ПРОГРАММИРОВАНИЕ, 2021, № 4, с. 20–30

ПАРАЛЛЕЛЬНОЕ И РАСПРЕДЕЛЕННОЕ ПРОГРАММИРОВАНИЕ

УДК 004.75

ОСОБЕННОСТИ ВЗАИМОДЕЙСТВИЯ УСТРОЙСТВ С ИНФРАСТРУКТУРОЙ ИНТЕРНЕТА ВЕЩЕЙ НА ПРИМЕРЕ ИНФРАСТРУКТУР Amazon Web Services И Microsoft Azure

© 2021 г. С. И. Жуков[a,*]

[a] *Московский государственный университет имени М.В. Ломоносова*
Научно-исследовательский вычислительный центр
119234, Москва, ул. Колмогорова, д. 1С4, м, Россия

*****E-mail: serge.zhukov@auriga.com*

Поступила в редакцию 28.10.2020 г.
После доработки 18.11.2020 г.
Принята к публикации 24.11.2020 г.

Облачные инфраструктуры Amazon Web Services и Microsoft Azure поддерживают взаимодействие с IoT-устройствами (устройствами интернета вещей) по протоколу MQTT. Однако, интерфейс IoT-инфраструктуры несколько отличается, и разработка программного обеспечения для устройства, которое могло бы работать с обеими инфраструктурами, требует учета этих особенностей.

DOI: 10.31857/S0132347421040087

1. ВВЕДЕНИЕ

Контекстом данной работы явилась разработка встроенного программного обеспечения (ПО) для устройства – интеллектуального контроллера для дата-центров, реализующего электропитание, охлаждение, контроль доступа и другие функции. Конкретная функциональность контроллера зависит от набора подключенных к нему сенсоров и контроллеров более низкого уровня, а также от набора правил и логики, задаваемых пользователем. В частности, контроллер может заниматься управлением охлаждением, контролем и мониторингом доступа, сбором данных с сенсоров и их агрегированием для формирования массивов больших данных, оповещением о событиях и неисправностях.

Одной из задач при разработке встроенного ПО была реализация интерфейса “интернета вещей” (Internet of Things – IoT). При этом интеллектуальное устройство подключается через сеть Интернет к облачной инфраструктуре, поддерживающей “интернет вещей”, и взаимодействует с ней (как правило, посредством протокола MQTT). Сейчас существует несколько таких инфраструктур; наиболее популярные – Amazon Web Services (далее – AWS), Microsoft Azure (далее – Azure), Google Cloud IoT, IBM Watson IoT.

Устройство передает в облако данные, собираемые с сенсоров, для хранения и анализа, а также принимает команды управления, передаваемые через инфраструктуру. Также ПО устройства должно поддерживать механизмы для регистрации устройства в инфраструктуре и доставки IoT-конфигурации на устройство по запросу от него.

Первоначально в качестве основной облачной инфраструктуры рассматривалась AWS, однако затем возникла задача – обеспечить также возможность подключения устройства к другим инфраструктурам. При этом необходимо структурировать встроенное ПО устройства таким образом, чтобы возможно большая часть интерфейса IoT оставалась независимой от используемой инфраструктуры, а инфраструктурно-зависимый код находился бы в небольших по размеру интерфейсных модулях.

Эта задача была успешно решена для инфраструктуры Microsoft Azure, при этом выяснилось, каковы отличия IoT-интерфейса между этими инфраструктурами и как их преодолеть на уровне встроенного ПО IoT-устройства. Для доступа к инфраструктуре использовалась клиентская библиотека AWS с небольшими изменениями для поддержки Azure. Разработаны также общие механизмы для регистрации устройства в инфраструктуре и доставки IoT- конфигурации на устройство.

Сравнение возможностей обеих инфраструктур производилось в следующих направлениях:

– особенности реализации протокола MQTT;

– аутентификация устройства в IoT-инфраструктуре;

– поддержка топиков MQTT;

– сохраненное состояние устройства (Shadows и Twins);

– методы прямого действия (Azure);

– MQTT над WebSockets.

В качестве возможного подхода к решению задачи рассматривалась также архитектура Web of Things – архитектура, специально разработанная для обеспечения совместимости IoT-инфрастуктур и IoT-приложений. Однако, этот подход более применим к интерфейсу между приложением и инфраструктурой, чем к интерфейсу между инфраструктурой и устройством.

Дальнейшее изложение материала отражает анализ и сравнение возможностей инфраструктур AWS и Azure в контексте описанной выше задачи.

2. ОСОБЕННОСТИ РЕАЛИЗАЦИИ ПРОТОКОЛА MQTT

Протокол MQTT – протокол прикладного уровня, используемый для передачи сообщений между IoT-устройствами и облачной IoT-инфраструктурой. Для него существует промышленный стандарт [1]. Протокол основан на модели Publish-subscribe. При этом участники взаимодействия могут подписываться (subscribe) на сообщения с определенной темой. Когда кто-то из участников отправляет сообщение, оно отправляется широковещательно (publish) и его получают те участники, которые подписаны на него. Участниками взаимодействия являются как IoT-устройства, принадлежащие к определенной группе, так и исполняемые объекты внутри облачной инфраструктуры (приложения, диспетчеры событий, так называемые лямбда-функции и т.д.). Данные в протоколе MQTT передаются в текстовом структурированном формате JSON (Java Script Object Notation, [2]).

