state/humid");
end;

function messageHandler(message, topic)
  print("<app> Message: " ..message)
end;

mqtt:onConnect(connectHandler)
mqtt:connect("192.168.11.6")
mqtt:onMessage(messageHandler)

function keyHandler(evt)
  if (evt.key == 'GREEN') then
    mqtt:publish("nodemcu/control/led", "on")
  elseif (evt.key == 'RED') then
    print("<app> LED OFF")
    mqtt:publish("nodemcu/control/led", "off")
  end
end;

event.register(keyHandler);
```

Listing 3. Lua code excerpt implemented for experiments.

VI. CONCLUSION

This work tackled a new paradigm that can become usual in the near future: DTV receivers connected to an IoT network and able to communicate with other elements in that same environment. In summary, the framework proposed here added new features to middleware Ginga [3], which allowed integration of DTV receivers into IoT applications, through an API developed in Lua. Indeed, the incorporation of IoT protocols has the potential to provide greater adoption of Ginga itself, since that it is only superficially explored by applications currently sent by broadcasters. Consequently, a home network
centered around DTV receivers would then be possible, even with integration between DTV and IoT environments, which would turn DTV sets into service gateways. Such an integrated environment may also include integrated broadband-broadcast implementations, which can be enhanced by IoT nodes.

In addition, the definition of a standardized middleware-level interface favors the development of IoT applications for DTV and ensures portability among commercial Ginga-enabled receivers. The experiments carried out in the context of this work have shown that incorporating features through new Lua APIs is effective, since those extensions are straightforward and Ginga already provides support to TCP, which led to the development of Lua MQTT modules. Future work includes new Lua APIs and research regarding automatic device scanning and configuration, on a local network.

ACKNOWLEDGMENT

This research was partially supported by Priority Program for the Training of Human Resources - CAPDA/SUFRA-MA/MDIC, under the terms of Federal Law 8.387/1991.

REFERENCES