В обоих рассматриваемых облачных инфраструктурах реализован протокол MQTT версии 3.1.1 с определенными ограничениями относительно стандарта. Он реализован поверх соединения TLS между устройством и инфраструктурой, так что сообщения передаются в зашифрованном виде и поверх установленного TCP-соединения.

Основные ограничения, налагаемые обеими рассматриваемыми инфраструктурами:

– поддерживается только одно соединение между устройством и инфраструктурой;

– ограничено количество сообщений в единицу времени, которое посылает устройство;

– не поддерживаются сообщения с гарантированной однократной доставкой (определенное протоколом качество сервиса QoS=2).

3. АУТЕНТИФИКАЦИЯ УСТРОЙСТВА

Аутентификация устройства при установлении соединения TLS для AWS возможна только с использованием сертификата X.509 [3] (за исключением режима MQTT над WebSockets), а для Azure – по имени пользователя/паролю или также с использованием сертификата X.509. Для единообразия был выбран метод аутентификации с использованием сертификата для обеих инфраструктур.

В AWS сертификат устройства автоматически создается при регистрации устройства в IoT-инфраструктуре и хранится внутри инфраструктуры. После этого достаточно загрузить его на устройство и указать ссылку на него в конфигурационном файле клиентской библиотеки.

В Azure сертификат устройства нужно создавать вручную; при этом родительский сертификат нужно зарегистрировать в облаке. После регистрации инфраструктура принимает для аутентификации устройства любой сертификат, созданный на основе зарегистрированного родительского сертификата.

4. РЕГИСТРАЦИЯ УСТРОЙСТВА В ИНФРАСТРУКТУРЕ И ЗАГРУЗКА IOT-КОНФИГУРАЦИИ НА УСТРОЙСТВО

Чтобы устройство могло установить соединение с IoT-инфраструктурой, оно должно быть зарегистрировано в ней и иметь уникальный сертификат. Из соображений безопасности регистрация устройства реализуется через отдельное облачное приложение, доступное только пользователю инфраструктуры с правами локального администратора.

При регистрации устройства администратор задает имя устройства, его MAC-адрес (он уникален для каждого устройства) и тип устройства. В облачной базе данных (внутренняя база данных устройств в случае AWS, BLOB (Binary Large OBject) – контейнер в случае Azure) создается запись для данного устройства, содержащая информацию о нем, включая сертификат.

Для упрощения работы функция создания нового сертификата и конфигурационного файла для клиентской библиотеки в Azure была реализована в виде облачного приложения, реализующего HTTP REST API [4]. Регистрация нового устройства в инфраструктуре производится через это приложение. При этом оно создает соответствующие данные (сертификат, приватный ключ, конфигурационный файл), которые затем могут быть загружены на устройство путем вызова REST API. Для хранения этих данных приложение создает специальный контейнер (так называемый BLOB-контейнер) в хранилище данных Azure.

Аналогичное приложение было разработано и для инфраструктуры AWS, хотя регистрация устройства в AWS может быть сделана через AWS Console, которое представляет собой встроенный

Web-интерфейс инфраструктуры. Использование приложения позволяет осуществить регистрацию устройства “в один клик”, а также просмотреть список зарегистрированных устройств и удалить регистрацию устройства, не выходя из приложения.

Для зарегистрированного устройства доступна операция загрузки IoT-конфигурации на него. Это может быть сделано либо при инициализации устройства, либо в процессе работы устройства. Для обеих инфраструктур эта операция реализована примерно одинаково. Её общее описание можно представить в виде перечисленных далее действий.

– Встроенное программное обеспечение устройства реализует специальную функцию, которая может быть вызвана через локальный пользовательский интерфейс (Command Line Interface, touchscreen, локальный Web интерфейс). Ей передается имя устройства.

– Эта функция формирует HTTP POST запрос, включающий в его параметры имя устройства и локальный MAC-адрес. Запрос посылается на выделенный URL, принадлежащий соответствующей IoT-инфраструктуре; к этому URL привязано написанное нами облачное приложение (лямбда-функция в случае AWS, Web API приложение в случае Azure).

– Это приложение запускается инфраструктурой, получает параметры запроса и обращается к базе зарегистрированных устройств, выбирая запись по имени устройства и MAC-адресу из параметров запроса.

– В случае успешного нахождения записи, приложение формирует конфигурационный файл, включающий в себя URL оконечной точки, номера портов, сертификат и приватный ключ и другие параметры соединения, и возвращает их в ответе на запрос POST в JSON формате. Встроенное программное обеспечение обрабатывает эти данные и формирует конфигурационные файлы, чтобы подключиться к инфраструктуре после перезапуска.

– В случае неудачного поиска, приложение возвращает код ошибки 404 (страница не найдена) в ответ на запрос. Встроенное программное обеспечение сообщает пользователю об ошибке.

Часть кода, выполняемого на устройстве, для получения конфигурационного файла из облака (с использованием библиотеки CURL), приведена ниже:

```
  //
  //  Function:
  //      PopulateCloudConfigurationFilesInternal
  //  Synopsis:
  //      This function retrieves configuration file and certificates
  //      by POSTing to the designated URL in the cloud, passing it
  //      the thing name and MAC address
  //  Parameters:
  //      api_url - the designated URL which implements the API
  //      root_ca_url - the URL for the root certificate
  //      curl - the CURL context
  //      directory - where to put the configuration file and certificates
  //      thing_name - the thing name
  //      mac_address - the MAC address
  //      curl_error - the error message from CURL is placed here
  //  Return value:
  //      Error code: 0 for success, negative value for a failure
  //
static size_t write_data_function(void *buffer, size_t size, size_t nmemb,
void *userp)
{
    std::string *result = (std::string *)userp;
    result->append((const char *)buffer, size*nmemb);
    return size * nmemb;
}
static SaErrorT PopulateCloudConfigurationFilesInternal(const std::string
&api_url,
```

```
       const std::string &root_ca_url, CURL *curl, const std::string &directory,
       const std::string &thing_name, const std::string &mac_address,
    std::string &curl_error)
{
         CURLcode res;
         long httpCode = 0;

         struct curl_slist *headers = NULL;
                  headers = curl_slist_append(headers, "Content-Type: ap-
plication/json");
    headers = curl_slist_append(headers, "Accept: application/json");
       std::string post_params = "{\"thingName\":\"" + thing_name + "\",\"mac-
Address\":\"" + mac_address + "\",\"function\":\"getCertificate\"}";
    curl_easy_setopt(curl, CURLOPT_URL, api_url.c_str());
    curl_easy_setopt(curl, CURLOPT_POSTFIELDS, post_params.c_str());
    curl_easy_setopt(curl, CURLOPT_POSTFIELDSIZE, post_params.size());
    curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers);

    std::string result;
    curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_data_function);
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, &result);
    res = curl_easy_perform(curl);
    curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &httpCode);
    curl_slist_free_all(headers);
    if (res || httpCode >= 300) {
        //  Download error
        if(res == 0 && httpCode == 404) {
            // Not found on POST - configuration does not exist
            return SA_ERR_HPI_NOT_PRESENT;
        }
        curl_error = res ? "Failed to obtain a certificate from the cloud: "
+ std::string(curl_easy_strerror(res)) : "HTTP Error " + std::to_string(http-
Code);
        return SA_ERR_HPI_ERROR;
    }
    try {
        json jsondata;
        std::istringstream ifile(result);
        ifile >> jsondata;
        // Subsequent code parses the JSON response in "jsondata" and places
the fields
        // of the JSON object to appropriate files
    …
        return SA_OK;
    } catch (std::exception &e) {
         dbg_print(DBG_ERROR, "Exception reading json data from \'result\':
%s\n", e.what());
        return SA_ERR_HPI_NOT_PRESENT;
    }
}
```

5. ИСПОЛЬЗОВАНИЕ КЛИЕНТСКОЙ БИБЛИОТЕКИ AWS НА IOT-УСТРОЙСТВЕ ДЛЯ ПОДКЛЮЧЕНИЯ К AZURE

Обе инфраструктуры предоставляют библиотеки для программного обеспечения IoT-устройств на языке C/C++, обеспечивающие доступ к функциям протокола MQTT, в том числе – установление соединения с инфраструктурой. Обе библиотеки предоставляют примерно одинаковые функциональные возможности, однако интерфейс их существенно отличается. Чтобы избежать массированной адаптации кода, вызывающего библиотечные функции, было решено использовать одну из библиотек для работы с обеими инфраструктурами.

Это оказалось возможным для клиентской библиотеки AWS [5]. Потребовались лишь небольшие изменения для заполнения полей “username” и “password” в MQTT-команде установления соединения. Эти поля не используются в AWS, но используются в Azure даже при аутентификации устройства по сертификату.

Кроме того, были добавлены распознавание и обработка жесткого разрыва соединения инфраструктурой Azure в случае ошибок протокола. В этом случае библиотека автоматически переустанавливает соединение и восстанавливает подписку на соответствующие топики MQTT, так же как и при разрыве соединения по другим причинам. Разрыв и восстановление соединения между устройством и инфраструктурой может происходить достаточно часто, в основном из-за географической удаленности устройств от соответствующей оконечной точки инфраструктуры.

6. ПОДДЕРЖКА ТОПИКОВ MQTT

Топик в MQTT соответствует теме публикуемого сообщения и состоит из нескольких частей разделенных символом ‘/’.

Для работы с IoT-устройствами нами была разработана система топиков, соответствующая набору функций, поддерживаемых устройствами:

– асинхронные события от устройства: посылаются на топик “<device-name>/events”;

– данные телеметрии от сенсоров устройства: посылаются на топик “<device-name>/response/sensor/<sensor-number>/reading”;

– действия типа запрос-ответ, например, запрос от устройства списка сенсоров, реализуются следующим образом: устройство подписывается на топик “<device-name>/request/#”, где “#” по соглашениям MQTT обозначает произвольное содержимое, и отвечает на топик, начинающийся с “<device-name>/response”. Например, чтобы получить общую информацию об устройстве, облачное приложение посылает MQTT-сообщение “<device-name>/request/general_information” и ожидает ответа с топиком “<device-name>/response/general_information”;

Инфраструктура AWS поддерживает использование практически любых топиков, разрешенных протоколом. Поэтому при работе с AWS набор топиков, разработанный для IoT-устройства, использовался в неизменном виде.

Azure поддерживает только топики нескольких жестко заданных форматов, при использовании устройством топика в любом другом формате инфраструктура немедленно и без объяснений разрывает MQTT-соединение.

Для MQTT-сообщений, исходящих от устройства (события, телеметрия, ответы на запросы от клиентов), единственный разрешенный формат топика в Azure имеет вид: “devices/<device-name>/messages/events/<property-bag>”. Здесь <device-name> – это имя, под которым устройство зарегистрировано в Azure IoT-инфраструктуре, а <property-bag> – набор пар “<имя>=<значение>”, разделенных символом ‘&’.

Для MQTT-сообщений, направленных устройству, формат топика имеет вид “devices/<device-name>/messages/devicebound/#”, где “#” по соглашениям MQTT обозначает произвольное содержимое.

Чтобы адаптировать принятую для IoT-устройства систему топиков для Azure, было решено применить следующую трансляцию: асинхронные события от устройств посылаются на основной топик “devices/<device-name>/messages/event”. Для топиков, используемых для запросов и ответов, а также для телеметрии, часть топика после имени устройства делается значением свойства с именем “subtopic” с заменой символов ‘/’ на ‘.’.

Например, для получения списка сенсоров на устройстве ему отправляется запрос с топиком “<device-name>/request/sensor/list”, а устройство посылает ответ с топиком “<device-name>/response/sensor/list”. В случае Azure топик запроса выглядит так: “ devices/<device-name>/messages/devicebound/subtopic=request.sensor.list”. В ответе устройства используется топик “devices/<device-name>/messages/events/subtopic=response.sensor.list”.

Данные телеметрии от сенсора номер 12 посылаются с топиком “<device-name>/response/sensor/12/reading”, который транслируется в Azure в “devices/<device-name>/messages/events/subtopic=response.sensor.12.reading”.

Выполняемая на устройстве функция трансляции топика в формат Azure приведена ниже:

```
std::string cSmrMQTTDispatcher::TranslateAzureTopic(const std::string &topic)
{
    if (startsWith(topic, "$iothub/")) {
        // Topic is already in Azure format, nothing to do
        return topic;
    }
    std::string result;
    const std::string messages_events = "/messages/events/";
    size_t pos = topic.find(messages_events);
    if (pos != std::string::npos) {
        // Topic is in standard format, need to translate
        result = topic;
        pos += messages_events.size();
        if (pos < result.size()) {
            result.insert(pos, "subtopic=");
            while((pos = result.find("/", pos)) != std::string::npos) {
                result[pos] = '.';
            }
        }
    }
    return result;
}
```

7. ПРОГРАММНЫЕ МЕХАНИЗМЫ SHADOWS И TWINS

Эти механизмы имеют название device shadow (“тень устройства”) в AWS и device twin (“близнец устройства”) в Azure [6, 7]. Они имеют очень похожую реализацию и предназначены для доступа к свойствам устройства в любой момент, независимо от статуса подключения устройства.

Shadow/twin представляет собой документ в формате JSON. Он включает в себя два набора свойств, а именно, desired и reported. В наборе reported хранится закешированная информация об устройстве в виде набора свойств. Когда устройство подключено к инфраструктуре, оно периодически обновляет информацию о себе в shadow/twin. Если устройство не подключено в данный момент, клиент может получить информацию от shadow/twin, сохраненную с момента последнего обновления (включая время последнего обновления).

В наборе desired хранятся запросы клиента на изменения свойств устройства (“желательные” значения свойств для клиента). В момент изменения этого набора клиентом или когда устройство подключается к инфраструктуре (если оно было отключено в момент изменения), устройство получает извещение по специальному топику, и может прочитать запрос на изменение свойств в наборе desired и изменить указанные свойства. После изменения устройство обновляет информацию в shadow/twin, перенося изменения обратно в набор reported.

Поскольку реализация shadows и twins очень похожа, функциональность встроенного программного обеспечения для работы с этими механизмами в AWS и в Azure отличается минимально. Имена свойств в Azure twins не могут содержать пробелы, в AWS shadow такое ограничение отсутствует, поэтому при адаптации к Azure пробелы в именах свойств были удалены. Для работы с shadow/twin используются специальные топики MQTT, для которых существует примерное соответствие [8, 9]. Основные топики перечислены в таблице 1.

В обеих инфраструктурах размер сохраняемых данных ограничен (8 Кб в AWS, 32 Кб в Azure), поэтому в shadow/twin сохраняются только самые основные свойства IoT-устройства. В нашем случае они включают следующее:

– идентификационные атрибуты устройства: имя, тип устройства, название модели и изготовителя, дата изготовления, аппаратная версия, серийный номер;

– сетевые адреса: IP-адрес (IPv4 или IPv6), MAC-адрес, маска подсети, адрес шлюза и DNS-серверов;

Таблица 1.

Функция топика	Инициатор сообщения	AWS shadow	Azure twin
Обновление информации в секции reported	Устройство	$aws/things/<device-name>/shadow/update	$iothub/twin/PATCH/properties/ reported/?$rid=<rid> (где <rid> – уникальный числовой идентификатор запроса)
Обновленная информация принята	Shadow/twin	$aws/things/<device-name>/shadow/update/ accepted	$iothub/twin/res/204/?$rid= <rid> (<rid> соответствует <rid> запроса)
Обновленная информация отвергнута	Shadow/twin	$aws/things/<device-name>/shadow/update/ rejected	$iothub/twin/res/<HTTP-error-code>/?$rid=<rid> (<rid> соответствует <rid> запроса)
Обновление информации в секции desired	Shadow/twin	$aws/things/<device-name>/shadow/delta	$iothub/twin/PATCH/properties/ desired/...
Чтение информации из shadow/twin	Устройство	$aws/things/<device-name>/shadow/get	$iothub/twin/GET/?$rid=<rid> (где <rid> – уникальный числовой идентификатор запроса)
Чтение информации из shadow/twin – ответ с данными	Shadow/twin	$aws/things/<device-name>/shadow/get/ accepted	$iothub/twin/res/200/?$rid= <rid> (<rid> соответствует <rid> запроса)
Чтение информации из shadow/twin – запрос отвергнут	Shadow/twin	$aws/things/<device-name>/shadow/get/ rejected	$iothub/twin/res/<HTTP-error-code>/?$rid=<rid> (<rid> соответствует <rid> запроса)

– версия встроенного программного обеспечения;

– информация о географическом местонахождении устройства.

8. DIRECT METHODS B AZURE

Протокол MQTT поддерживает только одностороннюю передачу сообщений и не поддерживает взаимодействие типа запрос-ответ. Однако, в Azure поверх MQTT реализован механизм direct methods (“методов прямого действия”), которые поддерживают такое взаимодействие и позволяют клиенту передать запрос на устройство и синхронно получить от него ответ (т.е. удаленно вызвать метод на устройстве) [10].

Механизм direct methods реализован следующим образом: клиент вызывает специальную API-функцию инфраструктуры, передавая ей имя устройства, имя метода, параметры в формате JSON и таймаут ожидания ответа от устройства. Инфраструктура формирует MQTT сообщение с топиком “$iothub/methods/POST/<method-name>/?$rid=<rid>” и отправляет его устройству. Здесь <method-name> – это имя метода, а <rid> – уникальный идентификатор запроса, в виде шестнадцатеричного числа.

Устройство, получив данное сообщение, обрабатывает запрос, вызывает соответствующий метод и посылает в ответ MQTT-сообщение с топиком “ $iothub/methods/res/<HTTP-result-code>/?$rid=<rid>. Здесь <rid> должен совпадать с <rid> в запросе, а <HTTP-result-code> информирует инфраструктуру о успехе или неудаче выполнения метода. Используются стандартные коды возврата HTTP. Так, в случае успеха код равен 200, если метод возвращает данные в формате JSON, или 204, если метод не возвращает данных. В случае неудачи используются коды 4xx, 5xx. Если метод возвращает данные, они передаются в составе MQTT-сообщения и возвращаются клиенту как результат выполнения метода.

Direct methods предоставляют большое удобство для синхронного взаимодействия с устройством. В случае AWS такой механизм отсутствует, и для вызова синхронных методов используется следующая парадигма:

– устройство при старте подписывается на шаблон топика “<device-name>/request/#” (где “#” по соглашениям MQTT означает произвольное содержимое);

– клиент (облачное приложение) посылает MQTT-сообщение с топиком в формате “<device-name>/request/...” (например, “<device-name>/request/sensor/list” для получения списка сенсоров или “<device-name>/request/sensor/12/get” для чтения значения сенсора);

– устройство обрабатывает запрос и посылает ответ на топик “<device-name>/response/...” (например, “<device-name>/response/sensor/list”);

– специальное правило внутри инфраструктуры перехватывает топики формата “<device-name>/response/#” и сохраняет топик и содержимое в базе данных DynamoDB с отметкой о времени сохранения;

– клиент проверяет базу данных и при появлении записи с соответствующим топиком и временем занесения использует ее как результат запроса.

Использование базы данных в рассматриваемом случае связано с тем, что для облачных приложений AWS отсутствует доступ к подписке на топики MQTT.

В случае Azure вызов синхронных методов на устройстве становится для облачных приложений намного проще и сводится к вызову соответствующего API облачной инфраструктуры. Этот API выполняется синхронно; при возврате из API приложению сразу же доступен результат вызова метода на устройстве.

Поддержка Azure direct methods на уровне устройства была реализована в общем виде и не потребовала детальной переработки каждого метода. Аналогично трансляции топиков, имя метода формируется на основе хвостовой части топика после частей “/request/” и “/response/” с заменой символа ‘/’ на ‘.’. Например, метод для получения списка сенсоров имеет имя “sensor.list”, а для получения значения сенсора с номер 12 – “sensor.12.get”.

При получении сообщения о вызове метода, имя метода транслируется в топик с добавлением “<device-name>/request/” спереди. Затем вызов метода и обычное сообщение обрабатываются общим кодом, а ответное сообщение, в случае вызова метода, транслируется в сообщение ответа на вызов метода с предварительно запомненным <rid> – уникальным идентификатором запроса.

9. ИСПОЛЬЗОВАНИЕ MQTT НАД WEBSOCKET

Протокол WebSocket [11] был разработан для прозрачной передачи произвольных данных поверх существующего соединения HTTP или HTTPS. Для этого используется механизм Upgrade протокола HTTP 1.1 [12], который позволяет изменить протокол для уже установленного HTTP-соединения. Для взаимодействия по протоколу WebSocket, клиент устанавливает HTTP или HTTPS соединение с сервером, и затем посылает запрос Upgrade, запрашивая “WebSocket” в качестве нового протокола. После согласия сервера на изменение протокола, обмен данными производится уже пакетами протокола WebSocket. Эти пакеты могут содержать произвольные данные. Протокол сохраняет границы сообщений при передаче (в отличие, например, от протокола TCP).

В случае MQTT над WebSocket, данными, которыми обмениваются клиент и сервер по протоколу WebSocket, являются MQTT-сообщениями. Основное преимущество использования MQTT над WebSocket заключается в том, что доступ к основному порту MQTT (8883) из внутренних сетей организаций часто бывает закрыт (из соображений безопасности), в отличие от порта HTTPS (443), доступ к которому открыт практически всегда. Поэтому использование MQTT над WebSocket уменьшает вероятность проблем при соединении устройства с инфраструктурой из-за требований IT-безопасности.

Обе инфраструктуры (AWS и Azure) поддерживают MQTT над WebSocket, и клиентская библиотека AWS поддерживает протокол WebSocket на стороне клиента. Существуют перечисленные далее отличия от обычного MQTT-соединения.

– в AWS, используется другой механизм аутентификации клиента, принятый для AWS HTTP REST API. А именнно, при активизации протокола WebSocket вместо сертификата клиент должен предоставить идентификатор доступа (access key ID) и секретный ключ (secret access key) для какого-либо пользователя, которому разрешен доступ к функциям IoT. Поэтому для поддержки WebSocket авторам пришлось дополнить механизм доставки IoT-конфигурации и сертификата на устройство. Дополнительно передаются идентификатор доступа и секретный ключ специально созданного пользователя AWS, имеющего права доступа только к функциям IoT. При использовании WebSocket, эти атрибуты передаются в клиентскую библиотеку при установлении соединения.

– в Azure, механизм аутентификации не меняется, но при активизации протокола WebSocket в заголовке Upgrade передаются несколько другие параметры по сравнению с AWS. Для успешной активизации достаточно, чтобы выполнялись следующие условия [13]:

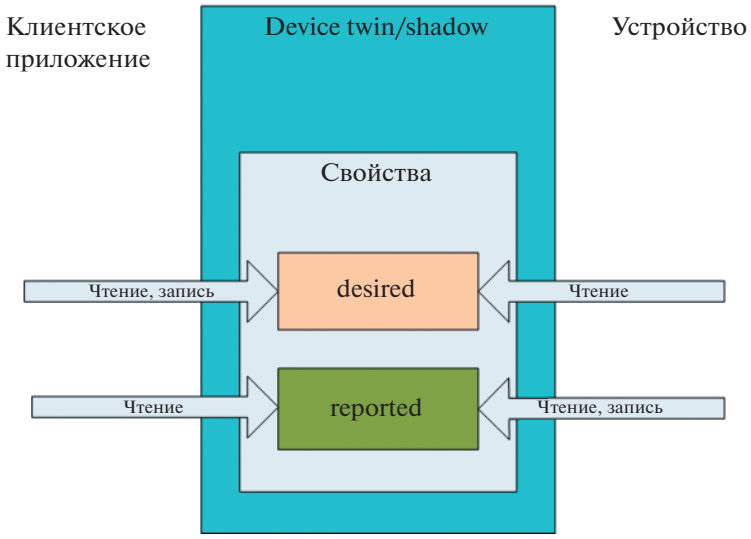

Рис. 1. Структура device twin/shadow.

• присутствуют обязательные параметры запроса: “Host:<endpoint address>”, “Connection: Upgrade”, “Upgrade: WebSocket”;

• параметр URL имеет значение “$iothub/websocket”;

• параметр “sec-websocket-protocol” присутствует и имеет значение “mqtt”.

Для поддержки Azure функция клиентской библиотеки, отвечающая за активизацию протокола WebSocket, была изменена: добавлен параметр “azure_mode”, и, в зависимости от значения этого параметра, заголовок Upgrade формируется с различными значениями параметров (для AWS или для Azure).

Решение о том, использовать ли стандартное соединение MQTT или MQTT над WebSockets, принимает пользователь при загрузке IoT-конфигурации на устройство. Соответствующий флаг записывается в IoT-конфигурацию. По умолчанию используется стандартное соединение; но если доступ к стандартному порту MQTT запрещен сетевым файрволлом, то можно использовать MQTT над WebSockets.

10. СРАВНЕНИЕ С АРХИТЕКТУРОЙ WEB OF THINGS

Архитектура Web of Things (WoT, [14]) разработана консорциумом W3C для обеспечения совместимости между различными IoT-платформами и IoT-приложениями. Это абстрактная архитектура, компоненты которой определяются следующими дочерними спецификациями:

– Спецификация Web of Things (WoT) Thing Description. [15] Эта спецификация задает стандартную модель данных для описания интерфейса IoT-устройства, в терминах свойств (properties), действий (actions) и событий (events), и в формате JSON. Приложение, работающее с устройством, может читать и записывать свойства, вызывать действия и подписываться на события (и получать их). Описание хранится вместе с устройством или отдельно от него, но доступно для считывания приложениями.

– Спецификация Web of Things (WoT) Binding Templates [16]. Этот документ описывает привязку модели данных и действий над ней к конкретным протоколам. Существуют привязки к протоколам HTTP, CoAP и MQTT. Например, в случае протокола HTTP, для каждого свойства задается URL, и чтение этого свойства отображается на операцию GET по этому URL, а запись – на операцию PUT по этому URL. Вызов действия отображается на операцию POST по соответствующему URL с определенными параметрами.

– Спецификация Web of Things (WoT) Scripting API [17]. Эта спецификация позволяет задавать на устройствах программную логику на языке Java Script, подобно тому как это делается в Web-браузерах. Это необязательный компонент архитектуры, потому что не все устройства имеют возможность содержать в себе интерпретатор JavaScript.

– Спецификация Web of Things (WoT) Security and Privacy Guidelines [18]. Этот документ содержит указания по защищенной реализации и конфигурации устройств, и рассматривает вопросы безопасности, которые требуют внимания при реализации WoT-систем.

Архитектура WoT позволяет разрабатывать облачные приложения, которые не знают заранее интерфейс устройств, с которыми они взаимо-

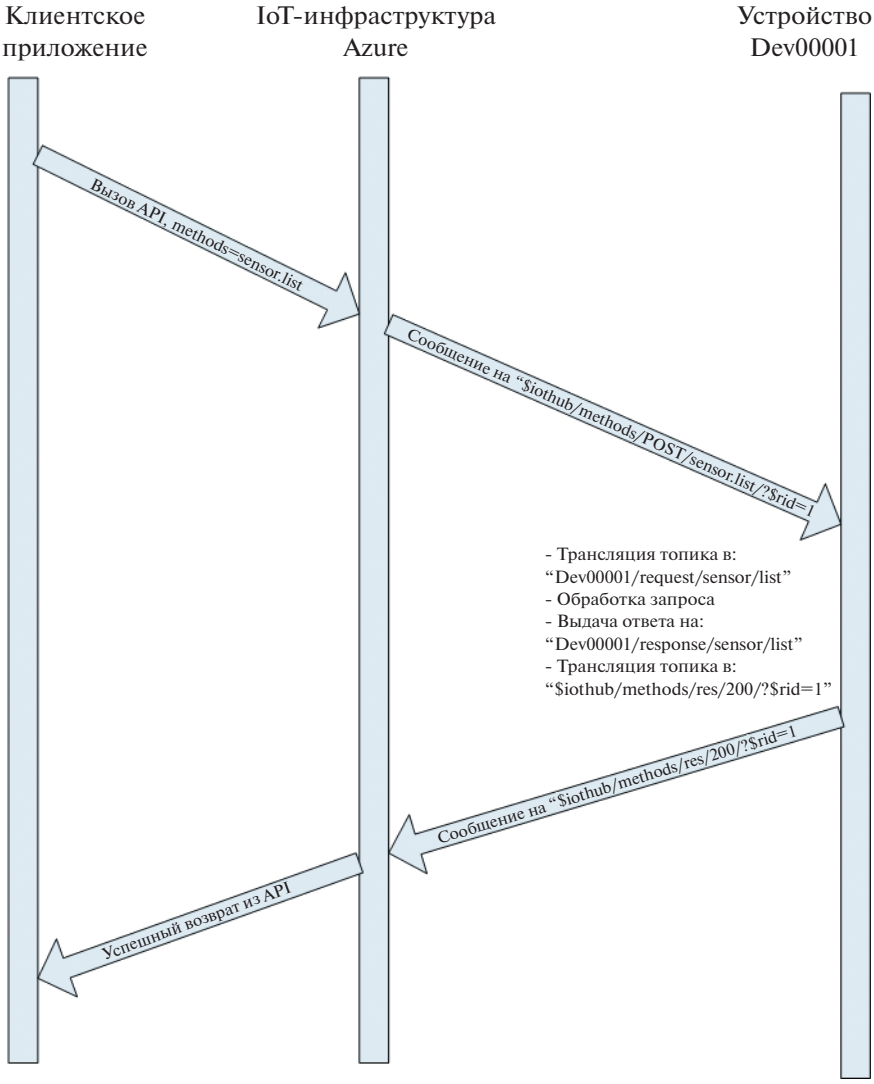

Рис. 2. Диаграмма взаимодействия при вызове direct method.

действуют – они получают эту информацию из описания интерфейса устройства. В этом смысле архитектура улучшает совместимость между приложениями и устройствами. Но, с точки зрения переносимости устройства, использующего протокол MQTT, между инфраструктурами, архитектура WoT не дает особых преимуществ, из-за следующих обстоятельств:

– Архитектура WoT основана на принципах REST API, и не очень хорошо приспособлена к модели Publish/Subscribe, на которой основан протокол MQTT; например, синхронное чтение свойств не поддерживается в этой модели. Как правило, в примерах описаний WoT-устройств с отображением на MQTT используются события и действия, но не используются свойства;

– В отображении на протокол MQTT топики статически отображаются на компоненты URL в описании. Учитывая, что в Azure множество поддерживаемых топиков сильно отличается от AWS, сделать единое описание устройства для обеих инфраструктур не получится;

– Расширения MQTT для конкретных инфраструктур (например direct methods в Azure) не поддерживаются архитектурой WoT.

Таким образом, технические решения, описанные в данной статье, актуальны и с учетом существования архитектуры WoT.

Поддержка архитектуры WoT, в частности, создание описания устройства для рассматриваемого контроллера имеет смысл как перспективный проект, в совокупности с реализацией взаимодействия с IoT-инфраструктурой на основе протокола HTTP/HTTPS, в дополнение к MQTT.

11. ЗАКЛЮЧЕНИЕ

Итак, выше рассмотрены некоторые отличия между IoT-инфраструктурами AWS и Azure, с точки зрения IoT-устройства, которое взаимодействует с инфраструктурой по протоколу MQTT. Можно сказать, что эти отличия довольно существенны, и что в каких-то аспектах имеет преимущество AWS (гибкость системы топиков), а в каких-то – Azure (direct methods). Тем не менее, при разработке программного обеспечения IoT-устройства, поддерживающего обе инфраструктуры, удалось сохранить большую часть кода независимой от инфраструктуры и локализовать зависимости в нескольких небольших интерфейсных модулях.

Разработанные технические решения можно будет использовать в будущем для подключения устройства к другим инфраструктурам, например, Google Cloud IoT. Также в перспективе – включение устройства в архитектуру Web of Things с созданием описателя устройства в соответствии с спецификацией WoT Thing Description.

СПИСОК ЛИТЕРАТУРЫ

1. MQTT Version 3.1.1. OASIS Standard. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html
2. ECMAScript® 2020 Language Specification. https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf
3. Recommendation ITU-T X.509. Information technology – Open Systems Interconnection – The Directory: Public-key and attribute certificate frameworks. https://www.itu.int/rec/T-REC-X.509
4. Alex Rodriguez. RESTful Web Services. https://developer.ibm.com/articles/ws-restful/
5. AWS IoT C++ Device SDK. https://github.com/aws/aws-iot-device-sdk-cpp/tree/release
6. Device Shadow Service for AWS IOT. https://docs.aws.amazon.com/iot/latest/developerguide/iot-device-shadows.html
7. What is Azure Digital Twins? https://docs.microsoft.com/en-us/azure/digital-twins/overview
8. Understand and use device twins in IoT Hub https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-device-twins
9. Shadow MQTT Topics. https://docs.aws.amazon.com/iot/latest/developerguide/device-shadow-mqtt.html
10. Understand and invoke direct methods from IoT Hub https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-direct-methods
11. RFC 6455. The WebSocket Protocol. https://tools.ietf.org/html/rfc6455
12. Protocol Upgrade Mechanism. https://developer.mozilla.org/en-US/docs/Web/HTTP/Protocol_upgrade_mechanism
13. MQTT-over-WebSockets needs to explain how to set up the WS connection https://github.com/MicrosoftDocs/azure-docs/issues/21306
14. Web of Things (WoT) Architecture. W3C Recommendation 9 April 2020. https://www.w3.org/TR/2020/REC-wot-architecture-20200409/
15. Web of Things (WoT) Thing Description. W3C Recommendation 9 April 2020. https://www.w3.org/TR/2020/REC-wot-thing-description-20200409/
16. Web of Things (WoT) Binding Templates. W3C Working Group Note 30 January 2020. https://www.w3.org/TR/2020/NOTE-wot-binding-templates-20200130/
17. Web of Things (WoT) Scripting API. W3C Working Draft 28 October 2019. https://www.w3.org/TR/2019/WD-wot-scripting-api-20191028/
18. Web of Things (WoT) Security and Privacy Guidelines. W3C Working Group Note 6 November 2019. https://www.w3.org/TR/2019/NOTE-wot-security-20191106/